Final

# Using Neo4J for Geospatial Data Storage and Integration

MASTER DISSERTATION

**Diamantino Romeu Garanito Ferreira**
MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

September | 2014

# Using Neo4J for Geospatial Data Storage and Integration

MASTER DISSERTATION

**Diamantino Romeu Garanito Ferreira**
MASTER IN INFORMATICS ENGINEERING



SUPERVISOR
Karolina Baras

## Abstract

Online geographic-databases have been growing increasingly as they have become a crucial source of information for both social networks and safety-critical systems. Since the quality of such applications is largely related to the richness and completeness of their data, it becomes imperative to develop adaptable and persistent storage systems, able to make use of several sources of information as well as enabling the fastest possible response from them. This work will create a shared and extensible geographic model, able to retrieve and store information from the major spatial sources available.

A geographic-based system also has very high requirements in terms of scalability, computational power and domain complexity, causing several difficulties for a traditional relational database as the number of results increases. NoSQL systems provide valuable advantages for this scenario, in particular graph databases which are capable of modeling vast amounts of inter-connected data while providing a very substantial increase of performance for several spatial requests, such as finding shortest-path routes and performing relationship lookups with high concurrency.

In this work, we will analyze the current state of geographic information systems and develop a unified geographic model, named GeoPlace Explorer (GE). GE is able to import and store spatial data from several online sources at a symbolic level in both a relational and a graph databases, where several stress tests were performed in order to find the advantages and disadvantages of each database paradigm.

## Keywords

Neo4J, MySQL, graph databases, interoperability, performance

## Resumo

As bases de dados geográficas têm crescido a um grande ritmo à medida que se tornam uma fonte crucial de informação tanto para redes sociais ou sistemas de segurança críticos. Como a qualidade de tais sistemas está altamente dependente da riqueza e completude da informação proveniente, torna-se imperativo desenvolver sistemas de armazenamento adaptáveis e persistentes, bem como capazes de ler de várias fontes de informação o mais rápido possível. Este trabalho foca-se em criar um modelo geográfico partilhado e extensível, capaz de receber e guardar informação das maiores fontes de dados geográficas online.

Um sistema de armazenamento geográfico tem também elevados requerimentos de escalabilidade, potência computacional e complexidade de domínio, causando frequentemente grandes problemas a uma base de dados relacional. Os sistemas NoSQL, em particular as bases de dados de grafos têm vindo a oferecer grandes vantagens nestes aspetos, sendo capazes de modelar vastas quantidades de dados interligados, bem como fornecendo um aumento de desempenho substancial ao lidar com cálculos geográficos, problemas de otimização e cálculo de relações – tudo isto a altos índices de concorrência.

Neste trabalho será apresentado o estado atual dos sistemas de informação geográficos, e foi desenvolvido um modelo unificado chamado GeoPlace Explorer (GE). O GE é capaz de importar e guardar dados espaciais provenientes de várias fontes de dados geográficas a nível simbólico tanto numa base de dados relacional como numa base de dados de grafos, onde finalmente foram efetuados vários testes de desempenho de modo a descobrir que vantagens e desvantagens cada paradigma nos oferece.

## Palavras-chave

Neo4J, MySQL, bases de dados em grafos, interoperabilidade, desempenho

**Contents**

## List of Figures

## List of tables



## List of code snippets

## Acronyms

ACID – Atomicity, consistency, isolation, durability

API – Application programming interface

CAP – Consistency, Availability, Partition tolerance theorem

CPU – Computer processing unit

DB – Database

GB – Gigabyte

GE – Geoplace Explorer

GIS – Geographic Information System

GPL – General public license

GPS – Geographic Position System

GUI – Graphical user interface

HTML – Hypertext markup language

IDE – Integrated Development Environment

LGD – LinkedGeoData

MB - Megabyte

NoSQL – Not Only SQL

OGC – Open Geospatial Consortium

OGDI – Open Geographic Datastore Interface

OSM – OpenStreetMap

PHP – PHP Hypertext Processor

RAM – Random Access Memory

RDBMS – Relational database management system

RDF – Resource description framework (metadata data model)

REST – Representational State Transfer (architectural style)

SPARQL – sparkle query language

SQL – Structured Query Language

UNIX – Uniplexed Information and computing service

XML – eXtensible Markup Language

# 1-Introduction

Geography has always been important to the human race. Stone-age hunters anticipated the location of their target, early explorers lived or died by their knowledge of geography, and current societies work and play based on their understanding of who belongs where. Applied geography, in the form of maps and spatial information, has served discovery, planning, cooperation, and conflict for at least the past 3000 years of our history [5].

If someone would describe a scenery, it is likely the person would categorize it into streets, fields, trees, buildings or rivers and give geographical references like 'adjacent to' or 'to the left of' to describe the location of the features. Such references can be simple or detailed, but in the end they are merely an interpretation of that person's own interests, never fully representing every aspect of reality and often inconsistent in terminology among sources. With the current pace of technology, there have been a multitude of solutions to store geographic data in its most symbolic forms, ranging from proprietary sources to extensible open-source editable maps. These tools are often called 'Geographic Information Systems', or GIS for short. As the number of geographic points and connections between these increases, the complexity begins to scale up exponentially, creating a need for powerful data representation and manipulation systems as well as new ways to integrate them with existing databases.

## 1.1 Problem domain

Geographic databases have been part of a growing industry in the past decade, and have had a vital role in many applications, such as geographically-aware search systems, route planners or recommendation systems. However, even with the latest advances in crowdsourcing and neo-geography, there is still a lack of a unified ontology that allows for automated sharing and retrieval between sources [8],[10],[13]. Since the quality of such applications is critically related to the richness of resources it becomes an important task to create powerful abstractions, able to integrate different geographical relationships between domains, or merge identical concepts into the same domain. This domain needs to be both consistent and broad enough to cover all types of geographical attributes present in all previous sources. A well-defined model is crucial

for this kind of system, facilitating its reuse and later extension. Such model should contain the following features:

a) **Support for generic data and property sets [31]**

   As the number of sources increases, the more different types of data must be dealt with, particularly of different kinds such as symbolic or geometric sets. Since many sets of information are likely to be present in one source but not in another, this leads all major classes to be extensible, generic and flexible in its meta-model;

b) **Control and lineage of information sources [31]**

   In the world of GIS, it is important to keep track both of spatial and chronological changes as well as its authors;

c) **Effective tools for data analysis**

   Geographical queries can often give us answers by finding interesting patterns across millions of nodes. This requires advanced tools in the fields of data management, analysis and output that can quickly calculate and deliver relevant data. However, geographical queries can be extremely taxing in terms of computational cost, so it's important to ensure that our system is able to handle requests as quickly as possible.

On the other hand, systems of this nature also tend to be both complex in structure and sizeable in terms of stored data. For several decades, the most common data storing model has been the relational model, which is implemented under relational database managing systems (RDBMS), with the common 'relational database' term being often used in its place. While powerful, RDBMS also have important limitations that can hinder the growth and manipulation of large sets of data. Such limitations can include:

a) **Up-scaling**

   Whenever a relational database needs more storage, it must be loaded on a more powerful computer. When it does not fit a single computer, it needs to be distributed across multiple computers but relational databases are not designed to

function with distributed systems, resulting in a decrease of performance when combining multiple tables;

**b) Schema-less data**

It is often difficult to fit schema-less data into relational databases, and this is often a problem when trying to combine several sources of data. RDBMS are substantially more effective when manipulating simple, structured data, but other alternatives can provide interesting results.

This thesis will focus on creating a unified world model, able to support several data structures and will also explore new possibilities in terms of data storage with a non-relational DBMS approach.

### 1.2 Research questions

The problem description above leads us to the following research question:

- **How do relational and NoSQL datastores compare in terms of performance to support an integrated system receiving input from several geographical sources?**

This thesis aims at developing an integrated spatial model, able to acquire results from both manual input as well as from several online geographical databases. This spatial model is then stored in two different datastores (relational and NoSQL), where a series of stress tests is performed to compare the two alternatives.

This thesis presents the following deliverables:

I. Design and implementation of both relational (MySQL) and graph (Neo4J) database models for an identical data set;
II. PHP based implementation to manually insert, update or import sets of data from several online geographical databases (Factual, GeoNames and OpenStreetMap) into both databases;
III. Scientific reports for all related literature research, as well as development and documentation for the proposed implementation, also covering an analysis on performed tests.

### 1.3 Methodology

In short, this research started by presenting an overview of the current geographical information systems, the features they offer as well as their progress, state of the art and limitations. Next, we analyze several different online geospatial databases, how they store and organize data and present the strengths and weaknesses of each other. The following step focused on finding new ways to model such data by using a different approach to relational databases. We then performed a quick study about NoSQL systems in general, with particular emphasis in graph databases which seemed to adapt to our problem more fluidly. After choosing an appropriate solution (Neo4J), we then explore its implementation possibilities to create a web application, able to retrieve, model and consolidate spatial data gathered from geographical sources into both relational and graph models. The final step consisted in performing several stress tests to find out and discuss which advantages each different approach brings to our problem.

### 1.4 Structure

This thesis consists of 5 chapters organized in the following fashion:

1. Chapter 2 describes the current state of geographical location systems, their main advantages and limitations and how they can be more effectively represented, shared and accessed from one database to another;
2. Chapter 3 of this work describes some of the current NoSQL databases, and how they can be used to model sets of spatial data. A small introduction to Neo4J is presented, as well as its own graph query language (cypher);
3. Chapter 4 documents all implementation aspects (class model, dependencies, wrappers, …) for our application, a simple GUI in PHP to manually insert as well as import and integrate data from several online geographical databases;
4. Finally, chapter 5 describes all tests performed with our dataset, and compares them with previous tests made to large sums of spatial data with the intent to benchmark performance between each database system used in the project.

# 2-Literature Review

In this chapter we will present a brief overview of online geographical databases, their evolution in the past years, the advantages they brought in several domains and their current state of the art. Some practical examples will be provided and certain tools will be analyzed in more detail. We conclude by exploring several projects that improve the data gathering process and the limitations they face.

## 2.1 Geographical Information Systems

A geographical information system (GIS) is a technology for performing spatial analysis and building all kinds of geographical maps [1], which are organized in layers and represent physical elements trough nodes, lines and polygons. GIS and spatial analyses are concerned with the measurable location of important features, as well as properties and attributes of those features. These systems let us visualize, question, analyze, interpret and understand data to reveal relationships, patterns and trends [2]. GIS offer us a new way to look and explore the world around us. Accurate geographical mapping allows for several benefits, such as better decision making in several areas (real estate selection, traveling), improved communication, better recordkeeping for organizations as well as increased efficiency for terrestrial movements.

Such information systems can also be used for integrating, storing, editing, analyzing, sharing and displaying all kinds of geographical information. The first known use of the term "Geographic Information System" was by Roger Tomlinson in the year 1968 in his paper "A Geographic Information System for Regional Planning". Tomlinson is also acknowledged as the "father of GIS".

GIS are usually composed of four stages:

1. Data input from maps, aerial photos, satellites, surveys and other sources;
2. Data storage, retrieval and query;
3. Data transformation, analysis, and modeling including statistics;
4. Data reporting such as maps or plans [4].

GIS, with its array of functions, should be viewed as a process rather than as merely software. One of the main purposes is to facilitate the decision-making progress. The way in which data is entered, stored, and analyzed within a GIS must mirror the way information will be used for a specific research or task [1,3]. In other words, a GIS is both a database system with specific capabilities for spatially-referenced data, as well as a set of operations for working with data. In a sense, a GIS may be thought of as a higher-order map.

### 2.1.1 Advantages

The main appeal of GIS stems from their ability to integrate and share great quantities of information about the environment and to provide a powerful repertoire of analytical tools to explore this data. The ability to separate data in layers, and then combine it with other layers of information is the reason why GIS hold such a great potential as research and decision-making tools. [1] Delivering a common data view to agencies, process partners and stakeholders improves communications across organizations, leading to improved decision making and efficiency.

GIS have also emerged as very powerful technologies because they allow geographers to integrate their data and methods in ways that support traditional forms of geographical analysis, such as map overlay analysis as well as new types of analysis and modeling that are beyond the capability of manual methods. With GIS it is possible to map, model, query, and evaluate large quantities of data all held together within a single database. Not only it becomes a consolidated approach to data retrieving, storing and visualization, but it also expands across several areas, such as cartography, surveying, civil engineering and demography [5]. Several systems already combine information regarding transportation networks, hydrography, population characteristics and economic activity – all of these can be useful for urban planning, environmental resource management, emergency planning or transportation forecasting [30].

GIS-based maps and visualizations greatly assist in understanding situations and prediction. They are a type of language that not only improves communication between different teams and departments [3], but can also cut down costs in evaluation and record-keeping activities.

### 2.1.2 Evolution

Web mapping systems have gained substantial popularity in the last 15 years, and have been rapidly expanding since the release of OpenStreetMap back in 2004, followed by Google Maps in 2005, Wikimapia in 2006 and Microsoft Silverlight in 2007. Latest releases include improved navigation, Google Fusion Tables, Google Maps Mobile and Wikitude Drive (2010) [2,11].

It is important to note that map coverage did not only become more widespread in recent years, it has also became significantly more accessible (and collaborative) to the average user. Early maps that were restrained to commercial and data providers via satellite images rapidly expanded to the combination of road networks, panoramic street-level imagery such as Google Street View and massive collaborative systems such as OpenStreetMap. Geographical Information Systems represent a jump from paper maps like the computer from the abacus [23]. Within the last thirty years, GIS technology has evolved from single purpose, project based applications to massive enterprise systems [36]. Such systems are currently being used by businesses, institutions, industry, local governments and the private sector to provide services to clients, manage resources, and to address multiple issues concerning to health and human resources, transportation, public safety, utilities and communications, natural resource, defense and intelligence, retail and many more.

With such a development, new concepts and technologies have arisen over the years. 3D Desktop applications, detailed and interchangeable layers, public data, map makers, map integration with social networks and mobile mapping have also appeared. The existing services have attracted several millions of users, both desktop and mobile. The scope of web mapping applications has widened from purely easy to use consumer-oriented tools to highly specialized applications with GIS functionalities that help solving and optimizing problems in several domains. Despite the advances of web mapping within the last few years, there is still a lot of potential to collaborate, to elaborate, to tell stories with creative new methods and to use the data in useful and interesting new ways [11].

## 2.2 Components

A GIS is comprised of hardware, software, data, humans, and a set of organizational protocols. These components must be well integrated for effective use of GIS, and the development and integration of these components is an iterative, ongoing process [5]. Hardware for GIS usually requires a fast computer due to how often spatial analyses are applied over large areas or at high spatial resolutions. This means that even simple operations can take a long time to compute as they have to be repeated tens of millions of times. Another important aspect is the storing of large volumes of information. GIS software provides the tools to analyze, calculate and display the spatial information. This encompasses functions such as data entry (manual or digital coordinate capture), data management (subsets, merging data, indexing, data projection), data analysis (spatial queries, interpolation, connectivity, proximity, buffering) and output (map design and layout, digital map serving, metadata output or digital graphic production).

There are several available GIS software packages, such as ArcGIS, GeoMedia, AutoCAD Map, QGIS and GRASS, however these are out of scope for this work and will only be mentioned briefly when relevant.

## 2.3 Online Geospatial Databases

Presently, vast quantities of geospatial data and information at local, regional and global scales are being continuously collected, created, managed and used by academic research, the public for spatial decision support and location based services [37]. A key aspect to any geospatial solution is to support efficient data maintenance and analysis in a heterogeneous operating environment. This requires highly scalable, highly available and secure database management systems. One of the biggest challenges is integrating time into database representations, another is integrating geospatial data sets from multiple sources (often with varied formats, semantics, precision and coordinate systems) [38].

Current GIS, which are optimized to store and query data that represents objects defined in a geometric space, often utilize online geospatial databases as sources of their data. Most spatial databases allow representing simple geometric objects such as points, lines and polygons [24] and can operate with varied data structures, queries, indexes and algorithms. These can support several operations such as spatial measurements

(computing line length and distances between geometries), spatial functions, geometry constructors and observer functions (queries which return specific information regarding features such as the location of the center of a circle). They also allow remote querying of results via an API or even integration with other existing geo databases (such as the LinkedGeoData initiative).

Some examples of online geodatabases include OpenStreetMap's project [44], the GeoNames project [28], and Factual [26]. The main difference between these is that OSM and GeoNames are both open-source projects, accepting data from thousands of users while Factual usually only maps out missing attributes or adds new ones. OSM features the largest collection of data (over 1 billion nodes), which surpasses other services in this regard.

### 2.3.1 Factual

Factual, launched in October 2009, is an open data platform developed to maximize data accuracy, transparency, and accessibility. It provides access through web service APIs and reusable, customizable web applications [25]. Factual features data sets about local place information, entertainment and information derived from government sources. At the current date and according to their website, Factual contains over 65 million places of data, which are updated and improved in real-time by Factual's data stack. Factual's API allows for remote access to stored data, through the use of queries. Access to the API server must be requested first with an oauth API key and secret, and has existing frameworks for several programming languages such as C#, Java, PHP, Phython and Ruby. Results can also be obtained through HTTP GET requests. It is possible to query several types of information, such as text searches, interest points, direct and reverse geocoding, geo filters, data submission and matching queries with provided information [26]. It can also clean and resolve data as results are submitted into the database, and afterwards connects it to other sources of factual data, or external sites (such as Facebook or Twitter) relating to the same coordinates, and further enhances this by delivering contextually relevant content, experiences and ads based on where mobile users are located.

A brief example on how to implement a factual API call is provided in Code snippet 1. It is important to create an authenticated handle to Factual so it can configure

```
    $query = new FactualQuery;
    $query->within(new FactualCircle(32.659936, -16.926097,
    1000)); //5 km radius
    $query->limit(10); $query->sortAsc("\$distance"); $res =
$factual->fetch("places", $query); print_r($res->getData());
    $res = $factual->reverseGeocode(-16.926097,32.659936);
print_r($res);
```

Code snippet 1- Example factual query.

class loading on instantiation. The query displays all nearby points of interest near the University of Madeira, sorted by distance.

### 2.3.2 GeoNames

GeoNames is another global geo-database initiative. According to their website [27], the GeoNames database contains over ten million geographical names corresponding to over 7.5 million unique features. All features are categorized into one out of nine feature classes and further subcategorized into one out of 645 feature codes [28]. Beyond names of places in various languages, data stored includes latitude, longitude, elevation, population, administrative subdivision and postal codes. However, implementing its own semantics schema can be a barrier to future interoperability with other geospatial databases and sources of information [8]. It operates on a low level, semantic schema with no set of constraints on the domain and range of attributes, which can be a barrier to the consistency of the data set. GeoNames also features a functional API call server, which allows for querying of information on a programming level in several programming languages (such as C# or PHP). Results can be returned via XML or JSON objects. GeoNames API allows for full-text based searches, postal code search, placename lookup, nearby postal codes, reverse geocoding, nearby populated places as well as providing other services such as recent earthquakes or weather status lookup [29].

For example, the query in Code snippet 2 retrieves all locations named ‘Funchal’, as well as their coordinates and corresponding country from a text search:

```
    $geo = new Services_GeoNames();
    $cities = $geo->search(array('name_equals' =>
'Funchal'));
    echo "Funchal::\n";
    foreach ($cities as $city) {
    printf(" - %s (%s) %s,%s\n", $city->name, $city-
>countryName, $city->lat, $city->lng);
    }
    echo "\n";
```

Code snippet 2 - GeoNames query that returns all places named 'Funchal'.

### 2.3.3 OpenStreetMap

OpenStreetMap (OSM) is a collaborative geospatial project that aims at creating a free editable map of the world created by Steve Coast in UK in 2004. Since that point, it has experienced a growth up to 1 million users (on 2013) who collect all kinds of high quality data in various ways, such as GPS devices, aerial photos, sea travels [33], and government files. Like most internet projects, most users are casual or inactive, with a small dedicated minority contributing the majority of additions and corrections to the map. The database now contains over 1 billion nodes, 21 million miles of road data and 78 million buildings [34]. OSM features a dynamic map where every urban or natural feature is built by the community, resulting in accurate, high quality data representations [13].

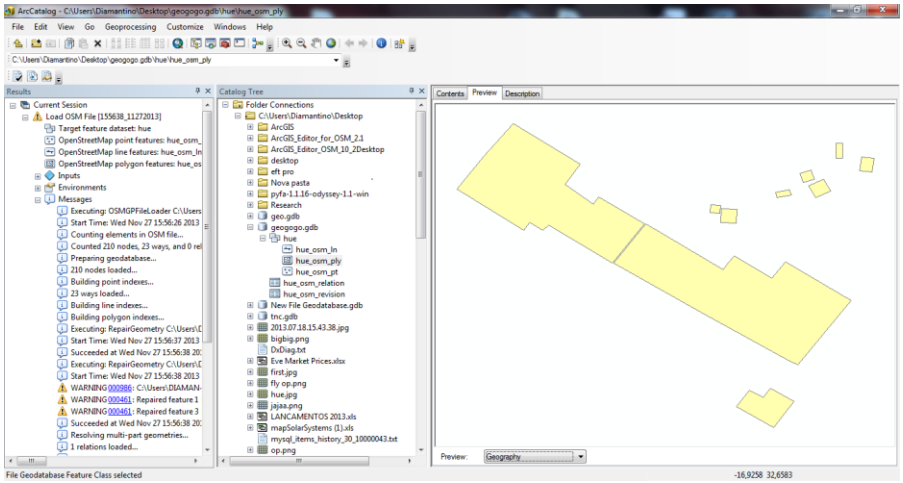

Figure 1 - Symbolic and Geometric layers in ArcMap.

The OSM project facilitates complete, regularly-updated copies of its database, which can be exported and converted to several GIS application formats through various frameworks. The data is stored in a relational database (Post-greSQL backend) which can be accessed, queried and edited by using a REST API, which uses HTTP GET, PUT and DELETE requests with XML payload. Data collection is acquired by GPS traces or by manual map modeling. Figure 1 shows a representation of OSM's symbolic data gathered from the XAPI and converted to ArcMap featuring the peripheral zone of University of Madeira.

OSM uses a topological data structure with four data primitives: nodes (geographical positions stored as coordinates), ways (orders lists of nodes) which can represent a polygon, relations (multipurpose data structure that defines arbitrary relationships between 2 or more data elements [45]) which are used to represent turn restrictions in roads, and tags (arbitrary metadata strings) which are mainly used to describe map objects. Each of these entities has a numeric identifier (called OSM ID) and a set of generic attributes defined by tags. For example, the natural tag describes geographical features which occur naturally, which has a wide set of values {bay, beach, save entrance,…wetland, wood} [13] Further tags are used to specify time zones, currencies [32] and alike.

According to previous studies [12], the volunteered geographical information submitted to OSM is fairly accurate, with more than 80% of overlap between other specialized datasets, and often with more correct references in several countries around the world.

### 2.3.4 LinkedGeoData

LinkedGeoData is an effort to add a spatial dimension to the Semantic Web. It utilizes the information collected by the OpenSteetMap project and makes it available in RDF knowledge. [35] The Semantic Web eases data and information integration by providing an infrastructure based on RDF ontologies, which are interactively transformed from OSM's data. This procedure is believed to simplify real-life information integration that require comprehensive background knowledge related to spatial features. [32] LinkedGeoData offers a flexible system for mapping data to RDF format, improved REST interface and direct interlinks to GeoNames and other geospatial data projects. Figure 2 presents an overview of LinkedGeoData's structure:

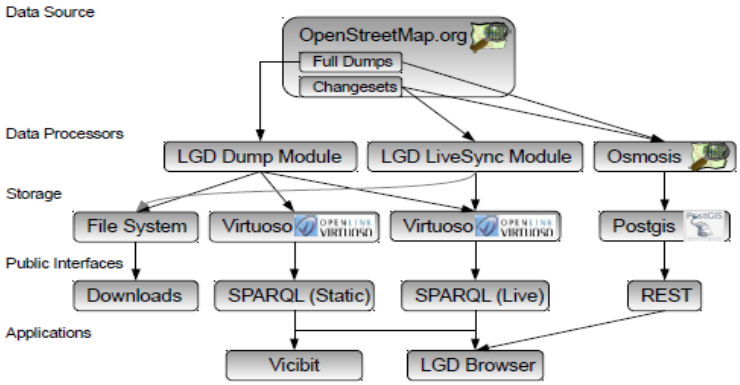


Figure 2 - LinkedGeoData's architecture, from low to high level entities.

The data acquired from OSM is processed in different routes. The LGD Dump Module converts the OSM planet file into RDF and loads the data into a triple store. This data is then available via the static SPARQL endpoint. The LGD Live Sync Module monitors and loads change sets to the RDF level in order to update the triple store accordingly. The Osmosis is a community developed tool, which supports setting up such a database from a planet file and applying change sets to it. For data access, LGD offers downloads, a REST API interface, Linked Data and SPARQL endpoints [32]. The REST API provides limited query capabilities about all nodes of OSM.

LGD does not only contain linked data to OSM, but to other data services as well. It currently interlinks data with DBPedia and GeoNames [32], which is done on a per-class basis, where all instances of a set of classes of LGD are matched with all instances of a set of classes of other data sources using labels and spatial information. Since a vast amount of data is changed on OSM every few minutes, several filtering and synchronization procedures are used to ensure the data is kept relevant. Converting relational databases to RDF is also a significant area of research that LDG heavily depends on. These enhancements may further contribute to new semantic-spatial search engines to enable more linked data applications, such as geo-data syndication. However, certain limitations are still in the way of a more robust and scaling service, such as a lack of aggregating ontologies between OSM and the filtering system.

### 2.4 Crowdsourcing

A recent trend in neogeography has emerged by complementing information to geo-databases via several outside sources, namely social networks [9], shared media websites, and user's GPS contributions, known as volunteered geographical information

(VGI). The potential of crowdsourcing is that it can be a highly exploitable activity, in which participants are encouraged to contribute to an alleged greater good. For example in Wikipedia, well over 99.8% of visitors to the site do not contribute anything [12], yet this does not deter the contributors from doing most of the work

Since geo spatial databases typically contain vast amounts of data, it is important to create mechanisms that can assist and verify the user-generated data submissions, which aim at improving internal data quality and resolving data conflicts between sources. These conflicts can be identified at the syntax (representation), structure (inconsistency), semantic (ambiguity) and cartographic (accuracy) levels [7].

Crowdsourcing in GIS brings the following advantages:

- Make datasets accessible by non-proprietary languages and tools;
- Introduce formal semantics to make data machine learning processable, so the process can be automated between sources;
- Enrich existing data with other sources in a combined way, therefore increasing location precision;
- Allow cross-referencing and querying of multiple data sources simultaneously;
- Automatically detect on-going events and new places of interest from media and social networks.

However, there are still several obstacles in this approach: data conflicts must be carefully managed, and proper conversion methods must be employed to ensure consistency (e.g different names and tags for the same places in different sources), as well as other mechanisms to verify data integrity from user submissions (data validation). Outdated sources of information must also be filtered. A layer structure by M-PREGeD has been proposed [7] to improve data quality by involving the users in the selection process. This practice adopts the use of shared vocabularies and links to help increase semantic consistency and accessibility among sources of data.

In this case, results are stored in a local database and further enriched with matching Linked Geo Data nodes, which are verified and corrected by the Linked

corrections layer, and only after that become available to the web applications layer. The user-generated matches are acquired by several applications, such as UrbanMatch [7] which is a game with a purpose for matching points of interest and photos, running on mobile geo-located services. The player is given a set of photos and is asked to match them with each location. The output is gathered from the players, verified and correlated and finally used to generate trusted links. A high number of similar answers are used to identify patterns of information and determine the accuracy of provided data. This process can be defined in 3 layers: the linked geo data layer which contains unprocessed data from several sources, the linked connections layer that is composed of all verification methods, such as the UrbanMatch game. Finally, this data is submitted to the web applications layer to the general audience. Figure 3 illustrates this design.

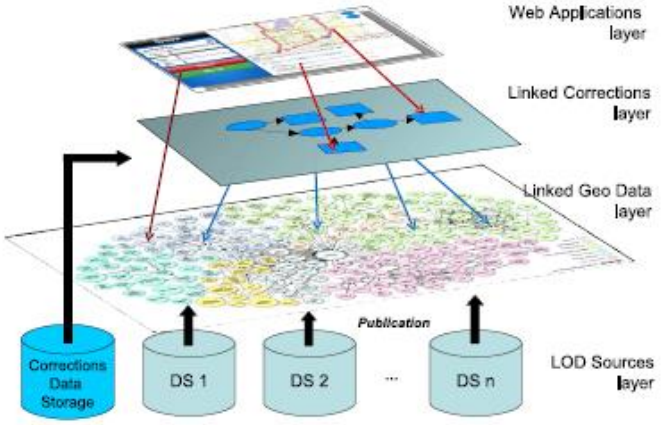


Figure 3 - Positioning of the correction layer.

.

Another process is done by analyzing metadata and tags from photos over different social media websites (such as Flickr and Twitter), by discovering associations between the media and the owner's geographical position, and is also possible to populate geo-sets with news data collected from other sources by employing extraction and retrieving systems [39].

These experiments not only help increase spatial accuracy for existing places, but can also be used to discover new points of interest. Conflicts can be detected and resolved by automatic procedures, and expanding these techniques is an on-going

process, especially when certain errors can occur [9] due to old or mislabeled data, incorrect semantic relationships, or simply by geographic evolution overtime. For example, there is a high correlation between places of interest like restaurants and a number of photos containing tags such as “food”, “dinner” and “eating” being uploaded on the same location. This quantitative evaluation consists of very specific statistical and data mining methods with several levels of precision. In recent studies [9] several new places in London were found with the help of these methods that were not present in GeoNames or Foursquare. Since different database providers (e.g Geonames, Foursquare) are constructed in different ways, this procedure takes into account how different places are submitted and represented internally, and tries to aggregate similar results into a location vector, which creates a classifier that calculates the likelihood that a given location contains a place in particular. Closer examinations detected conflicts or mislabeled data such as misleading Tweets and incorrect or semantically ambiguous tags, and this allows excluding potential outliers based on standard deviations in the dataset. This method was able to find new places of worship, schools, shops, restaurants and graveyards that were not yet present in LinkedGeoData and GeoNames. With finer iteration it becomes possible to detect new places of interest with higher precision, such as extending OpenStreetMap to indoor environments [14]. This approach aims at increasing the detail of geo data by adding new tags to inner structures, such as airports, museums or train stations by employing a 3D building ontology. Again, such data is likely to be supplied voluntarily by individuals at those places of interest. This kind of non-profit projects greatly enhances user participation.

### 2.4.1 Reliability

The proliferation of information sources as a result of networked computers has prompted significant changes in the amount, availability, and nature of geographic information. Among the more significant changes is the increasing amount of readily available *volunteered* geographic information (VGI), which is produced by millions of users. However, as many of these results are not provided by professionals but by amateur users, and therefore do not follow the common standards in terms of data collection, verification and use, this creates an issue which is very frequently discussed and debated in the context of crowdsourcing activities. Several studies [12, 40, 41 ,42]

have analyzed the systematic quality of VGI in great detail, and claim that for the most part results can be fairly accurate compared to other private or governmental entities, and represent the move from standardized data collection methods to data mining from available datasets.

Evaluating the quality of geographical information has received the attention of surveyors, cartographers and geographers many years ago, which have carefully deliberated a number of quality standards [12]:

a) Lineage: the history of the dataset
b) Positional accuracy: how well the coordinate of an object is related to reality
c) Attribute accuracy: how well an object is represented with tag attributes
d) Logical consistency: the internal consistency of the dataset
e) Completeness: measuring the lack of data for a particular object
f) Semantic accuracy: making sure an object's representation is correctly interpreted
g) Temporal quality: the validity of changes in the database in relation to real-world changes and also the rate of update

Several methodologies were developed to quickly determine the data quality, from statistical comparison (using average standard deviation between different databases to calculate positional accuracy, object overlap percentages, tile boards, road length comparison among maps [42] and spatial density among urban areas to more empirical methods (visual density representations, road comparison). The results state that there is an approximate overlap of 80% of motorway objects, roughly located within about 6m of distance recorded between OpenStreetMap and other proprietary map types in London, as well as less than 10% of total road length difference between OSM and TeleAtlas [42].

### 2.5 Interoperability

The growth of geospatial industry is stunted by the difficulty of reading and transforming suitable spatial data from different sources. Different databases have unalike conventions, classes, attributes and even entirely different semantic structures,

which makes it difficult to interconnect and correlate data from different sources into heterogeneous data [15]. As billions of dollars are invested worldwide in the production of geospatial databases, it becomes imperative to find other alternatives for data translation and open, collaborative frameworks. As most GIS use their own proprietary format, translating this data into other formats can be a time consuming and inefficient process. Transformations between formats can also result in loss of information because different formats support different data types. Data translation also takes up a large amount of secondary storage and can be costly to develop. Other providers sometimes restrict access by repackaging products for a particular GIS.

As such, there have been several solutions in the past to alleviate the issue, such as object oriented open frameworks and open architectures. Some examples are OGIS (Open GIS consortium) and DGIWG (Digital Geographic Information Working Group).

Other approaches consist in interpreting other formats through the use of a universal language (such as OGDI, the Open Geospatial Datastore Interface). As the translation process is a long and difficult progress, OGDI is based on reading different geospatial data formats directly, without translation or conversion [10]. In essence, it works as a comprehension tool instead of translation. This allows for different formats to be queried from a uniform data structure, as well as transparent adjustment of coordinate systems and cartographic projections. Such data structures can be implemented via simple object oriented concepts, such as point features instantiating a coordinate, line features being composed of 2 or more coordinates in the same directional vector, and area features consisting of several line features forming a closed ring. This transient vector structure is the backbone of the retrieval functions, and allows for transparent adjustments and understanding of multiple coordinate systems. OGDI also uses an API which can validate parameters and sequences, transform coordinates and projections and provide an entry point to OGDI functions for each driver.

In this chapter we have presented the main advantages as well as obstacles present in the world of neo-geography. Geographic representations can be either symbolic (categories and arbitrary distinctions between objects) or geometric (where the data tries to emulate spatial features as accurately as possible). As seen above, it is of

particular interest to develop more advanced crowdsourcing and interoperability methods in order to obtain and maintain larger amounts of data.

# 3-NoSQL Fundamentals and Neo4J overview

In this chapter, we will present an overview of current NoSQL databases and compare the underlying strengths between each other to choose the most fitting solution (Neo4J) and describe its basic functionalities as well as its own query language (Cypher).

## 3.1 The NoSQL movement

"As the popularity of data virtualization continues to rise, companies are increasingly relying on data storage and retrieval mechanisms like NoSQL to extract tangible value out of the voluminous amounts of data available today." [52]

Many different kinds of data models are labeled as 'NoSQL', which stands for not-only SQL. This provided a mechanism for both storage and retrieval that is modeled in a different way compared to relational databases management systems (RDBMS). NoSQL databases are becoming increasingly popular in big data and real-time web applications [44], such as Google's BigTable and Facebook's Cassandra as they are designed to support different principles than its relational counterpart [47], such as horizontal scaling (adding more nodes to a system), ability to work with schema-less data or better control over availability. Full transactional guarantees and simultaneous completion of transactions are not always provided by NoSQL databases, however this is prioritized over data availability instead (by internal synchronization) [52]. The following table 1 presents the main differences between NoSQL and RDBMS:

| **Feature** | **NoSQL** | **RDBMS** |
|---|---|---|
| **Data volume** | Huge volumes | Limited volumes |
| **Scalability** | Horizontally | Horizontally and Vertically |
| **Query languages** | Several (depends on DB) | SQL |
| **Schema** | Not predefined | Predefined |
| **Data types** | Unstructured, unpredictable | Structured relational data |
| **Principles** | CAP (Consistency, Availability, Partition Tolerance) | ACID (Atomicity, Consistency, Isolation, Durability) |
| **Transaction management** | Weak | Strong |

Table 1- Main differences between SQL and NoSQL databases.

.

However, NoSQL systems tend to be highly specialized solutions, often creating a compromise between other existing features in an SQL environment (such as true ACID* transactions or consistency in favour of partition tolerance [45]). Normally, NoSQL systems are only employed when a RDBMS approach fails to meet their intended requirements.

NoSQL databases can operate with several types of data [52]. These can either be a Key-Value store (a large hash table of keys and values), Document-based store (stores documents made up of tagged elements), Column-based store (each storage block contains data from only one column) and finally Graph-based storage, where a network database uses edges and nodes to represent and store data. Table 2 presents an overview of each different database.

| **NoSQL database** | **Description** |
|---|---|
| **Key-value** | Consists of pairs of keys and values. The key value uses a hash table with a pointer to certain data. The cache mechanisms enhance performance through the mappings. Strong availability but lacks in consistency of data. Examples: Amazon's Dynamo [53]. |
| **Document store** | The collection of key values is compressed as a document (encoding structure). Contains attribute metadata with stored content in formats such as JSON. A strong approach to schema-less data, but lacks in consistency, considered the next level key/value. Examples: Apache CouchDB, MongoDB. |
| **Column store** | Organizes data into column families that can be created during runtime at the schema definition. This can process very large amounts of distributed data over many machines. Keys point to multiple columns, arranged by column family [54]. Examples: Cassandra, HBase. |
| **Graph database** | Instead of tables and rows and the rigid structure of SQL, graph databases use a graph model that can be morphed and adapted as needed. They operate with paths of data instead of sets of data like SQL. Examples: Neo4J, Infinite Graph |

Table 2 - Description of each type of NoSQL database.

Graphs are a natural solution to several of our everyday problems, especially when dealing with a high number of interconnected data, complex geographical queries, hierarchical data recommendation systems and shortest-path operations. They facilitate the usage of schema-less data and can scale up to several billions of nodes per machine. Unlike document and column stores, they are also able to maintain a consistent state in the database as every kind of relationship is stored physically and not sorted by arbitrary categories.

## 3.2 Graph database fundamentals

This section will describe the fundamentals behind graph operations, the underlying model for graph databases. A brief explanation of their data structures and basic operations will be presented, as well as some implementation and performance details.

### 3.2.1 Graph theory

Graphs are mathematical structures used to model relationships between objects, made by sets of V vertices and lines called edges that connect them. Their origin is attributed by many to the pioneer Swiss mathematician Leonhard Paul Euler (1707-1783) and the famous 'Köningsberg bridge' problem, which revolutionized the way many complicated problems were looked at and solved in future. Nowadays, graphs can be used to model many types of relations and processes in physical, biological, social and information systems [43]. Many practical problems can be easily represented by graphs.

A graph, in its most simple form can be represented mathematically as:

$G = (V,E)$ where $V = v1,v2,v_n$ and $E = e1,e2,e_n$

This structure may encompass loops (two endpoints of an edge are in the same vertex). Different types of graphs can be created by adding characteristics to the primitive dots and lines. For example, we may assign a label to each vertex or assign a name to each line. Graphs may also be directed (with a specified one-way orientation between edges), weighted (where each node has its own weight, useful for probabilistic analysis) and semantic (where it models a cognitive structure such as relationships of concepts, like a graph of friends).

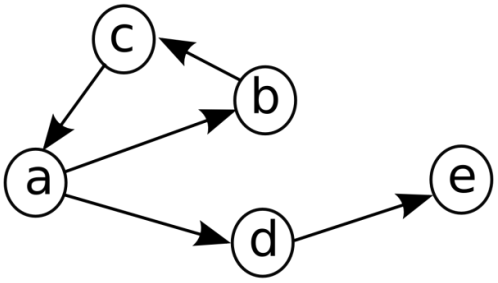


Figure 4- A simple directed labeled graph

In computer science, vertices can be seen as abstract data types also called nodes, with lines (segments) connecting these nodes. Again, more abstractly saying edges can be seen as data types that show relations between nodes. Most databases support a mix between directed, labeled and attributed and multi-graphs, as it allows us to construct complex data structures by simply adding or removing characteristics from the graph.

This morphism of properties can more fluently represent our problem domain from its lowest level [54].

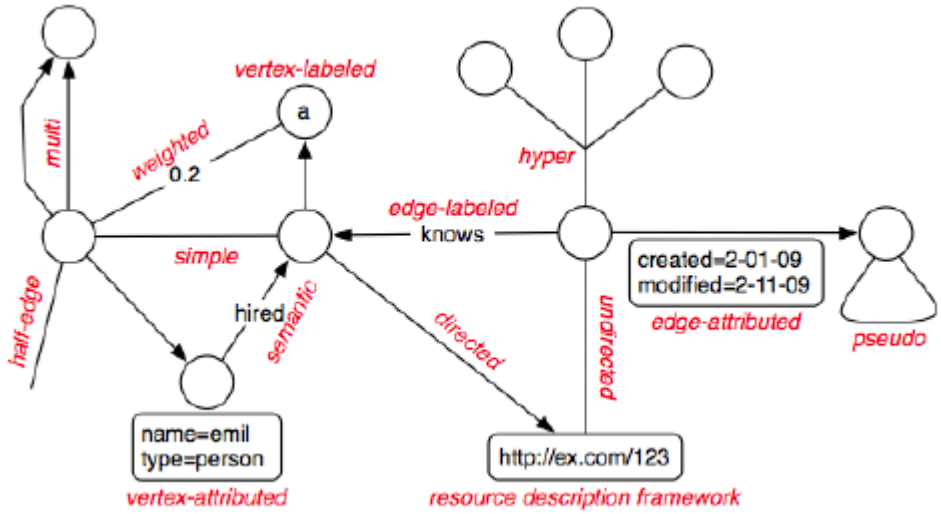


Figure 5 - An example of graph morphism. Several characteristics can be added to a graph to better express our domain.

Graphs remain a significant field of interest within algorithms and data modeling. Typical higher-level operations associated with graphs (such as finding a path between nodes, a depth-first search or a shortest path operation) are more intuitive to solve compared to other representation methods (such as the relational model) and present substantial improvements in response times.

### 3.2.2 Graph databases

Graph structures store relationships at the individual record level, while in a relational database the structure is defined at a higher level in table definitions and foreign keys. Relational databases work with sets while graph databases work with paths. This has very important ramifications later on, leading to some operations being faster in a RDBMS and others being faster in a graph database. For example, when trying to emulate path operations (e.g. friends of friends) by recursively joining tables in a relational database, query latency tends to grow exponentially and massively along with memory usage. Most graph databases don't suffer this kind of join pain because they express relationships at a fundamental level, and therefore are much faster than a relational database to express connected data – the strength of the underlying model. A consequence of this is that query latency in a graph database is proportional to how much of the graph we choose to explore in a query and not proportional to the amount of data stored in total. Relational models work significantly faster when the structure of the data is known ahead of time. They also use less storage space as they do not have to physically store every relationship on disk.

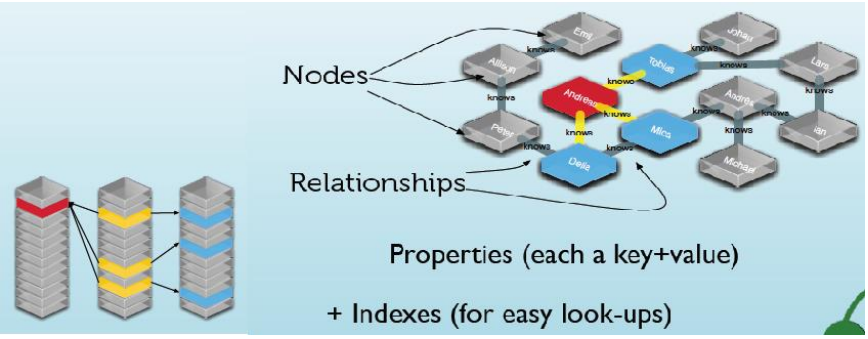


Figure 6 - The main differences in representing data for both a relational (left) and graph model (right) [55].

Graph databases store data in acyclic graphics, the most generic of data structures. They record data in nodes (which may have properties), and directed relationships between nodes (with possible attributes), which describe paths as seen previously. A graph can be navigated with a traversal, which essentially means traveling a sub-graph from node to node via a particular algorithm to find the desired data. Graph queries work by searching patterns such as sets or paths of data. Typical questions for a traversal are:

"How is A connected to B?" (set) or "Where's the nearest point of A?" (pattern). Indexes can also map from properties to nodes or relationships. They are used to optimize traversal queries that involve node or relationship properties, as it would be inefficient to transverse an entire graph to look for a particular propriety [46].

Graph databases are also particularly good at dealing with domain complexity. Highly interconnected data structures are easier to model in a graph, and complex queries with result-set data coming from different entities are more efficiently executed. However, they offer little gain in terms of set-oriented queries, where all data in our queries needs to be represented as a long list of uniform, tabular data items. Large chunks of binary data can be stored in a graph, but is often not a good idea given that putting binary data in a node can significantly impact transversal speeds.

Since graphs are a naturally data-driven structure, they become a flexible solution for several (schema-free) domain models, typically those that involve highly connected data, pattern matching, shortest path algorithms, and optimization.

As such, we can conclude that graph databases are better suited for irregular and complex structures, and boast strong vertical scalability while relational databases are faster to perform queries over regular and relatively simple structures.

### 3.3 The Neo4J datastore

Neo4J is an open-source, embeddable graph database implemented in Java. It makes use of a property graph data model to store information. It features a reliable, full ACID transaction model as well as supporting mass scalability, up to several billion nodes per machine. Initially released in 2007, Neo4J is available in multiple platforms (such as Windows, Mac or UNIX) and has grown to be the most popular graph database, used in multiple critical systems worldwide. Other features include a fast, optimized framework for query traversing and a transactional HTTP-Endpoint for Cypher (Neo4J's query language).

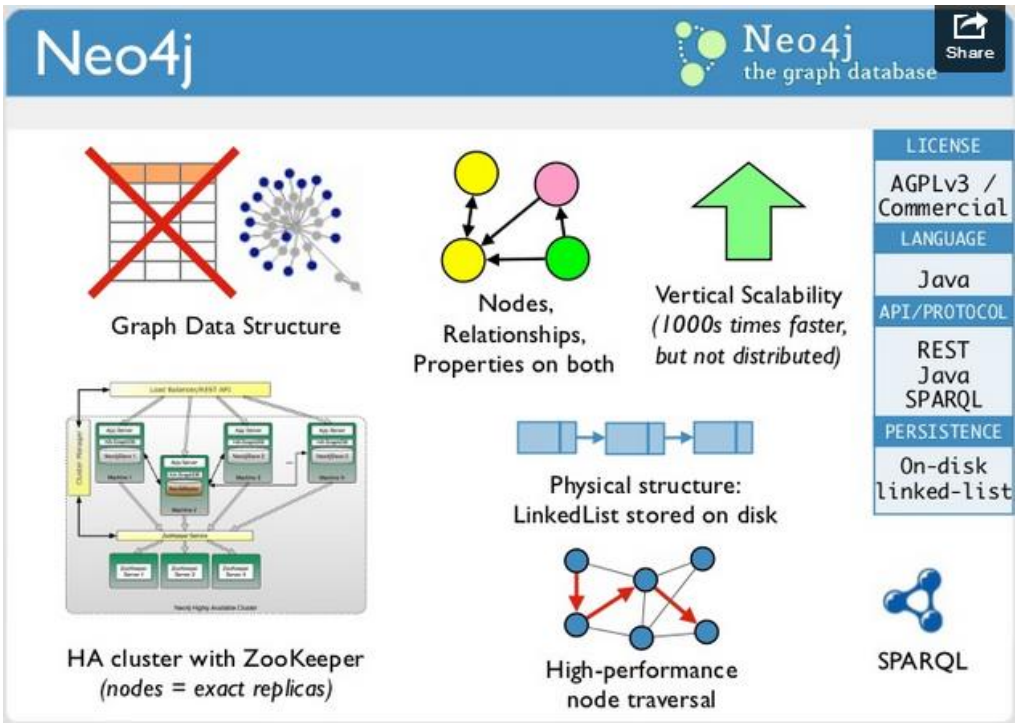


Figure 7 - Quick overview of Neo4J's graph database advantages.

It can be accessed from a REST interface as well as from an object-oriented Java API (also extendable to other languages). Neo4J is licensed under the free GNU General Public License. The most recent version (2.0) has been released in December, 2013. Neo4J can also include other libraries such as the Neo4J spatial, a toolkit designed to import OSM datasets and perform or test spatial queries. This allows for any data to be adapted to complex queries with geographic components.

Neo4J operates with a property graph data model. This consists of directed edges (nodes) and labeled vertices, with associated pairs of data (relationships), which can have values (properties). A relationship connects 2 nodes, has a well-defined type and can be also directed. Properties are key/value pairs that are attached to both nodes and relationships.

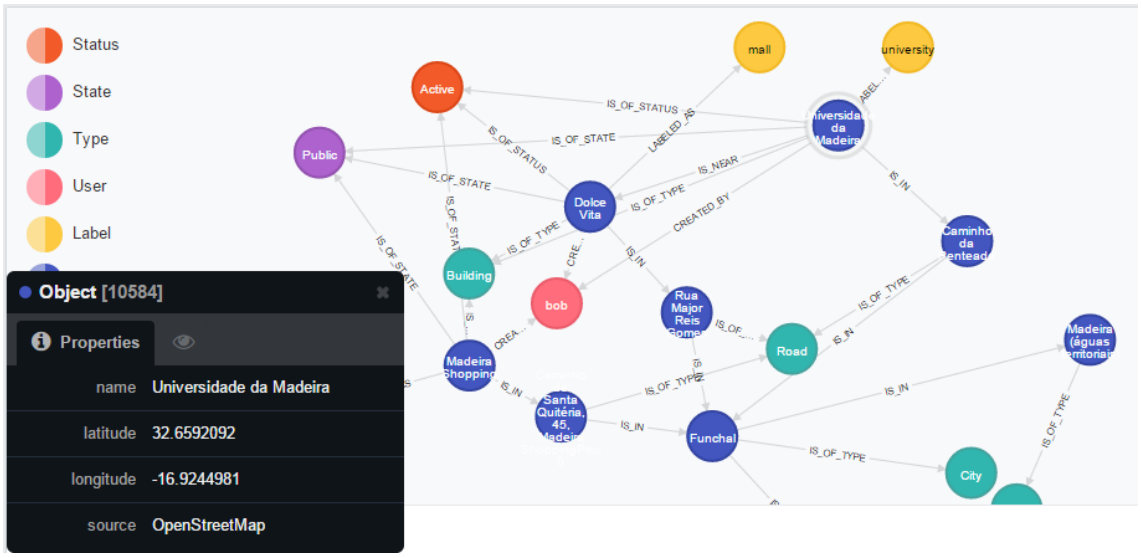


Figure 8 - A simple graph representation in Neo4J.

In figure 8 we can observe a simple semantic representation of a geographical domain. Nodes labeled as Object stand for our geographical places of interest where Type nodes define each Object into one of their categories (such as buildings, roads, cities or countries). The State nodes define weather an Object is public or private, and the Status nodes define weather an Object is currently active or not. Relationships such as IS_IN represent a place of interest that is geographically contained in another (for example, University of Madeira belongs to Funchal), while relationships such as IS_NEAR represent proximity within an arbitrary value. Properties are available both to nodes and do relationships (but are not visible in this diagram).

Operating with data in Neo4J requires a different approach compared to a relational database. Instead of relying on declarative sentences (much like SQL would), graph databases operate by navigating from a given node via relationships to the nodes that match our criteria. This can be done by using a node query language called Cypher, which will be explained in the following section.

Traversing a graph is done by following existing relationships according to some rules. In the previous figure 8, if we wanted to know which country University of Madeira belongs to we would have to identify that object node first and follow any IS_IN relationships that derive from it and from any objects it connects until we land on an object node that is classified as a country (has a IS_OF_TYPE relation to the object

Country). The path would be along the lines of: Uma->IS_IN->Funchal->IS_IN->Madeira->IS_IN->Portugal->IS_OF_TYPE->Country.

This kind of search becomes suitable for when we want to find a specific node based on its connectivity with others, but is significantly worse when one wants to find a node according to a property it has. If we wanted to find which countries have a population attribute over 10.000.000, we would have to perform a node-by-node look-up instead (preferably using indexes).

### 3.3.1 Cypher query language

Cypher is a declarative graph query language developed for Neo4J, which allows for expressive and efficient querying and updating of the graph database [46]. Very complicated queries can be easily expressed through Cypher due to its graphical syntax. Being a declarative language means that the language is focused on the clarity of expressing what to retrieve from a graph, not how to retrieve it, as a contrast with other imperative languages such as Java or Gremlin. Several semantics and commands Cypher have been inspired from other languages, such as SQL or Python.

The main structure is based on clauses chained together just like SQL. The 3 main clauses for reading a graph are MATCH, WHERE and RETURN. The MATCH clause describes the graph pattern we wish to follow. The WHERE clause (which works as a sub-clause to MATCH or other variants such as OPTIONAL MATCH and WITH) adds constraints to a pattern or can be used to filter certain properties. Finally, the RETURN clause describes what to return from our query. Relationships can be represented with a special syntax that connects both nodes and their corresponding relationship name.

For example, if we wish to list all places of interest that belong to Funchal, we would perform the following query:

```
MATCH (a: Object)-[IS_IN]->(Object {name: 'Funchal'}) RETURN a
```

The (a)-[REL]->(b) syntax represents a relationship that starts in node a and ends in node b. In this example, we start by attributing a label to the first unknown node (a) and describing which type of node it was labeled with (in this case, Object). The pattern of interest here is any node that connects to Funchal by a IS_IN relationship, which is

expressed with the MATCH clause and retrieved by the RETURN clause. In the previous graph, the answer would be 2 objects: University of Madeira and Dolce Vita.

The following query returns which country Madeira belongs to. It involves a more complex pattern matching:

```
MATCH (Object {name: 'Madeira'})-[IS_IN]->(a)-[IS_OF_TYPE]->(Type
{name: 'Country'}) RETURN a
```

a) Identify the starting node, which is labeled as an Object with a name property: `MATCH (Object {name: 'Madeira'})`
b) Follow our desired relationship. Finding where an object belongs to can be done with the previously explained IS_IN relationship, which connects the object Madeira to our target object (represented as a) `(Object {name: 'Madeira'})-[IS_IN]->(a)`
c) Our target node must have a type definition, so we will look for the relationship IS_OF_TYPE that connects it to a type named 'Country'. `(a)-[IS_OF_TYPE]->(Type {name: 'Country'})`
d) We return our target node `RETURN a`

According to our previous graph (figure 8), the result would be the Portugal node.

### 3.3.2 The Neo4JPHP wrapper

Neo4J can be accessed from either a REST interface as well as from an object-oriented Java API. However, this can also be extended to other languages such as PHP. The Neo4JPHP Wrapper, written by Josh Adell aims at providing access to the full functionality of the Neo4J REST API via PHP [48]. It can provide both a direct correspondence with REST via cypher query calls as well as a way to abstract the REST interface away, so that the application modeling can be done by using node and relationship object abstraction. Neo4JPHP is able to model several operations for each of the following entities with an object oriented approach:

a) Nodes and relationships

| | |
|---|---|
| `$arthur = $client->makeNode()` | Creates a node object |
| `$arthur->setProperty('name', 'Arthur Dent')` | Sets an array of properties in the node object |
| `$arthur->save()` | Saves the node object or relationship array to Neo4J |
| `$arthurId = $arthur->getId()` | Retrieves the id from the node object as a variable |
| `$character->getProperties($arthurId)` | Retrieves the node object properties |
| `$character->removeProperty('name')` | Removes a node object's property and assigned values |
| `$arthur->delete()` | Deletes a node or relationship |
| `$arthur->relateTo($earth, 'LIVES_ON')` | Creates a directed relationship from one node to another |
| `$arthurRelationships = $arthur->getRelationships()` | Returns the list of relationships associated with a node |

b) Labels

| | |
|---|---|
| `$label = $client->makeLabel('MyLabel');` | Creates a label |
| `$allLabels = $client->getLabels();` | Returns the list of labels |
| `$labels = $node->addLabels(array($myLabel, $myOtherLabel));` | Adds a label to an existing node |
| `$remainingLabels = $node->removeLabels(array($myLabel, $oneMoreLabel));` | Removes an array of labels from a node |

c) Indexes

| | |
|---|---|
| `$shipIndex = new Everyman\Neo4J\Index\NodeIndex($client, 'ships');`<br>`$shipIndex->save();` | Creates an index and saves it to the server |
| `$shipIndex->delete();` | Deletes an index |
| `$shipIndex->add($heartOfGold, 'propulsion', $heartOfGold->getProperty('propulsion'));` | Indexes an object on one of the properties |
| `$shipIndex->remove($heartOfGold);` | Removes all indexes from a given node |

d) Paths

| | |
|---|---|
| `$paths = $startNode->findPathsTo($endNode)->getPaths();` | Returns all existing paths between the 2 specified nodes |
| `$paths = $startNode-` | Returns the path with a maximum depth |

| | |
|---|---|
| `>findPathsTo($endNode, 'KNOWS', Relationship::DirectionOut)`<br>`    ->setMaxDepth(5)`<br>`    ->getPaths();` | of 5 |
| `<$paths = $startNode->fidind($endNode, 'KNOWS', Relationship::DirectionOut)`<br>`    ->setMaxDepth(5)`<br>`    ->getPaths();` | Returns the shortest path between 2 nodes with a maximum depth of 5, using the distance property to determine the cost |

e) Cypher queries

| | |
|---|---|
| `$queryString = "START n=node(1) ".`<br>`"MATCH (n)<-[:KNOWS]-(x)-[:HAS]->()".`<br>`"RETURN x";`<br>`$query = new Everyman\Neo4J\Cypher\Query($client, $queryString);`<br>`$result = $query->getResultSet();` | Specified a query string and then creates a server call to process it |

f) Transactions in Cypher

| | |
|---|---|
| `$transaction = $client->beginTransaction();` | Opens a new transaction |
| `$query = new Query($client, 'MATCH n WHERE id(n)={nodeId} RETURN n', array('nodeId' => 123));`<br>`$result = $transaction->addStatements($query);` | Adds a query to the transaction |
| `$transaction->commit();` | Commits the transaction. If unsuccessful, performs a rollback |

Table 3 - Main operations in Neo4JPHP.

The following PHP Code snippet creates a small graph with some nodes, sets some properties and relationships and saves them to the server:

```
require('vendor/autoload.php'); // or your custom autoloader

$client = new Everyman\Neo4J\Client('host.example.com', 7575); // Connecting
to a different port or host
$arthur = $client->makeNode(); // creates a new node
$arthur->setProperty('name', 'Arthur Dent') // sets a few properties

    ->setProperty('age', '27')
    ->setProperty('job', 'teacher')
    ->save(); // saves the node to the database
$arthurId = $arthur->getId(); // retrieves the node id

$character = $client ->getNode($arthurId); // loads the node id into the
variable
$myLabel = $client->makeLabel('Person'); // creates a label named 'Person'
$labels = $node->addLabels(array($myLabel)); // attributes this label to the
node we created
$personIndex = new Everyman\Neo4J\Index\NodeIndex($client, 'person');
$personIndex->add($arthur, 'name', $arthur->getProperty('name')); //indexes
the name
$bob = $client->makeNode(); // creates a new node
$bob->setProperty('name', 'Bob Prefect') // sets some properties
     ->setProperty('occupation', 'time traveler')
     ->save(); // saves the node
$bobId = $bob->getId();
$character2 = $client ->getNode($bobId); // retrieves the id for the Bob
node
$arthur->relateTo($bob, 'FRIEND') // creates a friend relationship between
Arthur and Bob
    ->setProperty('duration', '10 years')
    ->save();
```

Code snippet 3 - PHP code using Neo4JPHP internal commands.

The previous Code snippet can also be re-written into a series of REST calls using Cypher:

```
$client = new Everyman\Neo4J\Client('host.example.com', 7575); // Connecting
to a different port or host
$arthur = "CREATE (arthur: Person {name: 'Arthur Dent', age: '27', job:
'teacher'})";
$createNodeArthur = new Everyman\Neo4J\Cypher\Query($client, $arthur);

$bob = "CREATE (bob: Person {name: 'Bob Prefect', occupation: 'time
traveler'})";
$createNodeBob = new Everyman\Neo4J\Cypher\Query($client, $bob);
$createIndex = "CREATE INDEX on: Person(name)" ;
$Index = new Everyman\Neo4J\Cypher\Query($client, $createIndex);

$createRel = "MATCH (arthur:Person {name: 'Arthur Dent'}), (bob:Person
{name: 'Bob Prefect'}) CREATE (arthur)-[:KNOWS]->(bob)";
$createNodeBob = new Everyman\Neo4J\Cypher\Query($client, $createRel);
```



Code snippet 4 - The same graph created in Cypher queries.

Rewriting commands as Cypher queries significantly decreases the length of our code as well as some of its complexity, but in return it cannot be as easily reused or extended to other classes. It also assumes the developer to have previous experience in this language.

This chapter presented an overview of the chosen database models, as well as some of their technical details. The next chapter will present the implementation phase in detail.

# 4 - Geoplace Explorer

The next step in this work is to implement a unified geographical model, named the Geoplace Explorer (GE). GE's architecture can be briefly explained as a client-server application, able to perform remote HTTP requests to online geospatial datasets such as Factual, OpenStreetMap and GeoNames. This data is integrated and stored in a new model, able to encompass different attributes and details received from each source. It is then stored as a graph and saved in a Neo4J database. An identical dataset is also stored in a MySQL repository for comparison purposes. Basic data insertion and manipulation is also possible through a Web UI developed in PHP/HTML. This chapter presents a concrete overview of GE's functionalities. External plugins were also added to this project, in order to

GE consists of approximately 800 files (mostly dependencies, libraries and various configuration files) and the developed features account for roughly 3.400 lines of code.

## 4.1 The symbolic world model

The goal of this work was to create an interoperable geographic database, able to store and consolidate data from several sources, as well as storing it using an intuitive and accessible data model, which allows the application to store its knowledge. This framework is based on symbols which serve to represent relations between the agent and its environment. Such abstraction derives its power from the physical symbol system hypothesis [49], which consists of symbols (the phenomena in the abstract) and tokens (their physical instantiations). Much of the human knowledge is symbolic, which leads to a symbolic system to be easier to understand and facilitate many architecture capabilities in terms of planning, learning and artificial intelligence.

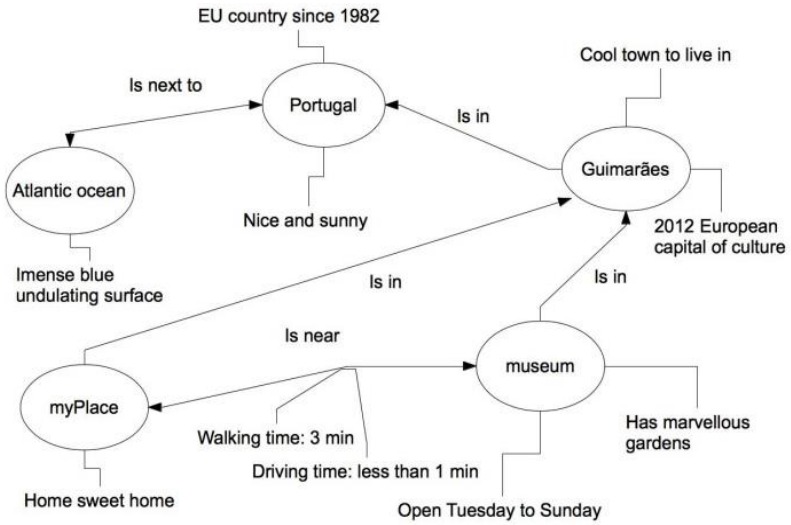


Figure 9- A simple overview of the symbolic model.

At its simplest form, the proposed model consists of objects, object attributes, relationships and relationship attributes. Objects can represent places and are further described by a set of attributes. Relations represent any kind of topological or other connection between places and can also be further described by a set of attributes. Figure 9 exemplifies this structure. This model was adapted from the thesis "Dynamic World Model for context-aware environments" [58], by Karolina Baras. The original model was used to map a generic world environment, and as such many of the table attributes have been renamed or adapted to better fit our geographical domain. For example, it is no longer needed to specify if relationships are symmetric or transitive and types of places have also been added (city, country, roam, etc.) according to the obtained data.

The Entity-relationship model (Figure 10) is briefly explained as a set of tables that store objects and their corresponding attributes. An object can represent any physical entity such as a school, supermarket, road or country. Objects can be semantically related to each other through relationships. These can be either transitive (where A->B->C implies A->C), symmetric (A->B implies B->A) or simple (A->B implies neither of the previous). Relationships are defined by a domain (starting node), co-domain (ending node) and a relationship name which can have 0 or more attributes. Creation timestamps for both, nodes and relationships, are stored at an individual level (*obj_cdate* and *rel_cdate*) and tied to the user that created them. This allows us to maintain lineage of our data set.

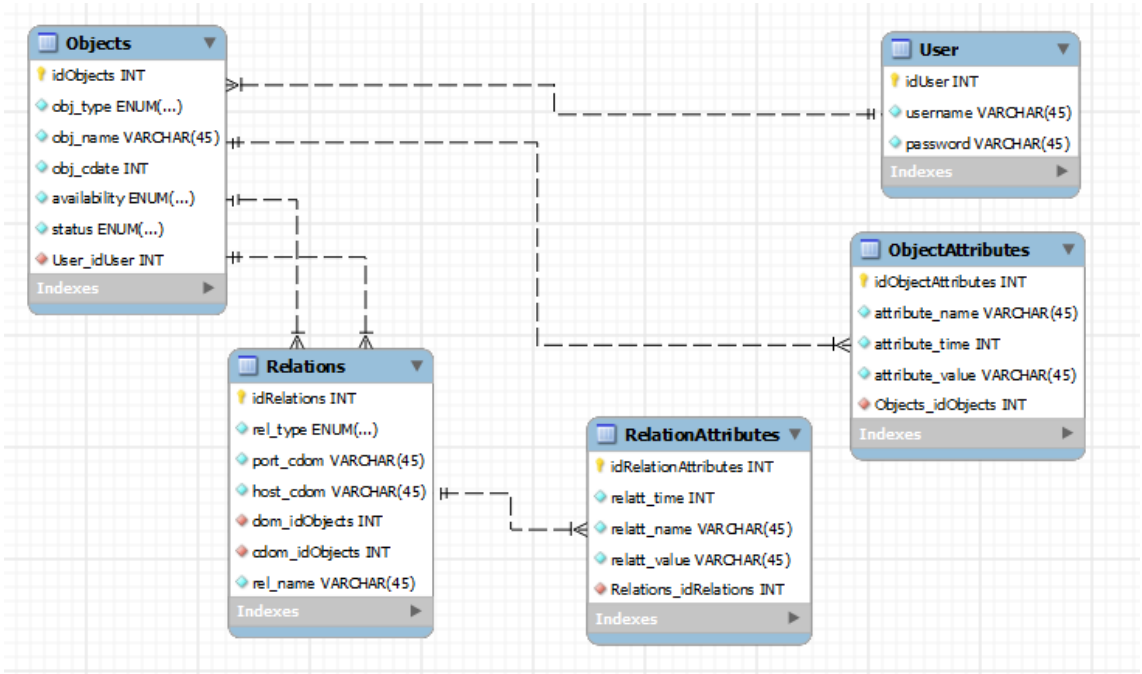


Figure 10 - Entity Relation model for Geoplace Explorer.

The graph equivalent of the studied symbolic world model is more difficult to express in abstract form, as it can accept *n* variations of data between objects, relationships as well as attributes. However, figure 11 demonstrates the most basic relations and most common labels that can be fulfilled. Since graphs are a naturally schema-less structure, they do not require as many ontological restrictions as the relational model and can be adapted as new data is inserted. All graph relationships are transitive by default.

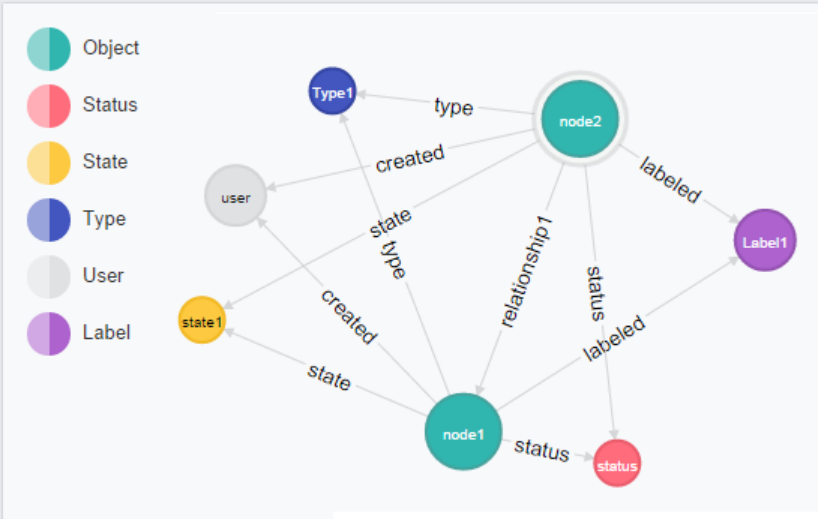


Figure 11 - Simplistic view of the graph model for Geoplace Explorer.

### 4.2 Environment configuration

The application environment for GE consists of several components: server configuration, API call dependencies for Factual, GeoNames and Nominatim, the server's bootstrap file for REST calls and communication between databases, the latest version of Neo4J community (2.1), the latest version of the InnoDB MySQL datastore, the latest PHP version (5.6) and the current Neo4JPHP wrapper.

This project uses the XAMPP open source Apache distribution for Windows, which contains the required Apache, PHP and MySQL services, available at http://sourceforge.net/projects/xampp/. The Neo4J server's latest stable release is available at the official page http://www.Neo4J.org/download/other_versions. Other minor dependencies such as stylesheets are included in the deliverables.

### 4.2.1 Neo4J Server configuration

The first step is to configure the Neo4J server so that our application can correctly communicate. It is possible to put test and development data in the same database instance, however a two database solution is often more effective to keep both data sets separate. Unlike SQL or many NOSQL database servers, Neo4J cannot have more than one database in a single instance, which means we must create two separate hosts. To do this, we must edit the file ~/Neo4J/dev/conf/Neo4J-server.properties and make sure the server is running on port 7474, and with another instance of the server the same procedure can be done for port 7475 (our development port) [51].

```
org.Neo4J.server.webserver.port=7474
```

By default, the Neo4J web server is bundled with a web server that binds to host localhost, only answering requests from the local machine. In order to enable access from external hosts, the property `org.Neo4J.server.webserver.address=0.0.0.0` must be set accordingly [50]. The server also includes built in support for SSL encrypted communication over HTTPS, and as such it is recommended to use a private key and certificate. This can be done by setting the location in the following line:

```
# Certificate location (auto generated if the file does not exist)
org.Neo4J.server.webserver.https.cert.location=ssl/snakeoil.cert
```

Neo4JPHP must be included in the project by downloading the latest release on GitHub (https://github.com/jadell/Neo4Jphp/archives/master). In order to load it, a bootstrap class must be created and included in the project as well. It references the autoloader files for server startup, the current server address and ports to connect, as well as other settings such as error reporting. It also differentiates our application environment from our test environment, which is shown in Code snippet 5.

```
require_once(__DIR__.'/vendor/autoload.php');
require_once(__DIR__.'/lib/Neo4Play.php');
error_reporting(-1);
ini_set('display_errors', 1);
if (!defined('APPLICATION_ENV')) {
    define('APPLICATION_ENV', 'development');}
$host = 'localhost';
$port = (APPLICATION_ENV == 'development') ? 7474 : 7475;
$client = new Everyman\Neo4J\Client($host, $port);
// ** set up error reporting, environment and connection... **//
Neo4Play::setClient($client);
require_once(__DIR__.'/vendor/autoload.php');
```

Code snippet 5 - Neo4JPHP bootstrap settings file.

### 4.2.2 API dependencies

In order to retrieve geographical data from online sources, some dependencies are required to include in this project. Factual's PHP Plugin requires PHP5 and the PHP5-curl version 7.10.5 or greater extension to allow for HTTP requests. This can be done in XAMPP by simply adding *extension=php_curl.dll* into the php.ini configuration file.

Factual's PHP Plugin can be obtained from https://github.com/Factual/factual-php-driver (GitHub) and currently supports every geospatial operation from the web API. It can included in the project by simply adding a *require_once("Factual.php")* statement.

In order to query the Factual database, one must first create a free API key at https://www.factual.com/keys which is limited by the amount of hourly and daily requests it can send. The provided OAUTH KEY and OAUTH SECRET must be specified when performing a connection to the server. This is done by creating a Factual object with the following parameters:

```
$factual = new Factual ("yourOAuthkey","yourOAuthSecret");
```

GeoNames_Service is also necessary to perform queries to the Geonames web project. All dependency files are handled by the PEAR package. It can be manually downloaded

from the official page at http://pear.php.net/package/PEAR/download. The GeoNames web service package is also available at the following address: http://pear.php.net/package/Services_GeoNames/download. In a similar fashion, one must also create a free username at http://www.geonames.org/login and enable the web service on the account page. From the project, the dependencies must be included with *require_once 'Services/GeoNames.php'* and an authentication object must also be created.

```
$geonames = new Services_GeoNames ("username");
```

The Nominatim WebService allows the user to retrieve search-based queries from the OpenStreepMap database in JSON format. This is necessary as OSM stores its data mainly in geometric format, and GeoPlace explorer needs access to plain categorical and symbolic data. The JSON parser file is included in this project, and can be easily read in PHP by a simple openXML command.

```
$xml = simplexml_load_file("http://nominatim.openstreetmap.org/search?
g=$field&format=xml
```

### 4.3 Application Architecture

GeoPlace Explorer can be split into several architectural layers. The main application connects to the SQL database trough PHP calls and to the graph database via the Neo4JPHP wrapper. This module is responsible for creating the world model structure from the recieved input and transmits data to the Neo4J database via REST API. Since graph databases do not possess a defined data structure, every relationship rule must be stored in the application logic and this constitutes our world model. Data from the three main repositories (Factual, Geonames and OSM) is gathered as HTTP Requests from the above mentioned API's. Manual insertion can also be performed by the user from the Web UI (as seen from Figure 12).

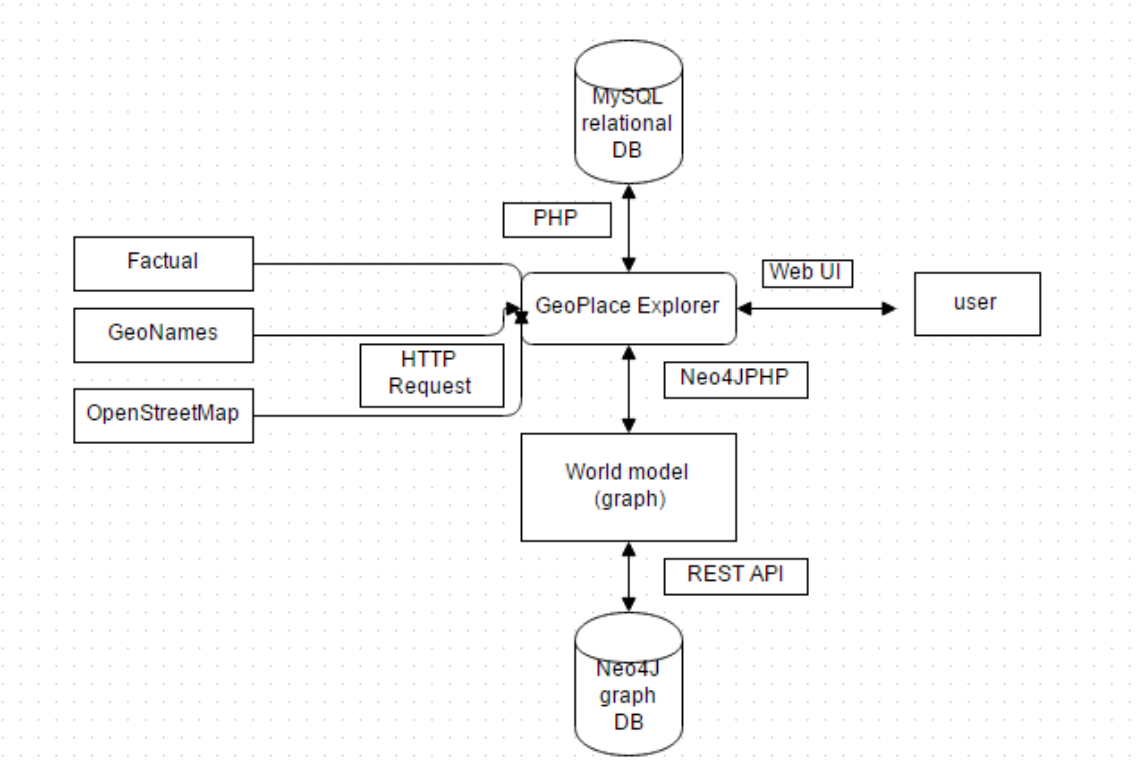


Figure 12 – GeoPlace explorer architecture diagram.

### 4.4 Implementation

GeoPlace explorer is a test tool created to internall visualize and manipulate spatial data. As such, no significant usability tests were performed while developing this application.

#### 4.4.1 Authentication

The first page of the application handles the user authentication process. It was implemented as a simple series of PHP sessions to validate users between pages. The first page "*newuser.php*" (Figure 13) allows the user to create a new account that can then be used to login and start adding data to our database. The new account creation takes a username and two password fields for validation and stores a creation date. Passwords are stored as an md5 hash into both databases, along with the creation time and date (expressed in Linux time). For simplicity purposes, only the data stored in Neo4J is used to login to GE from the *"login1.php"* page. This username is later tied to any nodes or relationships created by the user.

From a technical standpoint, usernames are stored as a labeled node in Neo4J and as table rows in MySQL (see code snippet 6).

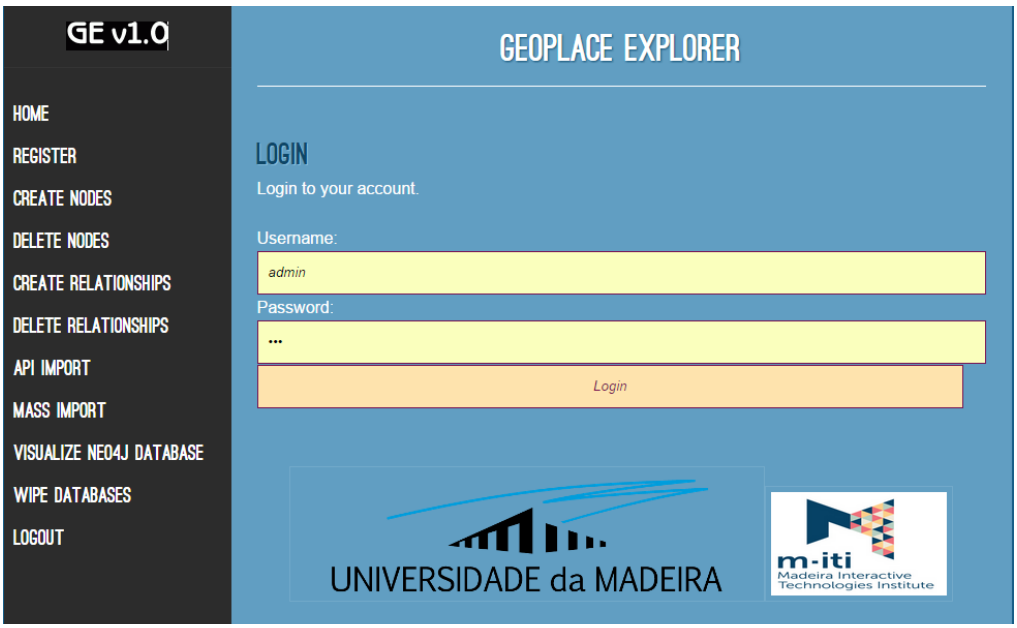

Figure 13 - Login screen.

```
if (isset ($_POST['user'])) {

 $user = $_POST['user'];

 $password = $_POST['pw'];

 $pass_enc = md5($password);

 $str = "MATCH (n:User) WHERE n.username = '$user' AND n.password =
'$pass_enc' RETURN n"; //check if user data is correct

 $check_user = new Everyman\Neo4j\Cypher\Query($client, $str);
```

Code snippet 6 - Username and password verification.

### 4.4.2 Node manipulation

After logging in to Geoplace Explorer, the user is presented with several choices, one of them present in the 'Add node' button leading to *'create_node.php'*. This allows for a manual node insertion, where several fields and properties can be specified. Each node represents a different geographic entity. It is required to insert a node name, an associated type (there is a default list of types which can be extended if

needed), a public/private setting (if the place of interest is either a public or private entity) and a custom attribute field (not mandatory). Default types can be Island, Country, Road, Building, Ocean, City, Sightseeing, Continent, Region or River but this can also be changed by the user (Figure 14).

Internally, our Neo4J database stores each record as an object, connected to a certain object type, object state, object status and author, with all associated attributes. MySQL stores this data as a single row in the *objects* table, with a possible foreign key to *objectattributes* in case a custom attribute is specified.

Nodes can be deleted from both databases from the *node_dismiss.php* page (Figure 15), which lists all existing nodes and corresponding types and properties. This operation starts a transaction to permanently delete the chosen node, as well as any existing relationships that previously enclosed it.

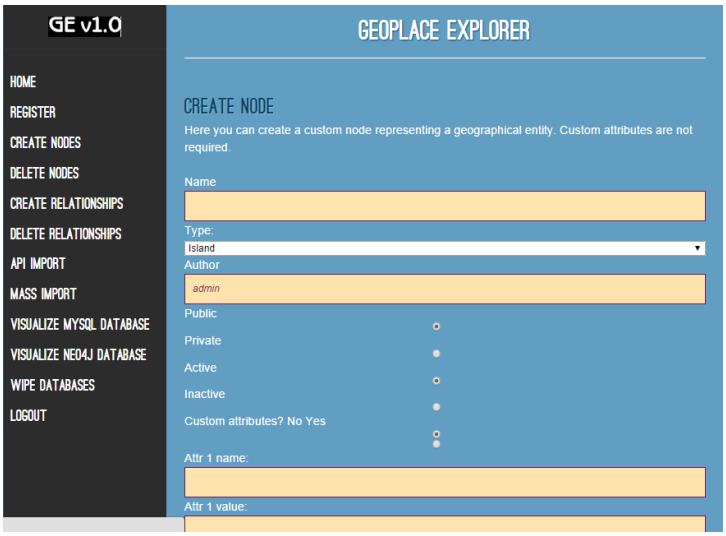


Figure 14 - Node creation page

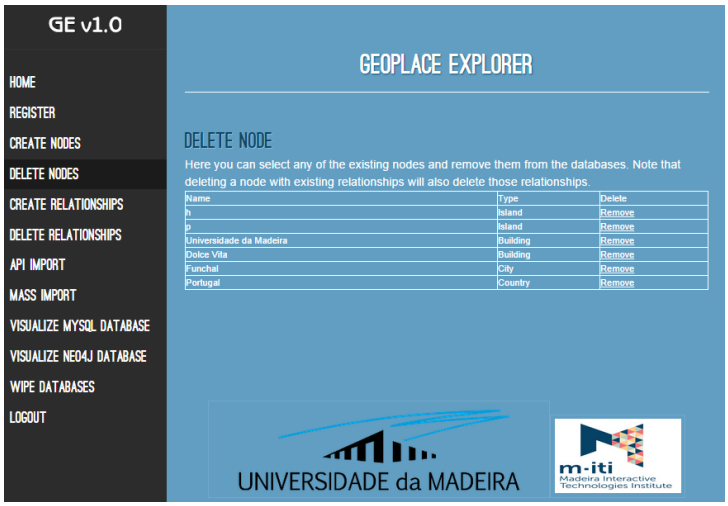


Figure 15- Node deletion page.

```
if (!empty($_POST['objectName']) && !empty($_POST['type']) &&
!empty($_POST['objectAuthor']) &&
!empty($_POST['objectavailability']) &&
!empty($_POST['objectstatus'])) {

if ($_POST['atrstate'] == "yes") {

//sql code below

$query1_sql = "SELECT idUser FROM User WHERE username = '$author'";
//RETRIEVE THE USER ID

$q1_sql = mysql_query($query1_sql) or die(mysql_error());

$str = mysql_fetch_array($q1_sql, MYSQL_ASSOC);

$userid = $str['idUser'];

if ($str) {$query2_sql = "INSERT INTO`objects` (`idObjects`,
`obj_type`, `obj_name`, `obj_cdate`, `availability`, `status`,
`User_idUser`) VALUES (NULL, '$type', '$name', '$final', '$state',
'$status', '$userid')";

$q1_sql = mysql_query($query2_sql) or die(mysql_error()); //insert
object

} if($q1_sql) {

$obj_id = mysql_query("SELECT idObjects FROM objects WHERE obj_name =
'$name'"); //retrieve objectid

$sql = mysql_fetch_array($obj_id, MYSQL_ASSOC);

$idobj = $sql['idObjects'];

//insert object attributes

$obj_attrib = mysql_query("INSERT INTO `objectattributes`
(`idObjectAttributes`, `attribute_name`, `attribute_time`,
`attribute_value`, `Objects_idObjects`) VALUES (NULL, '$atr1',
'$final', '$atr1val', '$idobj')") or die(mysql_error())
```

Code snippet 7 - PHP code for SQL node insertion

As code snippet 7 shows, most nodes and attributes are inserted manually in the Object table, with all appropriate validations taking place.

### 4.4.3 Relationships

Relationships between existing nodes can be created on *'rels.php'* page (Figure 16). The interface prompts the user to select one node (the starting node) as well as an ending node and specify a relationship name between them. Attributes can also be specified if necessary. The way relationships are stored varies significantly across each database. In Neo4J, a relationship is physically stored on disk while in SQL

relationships are stored at a higher level in table definitions. Relationships are by default considered transitive and can be deleted from the *"delete_rels.php"* page (Figure 17).

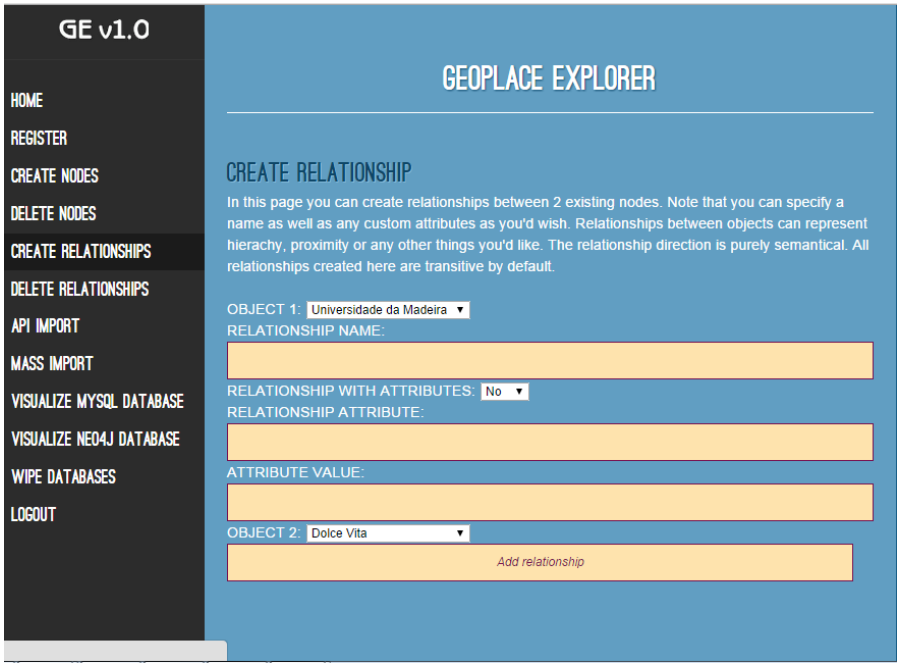


Figure 16 - Relationship creation page

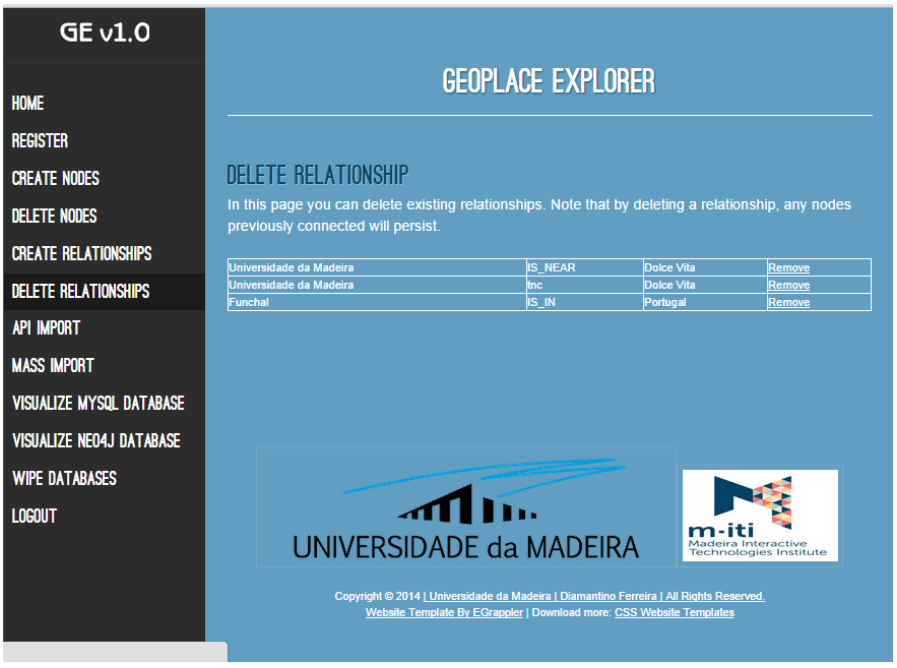


Figure 17 - Delete relationship page.

### 4.4.4 API data insertion and importing

The API insertion page is responsible for querying the various online geo-database services and returning all relevant places of interest. This is done by utilizing the Factual API, the GeoNames service endpoint and the Nominatim OpenStreetMap service. The user is prompted to insert a place name to lookup, which then returns all the related results, ordered by relevance and source. Each result consists of several attributes (which may vary depending on the source and the quality and completeness of data) presented in a table. As different sources can often return inconsistent results (for example, GeoNames does not keep addresses), it becomes important to create a schema-less structure to store our data. Several relationships of interest can also be found here, as a natural hierarchy of results is also present. Place names are located in a particular address, which belongs to a locality, which belongs to a country. Certain place names are also labeled according to several filters, such as recreations, universities, restaurants or transportation entities. Other data, such as business schedules and reviews are also included in the database as custom attributes; however, it is not shown on the webpage to avoid cluttering. Once a desirable result is found, the user can press the 'Add' button, which brings up a confirmation form prior to inserting data (Figure 18).

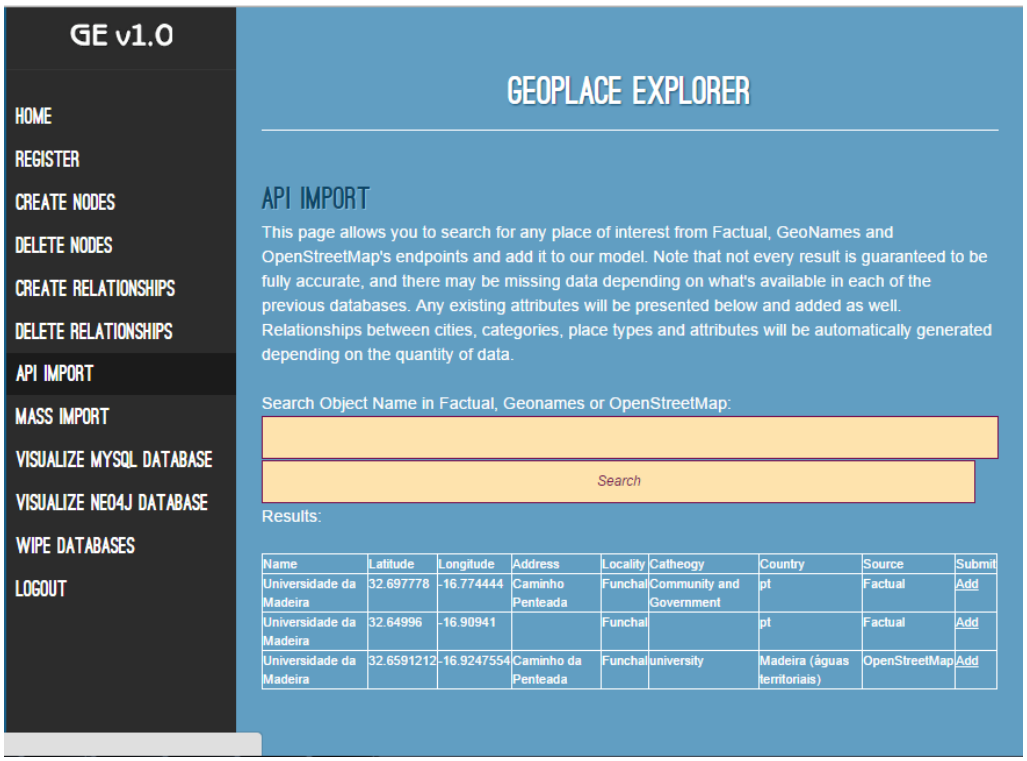


Figure 18 - API insertion page.

It is also possible to batch import a large number of records at once using Factual's reverse geo-coding feature (Code snippet 6), which returns a list of results centered on a location (latitude, longitude) and a given radius. This is useful later on to perform tests with larger amounts of data. Duplicate results are omitted.

```
$lat = 32; $long = -17; $total = 0;

      for ($lat = 32; $lat < 33; ) {

      $lat = $lat+0.1;

      for ($long = -17; $long < -16; ) {

$long = $long+0.1; echo $lat; echo ":"; echo $long; echo "---"

$query = new FactualQuery; $query->within(new FactualCircle($lat,
$long, 5000));

$query-
>select(array("name,latitude,longitude,country,address,locality,c
ategory_labels")); //factual name search

$query->limit(50); $res = $factual->fetch("places", $query);

$tnc= $res->getData(); array_push($big, $tnc);
```

Code snippet 8 - Mass import algorithm.

A graph database makes it easier to add unstructured data sources. For example, a typical pattern would consist in a place name that belongs to an address, which belongs to a city. Since not all data sources provide these details, we can build our graph independently for each result, so if a result doesn't contain an address or a city other layers can instead be connected directly to the place name. Code snippet 9 illustrates this in action, where a placename can be directly contained in a city.

```
$building_to_address = "MATCH (build: Object {name: '$name'}),
(add: Object {name: '$address'}) CREATE UNIQUE (build)-[:IS_IN]-
>(add)";

$address_to_city = "MATCH (add: Object {name: '$address'}),
(city: Object {name: '$locality'}) CREATE UNIQUE (add)-[:IS_IN]-
>(city)";

$query_ba = new Everyman\Neo4j\Cypher\Query ($client,
$building_to_address);

$query_variable = new Everyman\Neo4j\Cypher\Query ($client,
$address_to_city);

if ($query_ba->getResultSet() && $query_variable-
>getResultSet()) {echo "TNC"; } else {echo "test2";}

} else {$building_to_city = "MATCH (build:Object {name:
'$name'}), (city:Object {name: '$locality'}) CREATE UNIQUE
(build)-[:IS_IN]->(city)";
```



Code snippet 9 - Placename insertion in Cypher

# 5- Tests

This section presents all functions used to perform benchmarks between databases. As the abilities offered by a relational and non-relational model differ significantly in terms of clustering, spatial functions and indexing, it is important to analyze under which conditions each model can be more successful. The following tests are based around relevant requests, which represent typical queries the GE tool would have to perform. It is worth noting that due to hardware limitations and due to the demanding nature of these queries most tests could not be scaled extensively (most are limited to 100 users and/or 100.000 records in each database). These benchmarks are, however easy to scale over a larger system given the required resources.

## 5.1 Test specifications

Previous studies have been made around graph database performance [56], however in most cases they do not apply to a particular domain and simply measure query times centered around join statements with little relevance to the data schema. In this project, all tests aim to query the exact same information to both database models and measuring response times related to software algorithms (i.e., the hardware speed is minimized). In each case, a query is expressed both in Cypher and SQL. The first part describes each tool used as well as a small configuration guide. The test methodology is then explained and all gathered results will be presented and compared. In order to more accurately measure availability, each concurrency test simulates a total of 100 users performing the same request a certain number of times.

## 5.2 Environment

The following tests were performed with 2 different tools. Neo4J tests were performed with Gatling (http://gatling.io/), an open source stress load framework commonly used to analyze and measure performance in a variety of services, with a focus on web applications. Simulations can be coded using Scala, an object-functional and scripting language for software applications. SQL tests were performed with the mysqlslap tool, an built-in load-emulator able to support several requests and simultaneous connections to a MySQL database.

The documented tests were performed on the following hardware:

Processor: Intel® Core i3 CPU M330 @2.13 Ghz

RAM: 4.00 GB 1067 MHz clock rate

Operative System: Windows 7 Ultimate 64bits

### 5.2.1 Gatling for Neo4J

Gatling can be obtained from http://gatling.io/download/. This project uses version 1.5.6 due to compatibility issues with the newest release. Simulation scripts are defined by a scenario written in the Scala language, and tests take place as an asynchronous parallel model. The test configuration file (\gatling-charts-highcharts-1.5.6-bundle\gatling-charts-highcharts-1.5.6\user-files\simulations\neo4j\test.scala) queries the REST API in our Neo4J server over 10 seconds with 100 users for a specific cypher query (create a node) with a 5 millisecond interval between requests. Results are parsed JSON objects. Gatling can be invoked from a command line and simulation scripts must be placed inside the /user-files/simulations folder. The Gatling configuration file for our first test is available on Code snippet 7. Several test simulations were based on Max de Marzi's blog [57].

```
class CreateNode extends Simulation {
  val httpConf = httpConfig /* connects to the Neo4j rest
endpoint */
    .baseURL("http://localhost:7474")
    .acceptHeader("application/json")

  val createNode = """{"query": "create me"}""" /**

  val scn = scenario("Create Node")
    .repeat(1000) {
    exec(
      http("create node")
        .post("/db/data/cypher") /** sends in the cypher
query */
        .body(createNode)
        .asJSON
        .check(status.is(200)))
      .pause(0 milliseconds, 5 milliseconds) }
  setUp( /* Simulate for 100 users */
    scn.users(100).ramp(10).protocolConfig(httpConf)
  )
}
```

Code snippet 10 - Scala configuration file for the createNode simulation

### 5.2.2 Mysqlslap

Mysqlslap is a built-in utility to benchmark and compare MySQL performance. It can be invoked from command line. The following example (Code snippet 8) creates a simulation for a custom query performed 5 times by 100 different users.

Row insertions were performed by a simple SQL script, available on Code snippet 9, by iterating a large number of table insertions.

```
mysqlslap.exe --user=root --create-schema=sc_db --
query="SELECT obj_name FROM objects" --concurrency=100
--iterations=5
```

Code snippet 11 - Syntax for a custom query simulation.

```
DELIMITER $$
CREATE PROCEDURE add_objects_db()
BEGIN
  DECLARE i INT DEFAULT 1; DECLARE Random1 INT;
  WHILE i < 100000 DO
  SET Random1 = ROUND(123456 * RAND());

    INSERT INTO `sc_db`.`objects` (`idObjects`, `obj_type`,
`obj_name`, `obj_cdate`, `availability`,

`status`, `User_idUser`) VALUES (NULL, 'Building', i, Random1,
'Public', 'Inactive', '28');
    SET i = i + 1;
  END WHILE;
END$$
DELIMITER ;
CREATE PROCEDURE ADD_RELATIONS_DB()
BEGIN
  DECLARE k INT DEFAULT 1;
DECLARE Random1 INT;DECLARE Random2 INT;DECLARE Random3 INT;
DECLARE Random4 INT;DECLARE Random5 INT;DECLARE Upper INT;
DECLARE Lower INT;SET Upper = 99999;SET Lower = 1;
  WHILE k < 2000 DO
  SET Random1 = ROUND(((Upper - Lower -1) * RAND() + Lower), 0);
  SET Random2 = ROUND(((Upper - Lower -1) * RAND() + Lower), 0);
  SET Random3 = ROUND(((10*Upper - Lower -1) * RAND() + Lower),
0);
  SET Random4 = ROUND(((Upper - Lower -1) * RAND() + Lower), 0);
  SET Random5 = ROUND(((Upper - Lower -1) * RAND() + Lower), 0);
  INSERT INTO `sc_db`.`relations` (`idRelations`, `cdate`,
`rel_type`, `port_cdom`, `host_cdom`, `dom_idObjects`,
`cdom_idObjects`, `rel_name`) VALUES
(NULL, Random3, 'Transitive', 'localhost', 'localhost', Random1,
Random2, 'IS_IN');
INSERT INTO `sc_db`.`relations` (`idRelations`, `cdate`,
`rel_type`, `port_cdom`, `host_cdom`, `dom_idObjects`,
`cdom_idObjects`, `rel_name`) VALUES
(NULL, Random3, 'Transitive', 'localhost', 'localhost', Random2,
Random4, 'IS_IN');
INSERT INTO `sc_db`.`relations` (`idRelations`, `cdate`,
`rel_type`, `port_cdom`, `host_cdom`, `dom_idObjects`,
`cdom_idObjects`, `rel_name`) VALUES
(NULL, Random3, 'Transitive', 'localhost', 'localhost', Random4,
Random5, 'IS_IN');
    SET k = k + 1;
  END WHILE;
```

```
DELIMITER $$
CREATE PROCEDURE ADD_REL_ATT()
BEGIN
  DECLARE k INT DEFAULT 1;
  DECLARE value INT DEFAULT 5087;
DECLARE Random1 INT;
DECLARE Random2 INT;
DECLARE Upper INT;
DECLARE Lower INT;
SET Upper = 1;
SET Lower = 100000
  WHILE k < 5000 AND value < 11140 DO
  SET Random1 = ROUND(((Upper - Lower -1) * RAND() + Lower), 0);
  SET Random2 = ROUND(((2*Upper - Lower -1) * RAND() + Lower),
0);

 INSERT INTO `sc_db`.`relationattributes`
(`idRelationAttributes`, `relatt_time`, `relatt_name`,
`relatt_value`, `Relations_idRelations`) VALUES (NULL, Random2,
'atr1', Random1, value);
    SET k = k + 1;
    SET value = value +1;

  END WHILE;
END$$
DELIMITER ;

***
DELIMITER $$
CREATE PROCEDURE ADD_OBJ_ATR()
BEGIN
  DECLARE k INT DEFAULT 1;
  DECLARE value INT DEFAULT 1;
DECLARE Random1 INT;
DECLARE Random2 INT;
DECLARE Random3 INT;
DECLARE Upper INT;
DECLARE Lower INT;
SET Upper = 1;
SET Lower = 10000;
  WHILE k < 10000 AND value < 10000 DO
  SET Random1 = ROUND(((Upper - Lower -1) * RAND() + Lower), 0);
  SET Random2 = ROUND(((2*Upper - Lower -1) * RAND() + Lower),
0);
  SET Random3 = ROUND(((4*Upper - 2*Lower -1) * RAND() + Lower),
0);

 INSERT INTO `sc_db`.`objectattributes` (`idObjectAttributes`,
`attribute_name`, `attribute_time`, `attribute_value`,
`Objects_idObjects`) VALUES (NULL, Random1 , Random3, Random2,
value)
    SET k = k + 1;
    SET value = value +1;
  END WHILE;
END$$
DELIMITER ;
```

Code snippet 12 - SQL script used to populate the database.

### 5.3 Test results

All tests are performed for both databases with identical size datasets. In our test scenario, a total of ~100.000 nodes and 600.000 relationships records were randomly generated in each database while preserving the structure and complexity of typical records. The goal was to initially mass import a large number of results from each database, but this was not possible due to API traffic limitations.

#### 5.3.1 High concurrency node lookup operation

One request is sent to each database to measure the response time from a server call at a high concurrency (several simultaneous users). For Neo4J, this is done by querying the server status while searching for the root node (200) as shown by Code snippet 10.

```
class GetRoot extends Simulation {
  val httpConf = httpConfig
    .baseURL("http://localhost:7474")
    .acceptHeader("application/json")
  val scn = scenario("Get Root")
   .during(100) { exec(
       http("get root")
         .get("/db/data/node/0")
         .check(status.is(200)))
     .pause(0 milliseconds, 5 milliseconds) }
  setUp(
    scn.users(100).protocolConfig(httpConf))}
```

Code snippet 13 - Empty root node operation in Scala.

The test was scaled for 100 users over 100 seconds, repeating every 5 milliseconds, for an approximate total of 417.000 requests. Response time results are shown in figure 19.

STATISTICS — Expand all groups | Collapse all groups

| Requests | Executions | | | Response Time (ms) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | Min | Max | Mean | Std Dev | 95th pct | 99th pct | Req/s |
| Global Information | 438464 | 438464 | 0 | 0 | 630 | 5 | 18 | 20 | 80 | 4381 |
| get root node | 438464 | 438464 | 0 | 0 | 630 | 5 | 18 | 20 | 80 | 4381 |

Figure 19 - Response time results for Neo4J.

In this case, response times can vary significantly. With the highest value being 630 ms. However, the mean value was significantly lower (5 ms). 71% of the results were within 3 ms of response time while 23% were within 9 ms response time. Only a very small minority exceeded 20 ms (2%) with a standard deviation of 18 ms. We can, therefore exclude the highest result (630 ms) as an outlier since it is not contained in any confidence interval.

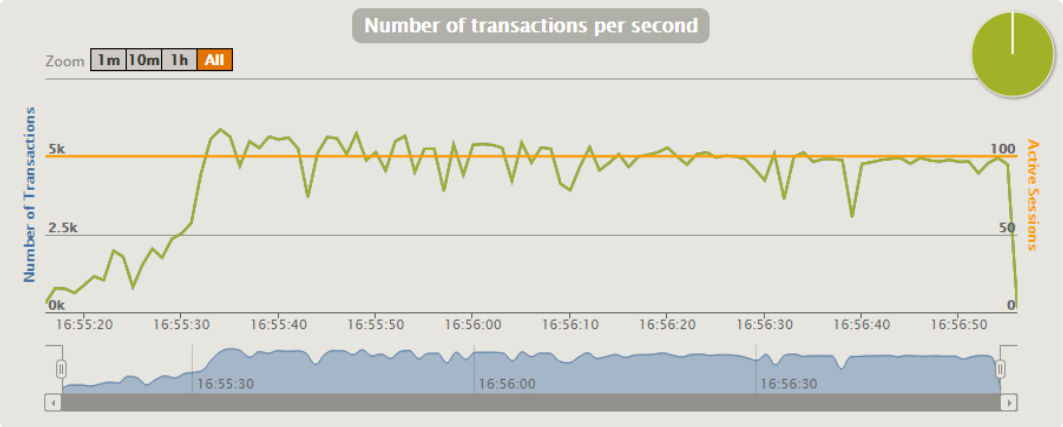


Figure 20 - Transactions per second results for Neo4J.

The number of simultaneous requests averaged to around 5.000 transactions per second, with steep fluctuations during the early and final stages.

In order to test the relational performance, an identical request was sent to MySQL, seen in code snippet 11. A certain object was requested from the object table.

```
mysqlslap.exe --user=root --create-schema=sc_db --
query="SELECT obj_name FROM objects WHERE idObjects = 1" --
concurrency=100 --iterations=100
```

Code snippet 14 - Command line test parameters used for test 1 on MySQL.

This test was also scaled to 100 concurrent users for a total of 100 iterations. Results are shown in figure 21 seen below:

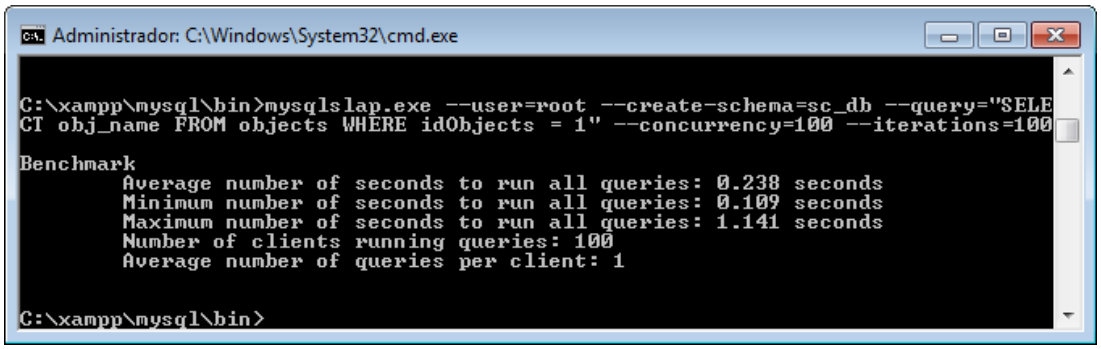


Figure 21 - MySQL test 1 results.

In this case, response times varied between 109 ms and 1141 ms, with an average of 238 ms per request, significantly higher than Neo4J.

### 5.3.2 Low concurrency property node lookup

This test aims to measure response times between databases for a simple node request at low levels of concurrency. For Neo4J, this was done by specifying a property node lookup trough Gatling:

```
import com.excilys.ebi.gatling.core.Predef._
import com.excilys.ebi.gatling.http.Predef._
import akka.util.duration._
import bootstrap._
import util.parsing.json.JSONObject

class node_property extends Simulation {
  val httpConf = httpConfig
    .baseURL("http://localhost:7474")
    .acceptHeader("application/json")
    .requestInfoExtractor(request => {
      println(request.getStringData)
      Nil })

  val node = exec((session) => {
    session.setAttribute("params", JSONObject(Map("n" ->
n)).toString())
  })
  val nodequery = """MATCH n WHERE n.name = 'tnc' RETURN n"""
  val cypherQuery = """{"query": "%s", "params": %s
}""".format(nodequery, "${params}")
  val scn = scenario("node_property")
    .repeat(100) {
      exec(chooseRandomNodes)
      .exec(
        http("node_property")
          .post("/db/data/cypher")
          .header("X-Stream", "true")
          .body(cypherQuery)
          .asJSON
          .check(status.is(200)))
      .pause(0 milliseconds, 5 milliseconds)}
  setUp(
    scn.users(1).ramp(10).protocolConfig(httpConf)
  )}
```

Code snippet 15 - Scala test file for the property node lookup test.

STATISTICS

Expand all groups | Collapse all groups

| Requests | Executions | | | Response Time (ms) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | Min | Max | Mean | Std Dev | 95th pct | 99th pct | Req/s |
| Global Information | 100 | 100 | 0 | 1570 | 1800 | 1612 | 48 | 1650 | 1750 | 1 |
| node_property | 100 | 100 | 0 | 1570 | 1800 | 1612 | 48 | 1650 | 1750 | 1 |

Figure 22 - Results from the second test in Neo4J

As figure 22 suggests, the average response time was of 1612 ms, with a standard deviation of 48. This means that roughly 39% of the results were within 1601 ms and that the 99$^{th}$ percentile interval contains results up to 1756ms. No outliers were detected.

The equivalent test was performed to the relational database, as seen in code snippet 13, with the results are available in figure 23.

```
mysqlslap.exe --user=root --create-schema=sc_db --
query="SELECT obj_name FROM objects WHERE obj_name = tnc" --
concurrency=1 --iterations=100
```

Code snippet 16 - Test case for MySQL property node lookup.

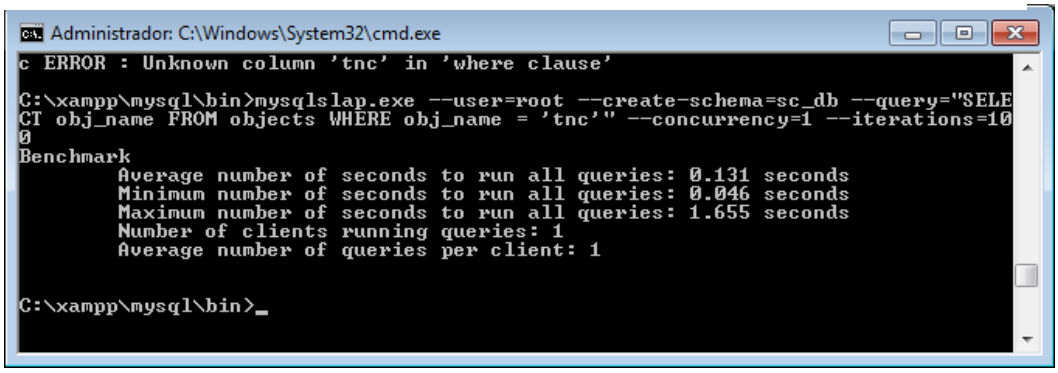


Figure 23 - Test case results for MySQL property node lookup.

We can conclude this lookup was much faster in SQL compared to the graph database. An average time of 131ms compared to 1612ms, approximately 12 times faster than Neo4J.

### 5.3.3 Single relationship lookup

This test will compare response times between databases for a simple relationship lookup. It prints out every existing relationship in each database with

a depth of 1. For this test we will use a concurrency factor of 1 with 100 repetitions, due to the heavy nature of the query.

The Scala configuration file for the Neo4J test is as follows in code snippet 14:

```
import com.excilys.ebi.gatling.core.Predef._
import com.excilys.ebi.gatling.http.Predef._
import akka.util.duration._
import bootstrap._
import util.parsing.json.JSONObject


class Findrelations1 extends Simulation {
  val httpConf = httpConfig
    .baseURL("http://localhost:7474")
    .acceptHeader("application/json")
    .requestInfoExtractor(request => {
      println(request.getStringData)
      Nil
    })

  val findRelations = """MATCH (node1)<-[IS_IN]-(node2)
RETURN node1, node2"""
  val cypherQuery = """{"query": "%s", "params": %s
}""".format(findRelations, "${params}")


  val scn = scenario("Find relations1")
    .repeat(100) {
      .exec(
        http("create relationships")
          .post("/db/data/cypher")
          .header("X-Stream", "true")
          .body(cypherQuery)
          .asJSON
          .check(status.is(200)))
      .pause(0 milliseconds, 5 milliseconds)
  }

  setUp(
    scn.users(1).ramp(10).protocolConfig(httpConf)
  )
}
```

Code snippet 17 - Scala configuration file for Neo4J test 3.

The test simulation was loaded into Gatling, and the results are available below in figure 24:

STATISTICS — Expand all groups | Collapse all groups

| Requests | Executions | | | Response Time (ms) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | Min | Max | Mean | Std Dev | 95th pct | 99th pct | Req/s |
| Global Information | 100 | 100 | 0 | 2550 | 4230 | 2718 | 177 | 2900 | 3020 | 0 |
| create relationships | 100 | 100 | 0 | 2550 | 4230 | 2718 | 177 | 2900 | 3020 | 0 |

Figure 24 - Results from the third test in Neo4J (response times).

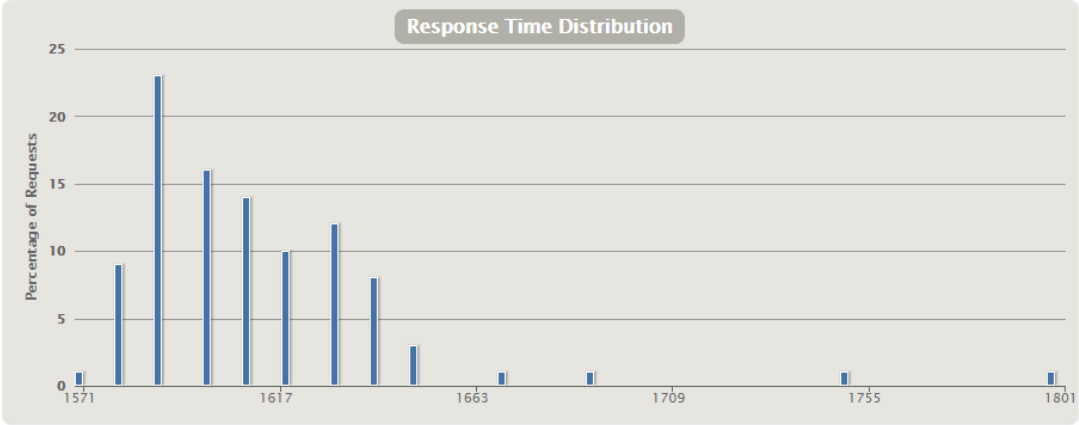


Figure 25 - Results from the third test in Neo4J (response times distribution).

As Figure 24 indicates, the mean response time was of 2718 ms, with a standard deviation of 177ms. Since our 99$^{th}$ percentile confidence interval only goes up to 3020 ms, we can safely discard the maximum result of 4230 ms as an outlier.

In MySQL, we can define the same query as code snippet 15 indicates:

```
SELECT A.rel_name, C1.obj_name AS object1_name, C2.obj_name
AS object2_name

FROM relations AS A

INNER JOIN Objects AS C1 ON A.dom_idObjects = C1.idObjects
INNER JOIN Objects as C2 ON A.cdom_idObjects = C2.idObjects
```

Code snippet 18- SQL query to find relationships of depth = 1.

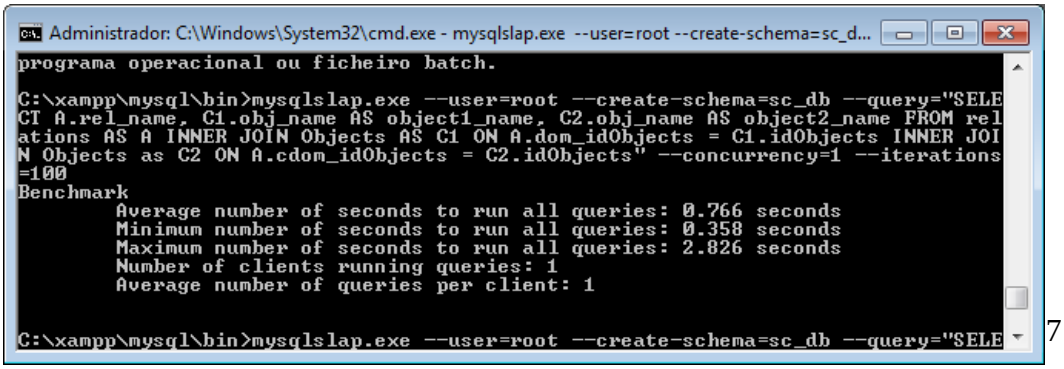


Figure 26- Test results for the 3rd test in MySQL.

An average of 766 ms is fairly lower than the obtained results from Neo4J (2718 ms). Even with a large variance between maximum and minimum values, MySQL has performed substantially better in this regard.

### 5.3.4 Multiple relationship lookup

This final test will measure the response time for a depth three relationship, a common request for our data hierarchy. Since this request has an expensive computational cost associated and due to hardware limitations, only 10 simulations were performed with a concurrency level of 1. This operation is available for both databases in code snippet 16, while results for the Neo4J database can be seen in Figure 26.

```
MATCH (node1)<-[IS_IN*3]-(node2) RETURN node1, node2
```

```
CREATE VIEW REL1
as
SELECT A.rel_name, C1.obj_name AS object1_name,
C2.obj_name AS object2_name
FROM relations
AS A INNER JOIN Objects AS C1 ON A.dom_idObjects =
C1.idObjects
INNER JOIN Objects as C2 ON A.cdom_idObjects =
C2.idObjects

CREATE VIEW REL2 AS
(SELECT A.rel_name, C1.obj_name AS object3_name,
C2.obj_name AS object4_name
FROM relations AS A
INNER JOIN Objects AS C1 ON A.dom_idObjects =
C1.idObjects
INNER JOIN Objects as C2 ON A.cdom_idObjects =
C2.idObjects)

CREATE VIEW REL3 AS

SELECT rel1.object1_name, rel1.object2_name,
rel2.object3_name, rel2.object4_name
FROM rel1
INNER JOIN rel2
ON rel1.object2_name = rel2.object3_name

SELECT rel3.object1_name, rel2.object2_name,
rel3.object3_name, rel2.object4_name
FROM rel2, rel3
INNER JOIN rel3
ON rel2.object2_name = rel3.object3_name
```

Code snippet 19 - Depth 3 relationship (Cypher | SQL).

STATISTICS — Expand all groups | Collapse all groups

| Requests | Executions | | | Response Time (ms) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | Min | Max | Mean | Std Dev | 95th pct | 99th pct | Req/s |
| Global Information | 10 | 10 | 0 | 39150 | 42570 | 41425 | 946 | 42570 | 42570 | 0 |
| find relations 2 | 10 | 10 | 0 | 39150 | 42570 | 41425 | 946 | 42570 | 42570 | 0 |

Figure 27 - Neo4J results for the final test.

### 5.3.5 Shortest path operation (Neo4J spatial extra)

This final test was performed using Neo4J spatial, a community developed library of utilities that facilitates the enabling of spatial operations on data. This UNIX package can read .OSM files (a special XML schema definition used to store data in the OpenStreetMap database) and export their data into a local Neo4J database, which then allows adding spatial indexes, geometric properties (points, line-strings and polygons) to imported data and performing spatial operations with it. Configuring Neo4J spatial was a very lengthy process, as most files had to be compiled from a command line in a Linux environment (via Cygwin, since Windows was the only available OS at the time). For this test, we used the Portugal.OSM file, available from the free at the GeoFabric Project: http://download.geofabrik.de/

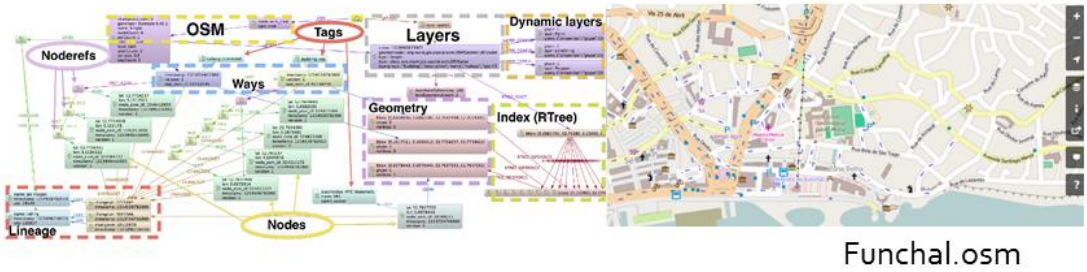


Figure 28 - Neo4J spatial data layout (left) and OSM map used for testing (right)

For this case, the Dijkstra's shortest path class was used to perform a computation between two nodes (University of Madeira and Dolce Vita Funchal). This operation was performed with the following Cypher command:

**p = shortestPath((node1)-[*..15]-(node2))**

The average result was of 0.16 seconds. This test was also performed in SQL, but with a different database manager called POSGRE SQL. POSTGRE is an open-source relational database manager that features a geospatial routing add-on called

pgRouting. This allows us to query spatial operations on a relational dataset using the following function:

**shortest_path('SELECT gid AS id, source, target, length AS cost FROM network',**

By creating an SQL query, we can then find the distance between our 2 points (start and end:

```
SELECT SOMA(cost) AS distance FROM

shortest_path('SELECT pid AS id, source, target, length AS
cost FROM network',

get_network_id(:start), get_network_id(:end), false, false);
```

Code snippet 20 - SQL query for shortest-path command

The average response time was of 0.78 seconds.

## 5.4 Measurements

### 5.4.1 Objective measurements

The 4 previous tests were performed within the Geoplace Explorer framework, with all relevant data in the previous section. Table 4 consolidates all measured values which will be discussed next.

| Simulation | MySQL | Neo4J |
|---|---|---|
| Import time | 31 minutes | 2.5 hours |
| High concurrency lookup | 238 ms | 5 ms |
| Low concurrency property lookup | 131 ms | 1612 ms |
| Single relationship lookup | 716 ms | 2718 ms |
| Depth 3 relationship lookup | 3.5 hours | 41.5 seconds |
| Disk space | 22.7 MB | 27.1 MB |
| Shortest-path | 0.78s (POSTGRE) | 0.16s |

Table 4- Simulation results for MySQL and Neo4J using GeoPlace Explorer data.

Data insertion took much longer to perform in Neo4J than in MySQL, approximately 5 times longer resulting in a database size of 27 MB. This makes sense considering graph databases store every relationship in disk, and the importing process seems to be also less efficient when compared to MySQL and also heavily bottlenecked by RAM. Neo4J was considerably faster when dealing with high concurrency requests (something to expect from a NoSQL database); however this also came at very steep CPU usage during tests. The difference between low concurrency, property node lookups clearly favors MySQL, since Neo4J lacks robust text-based indexing, and because graph queries scale for how much of the graph we wish to explore. Text-matching queries (as was the case in the property node lookup) were much slower to perform as well. Relationships with a depth of 1 were faster to perform in MySQL, due to B-tree indexes and because the amount of data was not very large. These principles are part of the pgRouting add-on for POSTGRE, as some graph indexes are used to optimize performance.

Higher-depth relationships, however scale up exponentially in complexity and were much faster to perform in Neo4J. This is mostly due to the graph nature of handling interconnected data. In computational terms, this happens because SQL is forced to calculate multiple JOIN operations between entire multiple tables, even if only a small subset of data is requested while hierarchical data (based on relations) and shortest-path operations (based on relation weight and neighbors) are heavily optimized in Neo4J since graph databases maintain direct referenced to interconnected nodes (a shortest path operation can be seen as a breadth-first-search). This means that query latency in a graph database is proportional to how much of the graph we choose to explore in a query and not proportional to the amount of data stored in total unlike a relational database.

In general, SQL databases are not designed to perform traversal queries and Neo4J provides the optimal solution for cases like this. In terms of vertical scalability, Neo4J proved to have large potential in this regard, but this is somewhat off-set by its poor memory management. The amount of time it takes to import large amounts of data is also a concern.

### 5.4.2 Subjective measurements

Subjective measurements cannot be mathematically obtained, however there are several interesting characteristics between both models that warrant discussion. During the development stage, it was very obvious that MySQL has a much higher level of maturity, stability and richness of functionality, which reflects in better support, reliability, compatibility, ease of use, external tools and integration options. Neo4J is a relatively recent project (version 1.0 launched in 2010), and graph databases are in general much less used in a consumer environment as well.

While Neo4J has a reasonable level of documentation, it still has a long way to go in terms of external tools, bug-testing and general resource management for some operative systems – this is partially related due to Neo4J being built over Java Virtual Machine. Most of the external tools (such as Neo4J spatial) are exclusively community developed and open-source in nature, and while the user base can be active and dedicated it is still difficult and time consuming to configure and use certain frameworks (such as Neo4J spatial and Neo4JPHP), requiring additional knowledge in terms of JVM configurations, UNIX operations, Object-oriented programming and configuration of external tools. In general, Neo4J requires a higher level of expertise to properly manage and maintain compared to a relational database.

As of yet there are limited options for integrating Neo4J with other technologies (only Java is available by default, and documentation is lacking). In terms of usability, Neo4J was surprisingly easy to learn. Cypher language was very natural and easy to understand, and graphs operations were not difficult to learn either. An interesting point to add is that most typical queries for this domain ended up being much simpler to express in Cypher than in SQL. A relationship of higher depths can be expressed as a simple one-liner in Cypher, while the equivalent would take complex recursive algorithms in SQL using views and multiple JOIN operations. Graph development, however has a higher burden in development time, since structures need to be defined as data arrives, unlike MySQL which relies on a predefined schema. On the other hand, this also means that any changes to the structural nature of the database can be easily changed on the fly

for Neo4J, while in MySQL they would need an SQL schema change that could potentially invalidate previous data.

In terms of stability, Neo4J performed slightly worse compared to MySQL. This can be attributed to the aforementioned level of maturity, however certain errors could be difficult to read and fix at times, and system crashes were a norm during the first few days of using this environment. The lack of true transaction support was also a small concern; however this is somehow alleviated by the write-ahead-log system, which only performs modifications on a temporary graph before committing. The Wait-for-Graph feature is also responsible for preventing deadlocks and has kicked multiple times during our test cases according to occasional console errors.

As a final consideration, scalability is also a very important aspect for this domain, with geo databases reaching up to billions of nodes and increasing. NoSQL systems naturally excel in this domain, and are already present in several online environments (Facebook's Cassandra and Google's BigTable, among others). As sharding and replication algorithms become more optimized for Neo4J, we can expect great things in future for graph-based systems. Furthermore, NoSQL databases were generally designed from the ground up for distributed computing. They are built to tolerate and recover from failure making them highly resilient. Since large, complex and fault-tolerant systems can be expensive to buy and maintain, NoSQL and graphs remain an attractive solution in this field as most of them are open-source projects.

# 6- Conclusion and future work

This chapter presents the final outline of this thesis. A conceptual framework for combining and storing geographical data was presented, and several tests were performed in order to find the most appropriate database system. We will present a quick overview of the development process and answer the proposed research questions according to the obtained results.

This thesis first started as a study for collaborative edition of geographical databases. Several existing solutions were presented, and many of the typical problems in the field were analyzed early on. Some of them were partially related to interoperability between sources, the completeness of data and the difficulty in accessing large amounts of information. Different sources use different ontologies (in syntactic, structural, semantic or cartographical levels) and deal with different sets of data. Together with the current scaling of technology, API development, crowdsourcing and linked data, it became imperative to find a way to consolidate this information clearly and consistently.

As such, and given the rise of several NoSQL solutions in the past years, it became our target to propose a world model that could easily adapt to any input and also to study how we could increase performance, stability and concurrency for such a system using a less conventional solution. Neo4J was a very natural choice for this problem, as it easily dealt with both the requirements for schema-less data as well as the high-performance requirements for geographical and traversal requests. Many solutions for interoperable models were suggested before, but most were still very strict in their own structure and demanded other sources to adapt to them. This means that if a certain piece of information was missing from a source (such a simple location, attribute or an abstract relation between entities), previous models would either not work, become inconsistent, or would need arbitrary adjustments to stay relevant. With graph structures, we adapt the incoming data to us instead.

In order to solve this, GeoPlace Explorer was then created, using PHP as it is especially aimed to web applications involving database communication. By adapting to the Neo4JPHP Wrapper and employing the typical HTML/PHP/SQL features, a successful integration of API calls was created, ready to submit, edit or delete any

desired content from either database when needed. Part of this work was published in a scientific paper at the SMART 2014 Conference[1].

The final step was to create performance tests, which would mimic common requests from our application. This was done with the help of stress testing tools such as Gatling and mysqlslap. After several simulations conducted, the typical conditions that favored performance for each database were discovered, such as graph traversals or simple index lookups propagating massively over benchmarks. Finally, we compared both solutions based on objective and subjective terms which, while hard to quantify also indicate serious flaws and conveniences between them. For example, while Cypher was a relatively easy language to learn and express queries compared to SQL, the development still took slightly longer when compared to MySQL. This is mainly due to the schema-less nature of graphs; while SQL databases simply rely on inserting data, graph databases need to be structured on the go during development. Depending on the situation, this can be either a good thing or a bad thing, as it can cause side effects if the application fails to manage data integrity. It is much easier to change structure for a graph while keeping existing data safe than it is for an SQL file.

In the end, we conclude that each storage model has its own paradigm, and neither is objectively better or worse than the other. NoSQL solutions are designed with certain goals in mind at the expense of other capabilities normally present in a RDBMS. While the graph database model was validated as the most appropriate solution for this project, it is worth noting that several advantages in the relational model still persist, such as full transactional support, stronger security settings or better response times for index based queries and it is entirely possible to create a hybrid approach (redundant databases) able to take advantage of both worlds. In essence, this would mean replicating the data into both databases as seen in Geoplace Explorer, with an intermediary architectural component responsible for analyzing incoming requests and redirecting them to the most appropriate database.

An interesting proposition for future work would be to extend GeoPlace explorer into a fully functional spatial database, using PostgreSQL spatial instead of MySQL.

[1] Ferreira, Diamantino RG, Fátima CR Fazenda, and Karolina Baras. "Online Geodatabases as a Source of Data for a Smart City World Model Building."*SMART 2014, The Third International Conference on Smart Systems, Devices and Technologies*. 2014.

This would bring interesting challenges into merging both layers of symbolic and geometric data, but in return it could potentially make use of several other features in a graph database (such as nearest point searches and shortest-path algorithms). Using the LinkedGeoData's framework it could also be possible to retrieve extra data from different sources, creating an even more dynamic world model, able to fully take advantage of both databases, thanks to our hybrid layer.

## References:


[1] "Geographic Information Systems as integrating Technology: context, concepts and definitions" http://www.colorado.edu/geography/gcraft/notes/intro/intro.html [Accessed: 12-12-2013]

[2] "Every day counts: Geographical data collaboration" http://www.fhwa.dot.gov/everydaycounts/edctwo/2012/pdfs/edc_geo_data.pdf [Accessed: 09-12-2013]

[3] "What is GIS – ESRI" http://www.esri.com/what-is-gis/overview#overview_panel [Accessed: 27-11-2013]

[4] "Wikipedia, Geographic Information System" http://en.wikipedia.org/wiki/Geographic_information_system [Accessed: 09-11-2013]

[5] A. Markoetz, T. Brinkhoff, B. Seeger "Exploiting the internet as a Geospatial database" presented at the Workshop on Next Generation Geospatial Information, 2003

[6] P. Boolstad, "GIS Fundamentals book, 4th Edition, 2012" pp 10-12 http://www.paulbolstad.net/gisbook.html [Accessed: 12-12-2013]

[7] Karam, R., & Melchiori, M. (2013, March). Improving geo-spatial linked data with the wisdom of the crowds. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops* (pp. 68-74). ACM.

[8] Maltese, V., & Farazi, F. (2013). A semantic schema for GeoNames.

[9] Van Canneyt, S., Van Laere, O., Schockaert, S., & Dhoedt, B. (2012, November). Using social media to find places of interest: a case study. In*Proceedings of the 1st ACM SIGSPATIAL International Workshop on Crowdsourced and Volunteered Geographic Information* (pp. 2-8). ACM.

[10] Clement, G., Larouche, C., Gouin, D., Morin, P., & Kucera, H. (1997). OGDI: toward interoperability among geospatial databases. *SIGmOD Record*, *26*(3), 18-23.

[11] Schmidt, M., & Weiser, P. (2012). Web Mapping Services: Development and Trends. In *Online Maps with APIs and WebServices* (pp. 13-21). Springer Berlin Heidelberg.

[12] Haklay, M. (2010). How good is volunteered geographical information? A

comparative study of OpenStreetMap and Ordnance Survey datasets.*Environment and planning. B, Planning & design*, *37*(4), 682.

[13] Mooney, P., Corcoran, P., & Winstanley, A. C. (2010, September). A study of data representation of natural features in openstreetmap. In *Proceedings of GIScience* (Vol. 150).

[14] Goetz, M., & Zipf, A. (2011). *Extending openStreetMap to indoor environments: bringing volunteered geographic information to the next level*. CRC Press: Delft, The Netherlands.

[15] Hahmann, S., & Burghardt, D. (2010, September). Connecting linkedgeodata and geonames in the spatial semantic web. In *6th International GIScience Conference*.

[16] Andrea Rodriguez, M., & Egenhofer, M. J. (2004). Comparing geospatial entity classes: an asymmetric and context-dependent similarity measure.*International Journal of Geographical Information Science*, *18*(3), 229-256.

[17] «5 things Foursquare and Facebook places can learn from Google+ | Clarion Consulting». [Em linha]. http://clarionconsulting.com/blog/5-things-foursquare-and-facebook-places-can-learn-from-google. [Accessed: 08-Jul-2013].

[18] «Dopplr Tour | Travel planning, advice & tips for the smart traveller». [Em linha]. Disponível em: http://www.dopplr.com/tour. [Accessed: 18-Jun-2013].

[19] «Facebook Also Said To Have A Deal With Localeze For Facebook Places | TechCrunch», *TechCrunch*. [Em linha]. Disponível em: http://techcrunch.com/2010/06/17/facebook-localeze-places/. [Accessed: 08-Jul-2013].

[20] «Facebook Asks Users to Clean Up Its Location Database with Places Editor and Favorite Places», *Inside Facebook*. [Em linha]. Disponível em: http://www.insidefacebook.com/2011/07/01/favorite-places-editor-location-database/. [Accessed: 08-Jul-2013].

[21] «Facebook's Places Editor - and what's wrong with it», *MediaBizTech*. [Em linha]. Disponível em: http://mediabiztech.wordpress.com/2011/06/30/facebooks-places-editor-and-whats-wrong-with-it/. [Accessed: 08-Jul-2013].

[22] «Google Releases Search for Database of 50 Million Places», *Mashable*. [Em

linha]. Disponível em: http://mashable.com/2010/10/27/google-place-search/. [Accessed: 08-Jul-2013].

[23] «How Google Builds Its Maps—and What It Means for the Future of Everything», *The Atlantic*, 06-Set-2012. [Em linha]. Disponível em: http://www.theatlantic.com/technology/archive/2012/09/how-google-builds-its-maps-and-what-it-means-for-the-future-of-everything/261913/. [Accessed: 25-Jul-2013].

[24] "Wikipedia, Spatial database" http://en.wikipedia.org/wiki/Spatial_database [Accessed: 10-12-2013]

[25] "Wikipedia, Factual" http://en.wikipedia.org/wiki/Factual [Accessed: 20-11-2013]

[26] "Factual Developer documentation, Factual " http://developer.factual.com/api-docs/ [Accessed: 10-11-2013]

[27] "Wikipedia, GeoNames" http://en.wikipedia.org/wiki/GeoNames [Accessed: 11-12-2013]

[28] "Geonames documentation, Feature codes" http://www.geonames.org/export/codes.html [Accessed: 10-11-2013]

[29] "Geonames, Web Services" http://www.geonames.org/export/web-services.html [Accessed: 10-11-2013]

[30] Hoffhine Wilson, E., Hurd, J. D., Civco, D. L., Prisloe, M. P., & Arnold, C. (2003). Development of a geospatial model to quantify, describe and map urban growth. *Remote sensing of environment*, *86*(3), 275-285.

[31] Chaves, M., Rodrigues, C., & Silva, M. (2007). Data model for geographic ontologies generation.

[32] Stadler, C., Lehmann, J., Höffner, K., & Auer, S. (2012). Linkedgeodata: A core for a web of spatial open data. *Semantic Web*, *3*(4), 333-354.

[33] "Wikipedia, OpenStreetMap" http://en.wikipedia.org/wiki/Openstreetmap [Accessed: 10-12-2013]

[34] "OSM Data Report 2012" https://www.mapbox.com/osm-data-report/ [Accessed: 11-12-2013]

[35] "LinkedGeoData" http://linkedgeodata.org/About [Accessed: 04-12-2013]

[36] "Geospatial databases and the evolving role of the surveyor", F. Derby

http://mycoordinates.org/geospatial-databases-and-the-evolving-role-of-the-surveyor/ [Accessed: 12-12-2013]

[37] "Spatial databases, course outline Spring 2012" http://uwaterloo.ca/geography-environmental-management/sites/ca.geography-environmental-management/files/uploads/files/GEOG%20387%20Outline%20S12.pdf [Accessed: 11-12-2013]

[38] "Geospatial databases and Data mining", chapter 3 http://www.nap.edu/openbook.php?record_id=10661&page=47 [Accessed: 11-12-2013]

[39] J. Lopez, J. Cesar "Geospatial database generation from digital newspapers: Use case for risk", MsC Science and Geospatial Technologies Dissertation, 2010

[40] Flanagin, A. J., & Metzger, M. J. (2008). The credibility of volunteered geographic information. *GeoJournal*, *72*(3-4), 137-148.

[41] Catlin-Groves, C. L. (2012). The citizen science landscape: from volunteers to citizen sensors and beyond. *International Journal of Zoology*, *2012*.

[42] Zielstra, D., & Zipf, A. (2010, May). A comparative study of proprietary geodata and volunteered geographic information for Germany. In *13th AGILE international conference on geographic information science* (Vol. 2010).

[43] "Wikipedia, Graph Theory" http://en.wikipedia.org/wiki/Graph_theory [Accessed: 12-07-2013]

[44] "RDBMS dominate the database market, but NoSQL systems are catching up – DB,Engines" http://db-engines.com/en/blog_post/23 [Accessed: 20-07-2013]

[45] "Wikipedia, NoSQL databases" http://en.wikipedia.org/wiki/NoSQL [Accessed: 14-07-2014]

[46] "Neo4J homepage, Why use a Graph database?" http://www.Neo4J.org/learn/graphdatabase [Accessed: 22-07-2013]

[47] "Slideshare, When do use NoSQL databases and why" http://www.slideshare.net/quipo/nosql-databases-why-what-and-when [Accessed: 14-08-2014]

[48] "GitHub Neo4JPHP documentation" by Josh Adell https://github.com/jadell/Neo4Jphp/wiki [Accessed: 08-02-2014]

[49] "Unified theories of cognition, Physchology

Today" http://ai.eecs.umich.edu/cogarch0/common/theory/utc.html [Accessed: 04-07-2014]

[50] "Securing access to Neo4J server, Neo4J official documentation" http://docs.Neo4J.org/chunked/stable/security-server.html [Accessed: 05-04-2014]

[51] "Everyman Software – Getting started with Neo4JPHP" by Josh Adell http://blog.everymansoftware.com/search?updated-max=2012-01-14T00:21:00-05:00&max-results=7&start=7&by-date=false [Accessed: 04-01-2014]

[52] "Just say yes to NoSQL, Pillar Global" http://www.3pillarglobal.com/insights/just-say-yes-to-nosql [Accessed: 20-08-2014]

[53] "Wikipedia, Amazon DynamoDB" http://en.wikipedia.org/wiki/Amazon_DynamoDB [Accessed: 11-08-2014]

[54] "Github documentation: graph morphisms" https://github.com/tinkerpop/blueprints/wiki/Graph-Morphisms [Accessed:10-07-2014]

[55] "Reality is a graph: embrace it, Neo4J presentation" by Michael Hunger http://pt.slideshare.net/jexp/intro-to-Neo4J-presentation

[56] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010, April). A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference* (p. 42). ACM.

[57] "Neo4J and Gatling sitting on a tree", Max de Marzi http://maxdemarzi.com/2013/02/14/neo4j-and-gatling-sitting-in-a-tree-performance-t-e-s-t-ing/ [Accessed: 20-09-2014]

[58] K. Baras, "Dynamic World Model for context-aware environments" Ph.D. dissertation, Dept. Inf. Sys. Univ. do Minho, Guimarães, Portugal, 2012.