

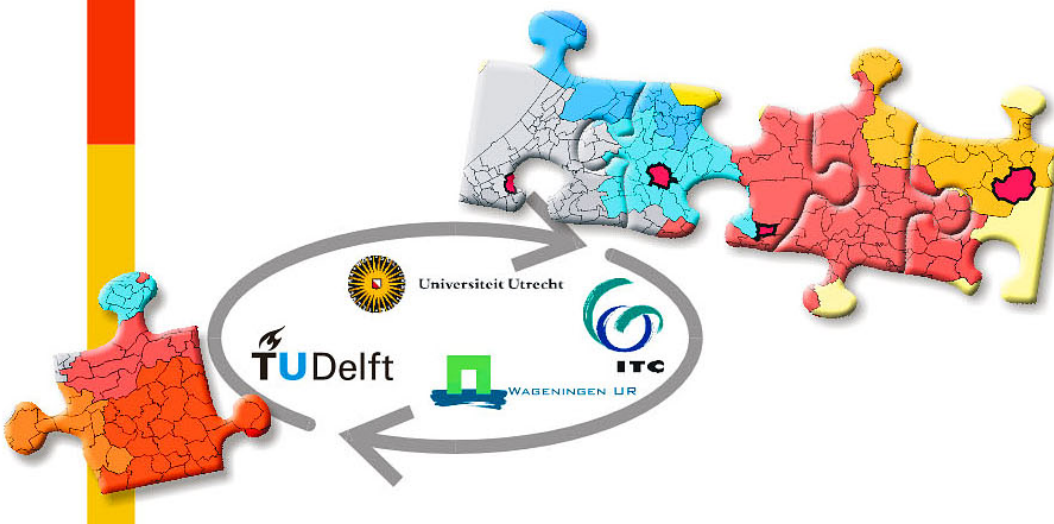
GIMA

Geographical Information Management and Applications

NoSQL spatial

Neo4j versus PostGIS

Bart Baas



NoSQL spatial

Neo4j versus PostGIS

Master Thesis

Date:

May 22, 2012

Version:

1.0 final

Author:

Bart Baas

Geographical Information Management and Applications (GIMA)

Supervisor:

W. Quak (TUD)

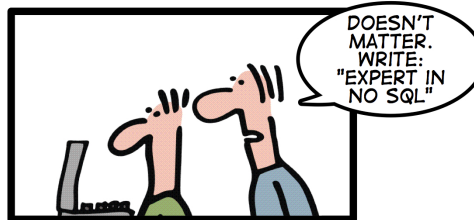
Professor:

P. van Oosterom (TUD)

Reviewer:

R. de By (ITC)

HOW TO WRITE A CV



Leverage the NoSQL boom

(Geek And Poke: NoSQL 2011)

Abstract

The relational data model is now more than 40 years old. It's good for very different scenarios and can handle certain types of data very well. But it isn't perfect. An increasing number of non-relational systems are being worked on, collectively called the 'NoSQL movement'. NoSQL is an umbrella term for a loosely defined class of non-relational datastores and best described as 'Not only SQL'. Examples are Google's BigTable and Amazon's Dynamo. These systems may provide advantages over relational databases, but generally lack the relational robustness for those advantages.

In an attempt to modestly contribute to the current research in the field of geographical information systems, this research reports on a comparison of one such NoSQL datastore called Neo4j with the traditional relational database PostgreSQL when storing and querying spatial vector data. The goal of this study was to determine whether a traditional relational database system like PostGIS, or a NoSQL system, such as Neo4j-Spatial, would be more effective as the underlying technology for operating OpenStreetMap data. Neo4j is an embeddable graph database. Embeddable because it can be added to a Java-based application and used just like any other library, and graph database because the data model it uses to express its data is a graph, storing nodes and relationships that connect them, supporting user defined properties on both constructs.

Multiple aspects are of interest when comparing spatial database systems. The evaluation methodology designed to compare the two, involves both objective measurements and subjective measurements based on documentation and experience. The objective tests include processing speed based on a predefined set of queries, disk space requirements, and scalability. Subjective tests include maturity/level of support, stability and ease of use. Meanwhile, the assessment framework applied for the objective tests could be used as a test suite for evaluating the performance and reliability of new spatial datastores. A test environment have been created using the same OpenStreetMap data in both the graph database and the relational database. While the systems have fundamental differences, identical operations have been developed that provides equal results from differently sized study areas.

Regarding the objective measurements, the results of this research show that the graph database is most beneficial when queries can be expressed as traversals over local regions of a graph. Queries that are well-suited to this approach are, for example, shortest path analyses or connectivity queries. Bounding box queries were faster on the relational database. Regarding the subjective measurements, Neo4j provides a lot of functionality. Transaction support is a welcome addition and there are numerous ways to execute queries, for instance using Java, CQL or a geopipeline. The database model is schema-less and allows additions or adjustments to the schema without any major impact on the data model. As a Java component, it is relatively easy to implement Neo4j-Spatial as an embedded component in any Java program. PostgreGIS is by far the more mature database with a lot of functionality, documentation and support.

Overall, this research shows that, in some cases, Neo4j should be considered as an alternative for specific tasks. A buyer's guide has been created in the form of a flow diagram to decide whether Neo4j-Spatial or PostGIS is suitable for a project. The two technologies, relational databases and non- relational database, will remain in usage side by side, each with the perfect fit for its own capabilities.

Keywords: GIS, Geo, databases, NoSQL, Neo4j, PostgreSQL, measurements.

Acknowledgements

This is a Master thesis on a NoSQL system called Neo4j, its capabilities, advantages and disadvantages when using it to store and query OpenStreetMap data. This study and research tasks started in July 2011 and were completed in May 2012.

First of all, I would like to thank my supervisor Wilko Quak for sharing with me his views and ideas on the thesis. Your comments have helped me to create this final product. Furthermore, I want to thank professor Peter van Oosterom for his critical remarks and for providing dozens of useful literature.

It has been hard work with time conflicts between work-life, study-life and sometimes a personal life, but I enjoyed every bit of my research. I would like to thank my parents, Willem and Petra Baas, for all the support they have given me during the entire duration of GIMA and of course Maarten Hulskemper for his big time support in improving my English grammar. Finally, special thanks goes to my girlfriend Daphne de Groot for her love and patience and her never failing ability to make me relax which made that I never lost motivation.

Best Regards,
Bart Baas

Table of Contents

Abstract.....	V
Acknowledgements.....	vii
List of Abbreviations.....	xi
1 Introduction.....	1
1.1 Problem description.....	1
1.2 Research objectives.....	3
1.2.1 Research questions.....	3
1.2.2 Deliverables.....	3
1.2.3 Scope.....	3
1.3 Methodology.....	4
1.4 Reading guide.....	5
2 NoSQL.....	7
2.1 NoSQL model.....	7
2.1.1 Consistency, availability and partition tolerance.....	9
2.1.2 Querying.....	10
2.2 Comparison.....	10
2.2.1 CouchDB with GeoCouch extension.....	11
2.2.2 MongoDB with geohashing.....	11
2.2.3 Neo4j with the spatial plugin.....	11
2.2.4 Conclusion.....	12
3 Conceptual framework.....	13
3.1 Assessment system.....	13
3.2 Environment.....	14
3.2.1 Software.....	14
3.2.2 Datasets.....	16
3.2.3 Hardware.....	18
3.3 Tests.....	18
3.3.1 Scale regions.....	18
3.3.2 Spatial bounding box count operations (B).....	20
3.3.3 Spatial bounding box get operations (G).....	20
3.3.4 Closest point operations (C).....	20
3.3.5 Shortest path operations (P).....	21
4 Neo4j internals.....	23
4.1 Graph theory.....	23
4.2 Neo4j.....	25
4.2.1 Nodes and Relationships.....	25
4.2.2 Data operation.....	26
4.2.3 File storage.....	26
5 Implementation.....	29
5.1 Environment configuration.....	29
5.1.1 Neo4j dashboard.....	29
5.1.2 PostGIS dashboard.....	31
5.2 Data structure.....	32
5.2.1 Geography.....	32
5.2.2 Neo4j.....	32
5.2.3 PostGIS.....	37
5.3 Geometry dissimilarities.....	39
5.4 Routing topology.....	42
5.5 Tests.....	45
5.5.1 Empty operation.....	45
5.5.2 Test B – bounding box count.....	46

5.5.3 Test G – bounding box get.....	46
5.5.4 Test C – closest node.....	47
5.5.5 Test P – shortest path.....	48
6 Measurements.....	51
6.1 Objective measurements.....	51
6.1.1 Storing OSM data.....	51
6.1.2 Creating a routing network topology.....	52
6.1.3 Spatial operations.....	53
6.1.4 Up-scaling.....	54
6.2 Subjective measurements.....	55
6.2.1 Maturity and level of support.....	55
6.2.2 Stability.....	56
6.2.3 Usability.....	56
6.3 Neo4j versus PostGIS.....	58
7 Conclusions.....	61
7.1 Wrapping up.....	61
7.2 Future research.....	63
References.....	64
Appendices.....	
Appendix I Brewer's (CAP) Theorem.....	
Appendix II Graph internals.....	
Appendix III Neo4j OSM data structure.....	
Appendix IV PostGIS OSM table structure.....	
Appendix V Maven pom file.....	
Appendix VI OSM import differences.....	
Appendix VII OSM equal geometries.....	
Appendix VIII Default.style file.....	
Appendix IX Neo4j StyleReader class.....	
Appendix X Neo4j OSMImporter (improved).....	
Appendix XI Creating route topology.....	
Appendix XII Networks.....	
Appendix XIII Source code, GUI's part.....	
Appendix XIV Source code tests.....	
Appendix XV Results of the operations.....	
Appendix XVI Tables of measurements.....	
Appendix XVII Memory measurements Neo4j.....	

List of Tables

Table 1: Input coordinates bounding box operation (WGS84).....	20
Table 2: Input coordinates closest point operation (WGS84).....	20
Table 3: Input coordinates shortest path operation operation (WGS84).....	21
Table 4: Graph types (Rodriguez and Neubauer 2010).....	24
Table 5: Additional JVM parameters for a Neo4j Java servlet.....	30
Table 6: Neo4j-Spatial layers data structure.....	35
Table 7: Neo4j-Spatial dynamic layers data structure.....	35
Table 8: Neo4j-Spatial geometries data structure.....	36
Table 9: Neo4j-Spatial ways data structure.....	36
Table 10: Neo4j-Spatial nodes data structure.....	36
Table 11: Neo4j-Spatial users data structure.....	36
Table 12: Neo4j-Spatial change-set data structure.....	37
Table 13: PostGIS point data structure.....	38
Table 14: PostGIS linestring data structure.....	39
Table 15: PostGIS polygon data structure.....	39

Table 16: PostGIS network data structure.....	39
Table 17: Initial results importing OSM file.....	40
Table 18: Final results importing OSM file (Medemblik).....	42
Table 19: Results importing OSM file.....	51
Table 20: Results spatial operations.....	53
Table 21: Results primitives Neo4j.....	54

List of code snippets

Snippet 1: OpenStreetMap file.....	16
Snippet 2: A small graph.....	26
Snippet 3: Import OSM data.....	33
Snippet 4: Bash script for OSM to PostGIS.....	38
Snippet 5: Closed line logic (improper).....	41
Snippet 6: Closed line logic (proper).....	41
Snippet 7: OSMImporter modifications.....	42
Snippet 8: Create routing network in Java.....	43
Snippet 9: Create a routing network in SQL.....	44
Snippet 10: Empty operation call: Java SQL.....	45
Snippet 11: Bounding box count operations: Java SQL.....	46
Snippet 12: Bounding box get operations: Java SQL.....	47
Snippet 13: Closest point operations: Java SQL.....	48
Snippet 14: Shortest path operations: Java SQL.....	49
Snippet 15: Neo4j resolving a transaction.....	56

List of Figures

Figure 1: The research process.....	4
Figure 2: The NoSQL family (Sankar 2010).....	7
Figure 3: CAP Two nodes in a network (Browne 2009).....	9
Figure 4: CAP Two nodes in a network messaging (Browne 2009).....	9
Figure 5: CAP Two nodes in a network in practice (Browne 2009).....	10
Figure 6: High level assessment system.....	14
Figure 7: Neo4j architecture.....	15
Figure 8: PostGIS architecture.....	16
Figure 9: About the hardware.....	18
Figure 10: OpenStreetMap dataset boundaries.....	19
Figure 11: A simple graph.....	23
Figure 12: Graph type morphism (Rodriguez and Neubauer 2010).....	24
Figure 13: An example of a Neo4j graph.....	25
Figure 14: Neo4j file storage.....	27
Figure 15: Neo4j dashboard.....	30
Figure 16: PostGIS dashboard.....	32
Figure 17: Neo4j's high-level data model for OSM data (simplified).....	33
Figure 18: Neo4j raw data.....	34
Figure 19: Initial import OSM data.....	40
Figure 20: Routing topology Amsterdam.....	44
Figure 21: Measurements – importing OSM data.....	52
Figure 22: Measurements – creating a route topology.....	52
Figure 23: Neo4j memory footprint.....	55
Figure 24: Neo4j versus PostGIS flow diagram.....	59

List of Abbreviations

Some frequently used abbreviated terms:

ACID	Atomic, Consistent, Isolated, Durable.
BASE	Basically Available, Soft state, Eventually consistent.
CPU	Computer Processing Unit.
CQL	Common Query Language.
DBMS	DataBase Management System
ECQL	Extended Common Query Language. ECQL extends the limitations of CQL, providing a more flexible language with stronger similarities with SQL.
EPSG	European Petroleum Survey Group – they are the authority for a lot of the spatial reference system used. In SRS their initials are prefixed as <EPSG:id>.
GB	Gigabyte.
GEOS	Geometry Engine Open Source – the engine used for many PostGIS advanced functions.
GI	Geo Information.
GIMA	Geographical Information Management and Applications, MSc programme.
GIS	Geographic Information System.
GiST	Generalized Search Tree.
GML	Geography Markup Language, the XML grammar defined by the Open Geospatial Consortium (OGC) to express geographical features.
GPL	General Public License.
GUI	Graphical User Interface.
GWT	Google Web Toolkit
I/O	Input/Output
IDE	Integrated Desktop Environment.
IT	Information Technologies.
MB	Megabyte.
NoSQL	Not Only SQL, non-relational datastore.
ODF	Open Document Format.
OGC	Open Geospatial Consortium – standards body for Geospatial interoperability between products.
OSGEO	Open Source Geospatial Foundation – incubator of many open-source projects.
OSM	OpenStreetMap.
RAM	Random Access Memory.
RDBMS	Relational Database Management System.
REST	REpresentational State Transfer.
s	The second (SI unit symbol: s) is the International System of Units (SI) base unit of time.
SQL	Structured Query Language.
SRID	Spatial Reference Identifier, this is the primary key in a spatial reference system catalog of options.
SRS (id)	Spatial Reference System – this is similar in concept to spatial database SRID, it defines the coordinate system, projection and datum of spatial data.
UML	Unified Modeling Language.
UNIX	Derived from <i>Unics</i> (UNiplexed Information and Computing Service).
WGS	World Geodetic System – standard for defining geodetic data, ellipsoid and geoid models that is the foundation of many spatial reference systems: longitude and latitude data using a reference ellipsoid. Most commonly used is WGS 1984.
WKB	Well-Known Binary, as specified by OpenGIS to express spatial objects.
WKT	Well-Known Text, as specified by OpenGIS to express spatial objects.
XML	eXtensible Markup Language.

1 Introduction

If someone would describe a landscape, it is likely the person would categorize it into roads, fields, trees, buildings or rivers and gives geographical references like 'adjacent to' or 'to the right of' to describe the location of the features. It is impossible for the person to describe all the details of the landscape, the person gives an interpretation based on his own interests. The same applies for digitizing a landscape; it is impossible to represent all the features of a landscape in a computer. Geographical data is a simplified representation of some aspects of the real world. Depending on the use and interests, the data could be from really simple to very complex. (Heywood et al. 2006, chapter 2; Burrough and McDonnell 1998, chapter 2)

As a user would describe a landscape based on interest, the way we store spatial data is also for a specific purpose. This research describes the limitations of traditional database storage and querying from a GI point of view. The construction of a Geographic Information System (GIS) based on a non-relational datastore is proposed. Some successful real world examples exist and promise more flexibility doing operations on the spatial features. The motivation for this approach will be explained in the next chapter.

1.1 Problem description

For several decades, the most common data storage model for geographic data and associated administrative data has been the relational data model. Codd (1970) introduced this term in his paper 'A Relational Model of Data for Large Shared Data Banks'. Software implementing this model is called a Relational Database Management System, or RDBMS, but the less strict term 'relational database' is often used in its place. Strictly speaking, a relational database is a set of tables containing data fitted into predefined categories. Each table contains one or more attributes in columns. Each row contains a unique instance of data for the attribute defined by the columns. Users are able to access or reassemble the data in different ways without having to reorganize the database tables.

Because of their rich set of features, query capabilities and transaction management, RDBMSs seem to fit almost every possible database task. But their feature richness is also their weakness. On the whole, relational databases do a good job in storing data, but there are some limitations (DeCandia et al. 2007; Kim and Lochovsky 1989, page 179):

- **Up-scaling**
Whenever a relational database needs more storage or computer power, it must be loaded on a more powerful computer. When the database does not fit on one single computer, it needs to be distributed across multiple computers. Relational databases are not designed to function with distributed systems; combining multiple tables across different computers is difficult and gives mostly a decrease in performance.
- **Programme and database languages**
Standard Query Language (SQL) is a convenient language to query a relational database. However, the Graphical User Interface (GUI) in a computer program is often developed in a different language, which leads to all kinds of conversion issues.
- **Data without a schema**
Relational databases are good for structured data. For example, sales figures fit well in organized tables. Unstructured data, such as images, word documents and the data of social networking sites are not very suitable for a RDBMS either.

The next paragraphs give a more in-depth view of these limitations.

Up-scaling

With the growth of the internet, distributed computing became more and more useful. Distributed

computing is a technique in which a task is performed by a collection of networked computers. The collection may consist of computer equipment in a single room, or computers at multiple locations, using conventional network technology and the Internet. The idea is to increase overall computing (processing) power by combining the power of individual computers, the system delivers high processing power using relatively inexpensive hardware. By spreading computer locations, only a part of the processing power will be reduced in case a single computer breaks down. A well-known example of a distributed system is Google's MapReduce (Dean and Ghemawat 2008) to produce its index of the World Wide Web.

When it comes to designing a distributed environment, three core systemic requirements exist in a special relationship: Consistency, Availability and Partition-tolerance (CAP). This concept – called the CAP theorem – was first introduced by Eric Brewer (2000) at the PODC¹ conference and was later formalized by Gilbert and Lynch (2002). Proof of the CAP theorem is discussed in paragraph 2.1.1, page 9. According to the theorem, a distributed system can satisfy only two of these guarantees at the same time, not all three. Traditional relational databases fall under the category Consistency and Availability (CA); their stumbling block is a partitioned network. New kinds of databases make it possible to choose between Consistency (CP) and Availability (AP). Examples are Google's BigTable (CP), MongoDB (CP), Amazon's Dynamo (AP), Voldemort (AP), SimpleDB (AP), CouchDB (AP) and BerkeleyDB (CP). As the development of these databases is constantly changing, this list is not complete. (Chang et al. 2008; DeCandia et al. 2007)

Program and database languages

The main differences between programming languages can be explained by their origins. In the computer application development world, object-oriented programming methods and tools have become standard. Object-oriented languages like Smalltalk, C++, Java or Python support the building of applications out of objects that have both data and behavior (Kim and Lochovsky 1989, page 5). Relational technologies support the storage of data in tables or relational schema. Manipulation of data is done via a declarative programming language like SQL, Lisp or XSLT that can be executed both internally and externally via stored procedures. As a result, program languages and database languages are based on different semantic foundations and optimization strategies: 1) The object-oriented paradigm is based on proven software engineering principles, 2) The relational paradigm is based on proven mathematical principles. These differences are known informally as 'Impedance mismatch' (Maier 1990). To a large extent, there is an overlap in the tasks programming languages and databases can perform. (Kim and Lochovsky 1989; Cook and Ibrahim 2006)

Innovative Graphical User Interfaces (GUIs) allow non-programmers to access and manipulate the records in a database. From an end-user's productivity point of view, it is considered a good practice to design GUIs that the non-technical users can execute very complex queries without having to know what happens at the backend. To combine objects and relational databases in such an interface, a developer needs to understand both paradigms, and their differences, and then make intelligent trade-offs based on that knowledge. Most issues of mapping data between databases and programming languages have largely been resolved, but significant issues remain (Cook and Ibrahim 2006).

Schema-less data

Unstructured data refer to information that either does not have a predefined data model and/or does not fit well into relational tables. This results in irregularities and ambiguities which make it difficult to store in traditional relational databases. Especially with user-driven content, it is difficult to pre-conceive the exact schema of the data that will be handled. Unfortunately, the relational model requires upfront schemas and makes it difficult to fit dynamic and ad-hoc data. (Weglarz 2004)

1 PODC is a conference that focuses on research in the theory, design, specification and implementation of distributed systems.

1.2 Research objectives

The issues described in the previous section are driving organisations to look at alternatives to the traditional relational database technology. Collectively, these alternatives have become known as 'NoSQL datastores'. NoSQL datastores may provide advantages over relational databases, but generally lack the relational robustness for those advantages. The aim of this paper is to discover some advantages of a selected NoSQL datastore as compared to a traditional relational database when storing and querying spatial vector data. The choice for the NoSQL datastore will be part of this project. For comparison to the relational model, the open-source project PostGIS will be used. A main objective can be defined to reach this overall goal viz.:

Analyze the advantages of a NoSQL datastore for spatial data as compared to a relational datastore.

The main objective is divided into sub-objectives:

- Examine the technical and geographical capabilities of current NoSQL spatially-aware datastores and consider a suitable NoSQL system for this research;
- Implement and run the chosen NoSQL and PostgreSQL systems containing the same data;
- Run identical spatial bounding box operations on both databases;
- Run identical shortest path operations on both databases;
- Compare the results of the operations on the different databases;
- Conclude on the advantages and disadvantages of storing spatial data in NoSQL.

1.2.1 Research questions

It is clear that the purpose of the two database paradigms are different, but how fundamental are the differences, and what advantages can a NoSQL datastore offer over a relational database? The research objectives, as defined above, lead to the following research question:

When does a NoSQL datastore for spatial data provide clear advantages compared to a relational datastore?

The main question is divided into sub-questions:

- Which NoSQL project could be a good candidate to compare with PostGIS?
- What are the technical characteristics of the chosen NoSQL datastore?
- What are suitable methods to query this NoSQL datastore?
- To what extent is an index structure supported for spatial data in the NoSQL datastore?
- How do spatial bounding box operations perform on both databases?
- How do shortest path operations perform on both databases?
- What are the advantages and disadvantages of storing spatial data in the chosen NoSQL datastore?

1.2.2 Deliverables

This research project has two main deliverables: 1) The design and implementation of a NoSQL system and a PostGIS system, containing identical spatial data and identical spatial operations. Source code will be published via the online project hosting website github. Experiences building and querying these stores will serve as input for the scientific report. 2) Scientific report: The report documents all aspects of the literature research, the development of the operational datastores, and the test results.

1.2.3 Scope

This thesis focuses on what type of project could benefit from the advantages of a NoSQL system over a relational system. It deals with issues of spatial operations on a NoSQL datastore and the performance it

will provide. Common database problems like scalability and cluster distribution of database management may be touched upon, but are not the subject or the goal of this research.

1.3 Methodology

In short, this research is performed by implementing a NoSQL datastore and comparing it with the well-known PostGIS database management system (DBMS) using the same data. This research is divided into two main parts: a theoretical part and an empirical part. The theoretical part consists of literature study of books, web pages and journal papers on the technical aspects of NoSQL. This is needed to select a suitable NoSQL datastore for this research in the first place. The selected NoSQL datastore is the subject of a more in-depth theoretical study afterwards. Because the topic is rather new, some wiki and blog pages are also important sources of information. The empirical part consists of experiences and evaluations of the implementation of the datastores.

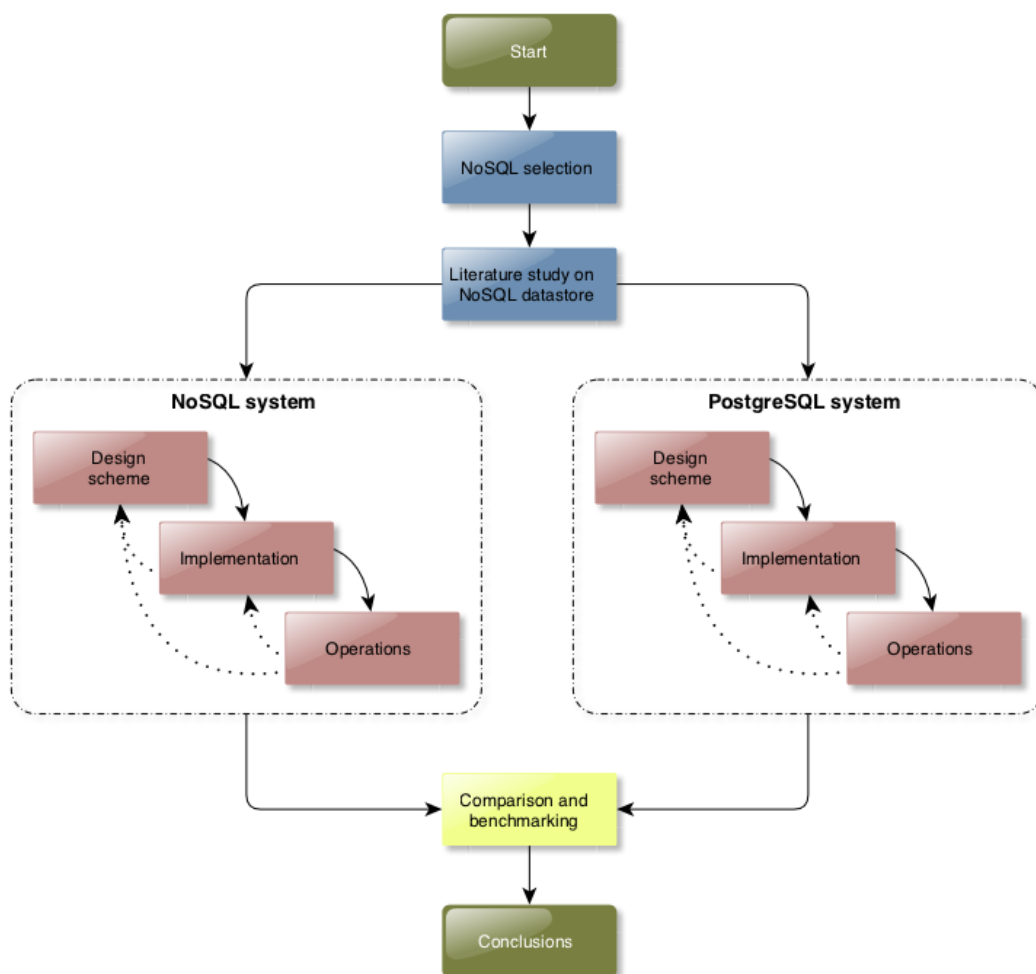


Figure 1: The research process

At first, the currently available NoSQL datastores will be examined in order to make a proper selection. The main decision factors in the decision are reliability, current spatial awareness of the system, ability to store schema-less data, and whether the project is open-source or not. The system that meets these requirements best will be chosen for the research.

After selection, the system chosen will be analyzed more in-depth with several aspects in mind: the data model (which defines how the system stores data), the query model (which examines the strength of the query language(s) used), the consistency aspect (including the trade-offs made in the process), and finally

failure handling (how the datastore handles various types of failures, like user process failures and network problems). To discover the advantages of a NoSQL model compared to a traditional relational model, the theoretical part will also describe the conceptual framework that will be used to perform a fair comparison between the two.

The empirical part will describe the technical details of the design, implementation and evaluation of the dataset using the NoSQL datastore. The spatial functionality offered by both relational and non-relational systems differs significantly in terms of available features, true geodetic support, spatial functions and indexing. Benchmarks play a crucial role in evaluating the functionality and performance of a particular datastore for both developers and end-users. Therefore, a benchmark will be used to evaluate the strengths of the NoSQL datastore. This part of the report includes the preprocessing of the data and setup of the test machines and the results of the benchmark. The theoretical part should give enough input to make a prototype NoSQL datastore. An iterative and incremental development approach will be used to create different prototypes and verify the design phase as soon as possible. Finally, in case when it is possible to query the datastore, the results are compared with the PostGIS database results running the same data.

1.4 Reading guide

The remainder of this document is structured as follows: Chapter 2 covers the NoSQL movement; Chapter 3 details the design of the assessment system; Chapter 4 covers graph basics, how graphs are queried and persisted, and their applications; Chapter 5 details the implementation of the assessment system; Chapter 6 presents the results of the measurements, including a short assessment guide; and finally Chapter 7 provides the conclusions.

Interested readers are encouraged to delve into the References section on page 64 to find more references on these topics, to engage in their own research efforts to understand the characteristics of these systems in the context of their own work.

The OpenStreetMap data model consists of basic elements which are either a *Node*, *Way*, *Relation* or *Tag*. In this report the OSM-specific data primitives are further referred to capitalized to avoid confusion with generic graph elements or plain language.

In this work, NoSQL systems are referred to *datastores*, since the term 'database system' is widely used to refer to traditional DBMSs. As the more general term 'datastore' also includes flat files that can store data, in the scope of this research, the expression 'database' will be used to refer to the stored data in Neo4j and PostgreSQL.

2 NoSQL

The term 'data model' has been used in the information management field with different meanings and in diverse contexts. In its most general sense, a data[base] model is (Silberschatz et al. 1996):

“..a concept that describes a collection of conceptual tools for representing real-world entities to be modeled and the relationships among these entities.”

Often, this term simply denotes a collection of data structure types, or even a mathematical framework to represent knowledge (McGee 1976). In the database literature the terms 'data model' and 'database model' are used interchangeably, but in the scope of this research, the expression 'data model' will be used.

2.1 NoSQL model

The relational model that was proposed by Codd (1970) gives us a clean and concise model to store different types of information. However, an increasing number of non-relational systems are being worked on. The vast majority of these projects have been developed due to relational database products being unable to meet specific requirements (see §1.2 at page 3). Many of these unconventional systems are referred to as 'non-relational' but often are collectively called the 'NoSQL movement'. These systems are very specialized and perform very well for specific applications, but intendedly lack other features. For example, a datastore called MongoDB has lightning-fast data inserts, while being less powerful at batch updates and transactions.

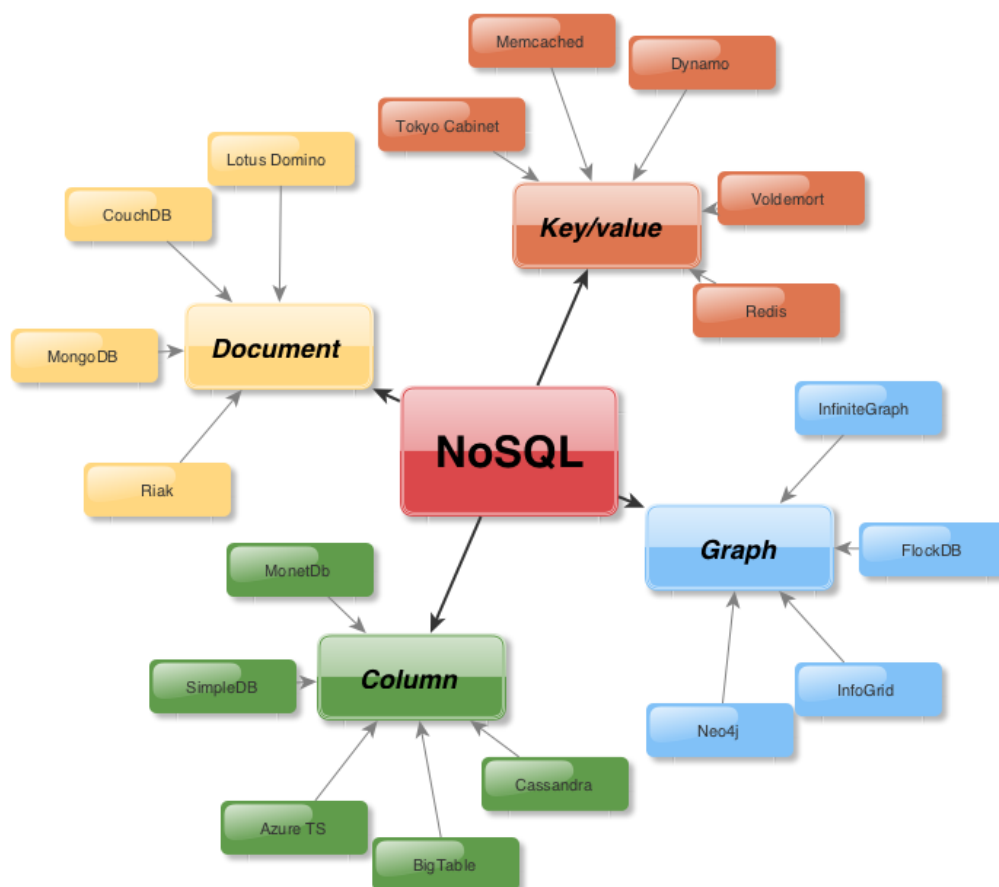


Figure 2: The NoSQL family (Sankar 2010)

To the letter, NoSQL is not a particularly accurate term, as most of the datastores have been developed to have a SQL frontend, and every SQL database could have the SQL interface removed. It really depends on what is meant by NoSQL. Searching the web, very different definitions for NoSQL exist. Carlo Strozzi (1998) used the term 'NoSQL' in 1998 to name his lightweight, open-source relational database that does not expose a SQL interface, but uses various UNIX commands to query the data. Because the database has a relational model, it is therefore not related to the current NoSQL movement. The term was reintroduced in 2009 by Eric Evans during an event (NOSQL meetup 2009) discussing open-source distributed databases. This time, however, it did not refer to a particular system or the query language itself, but rather was a departure from the relational model altogether. Whereas the term sometimes leads to the belief that the makers are 'against' SQL, they are perhaps more correctly described as 'Not Only SQL'. In the free encyclopedia, NoSQL is described as (wikipedia.org 2011):

“...a broad class of database management systems that differ from classic relational database management systems (RDBMSes) in some significant ways. These datastores may not require fixed table schemas, and usually avoid join operations and typically scale horizontally. Academics and papers typically refer to these databases as structured storage, a term that would include classic relational databases as a subset.”

Appropriate or not, the name attempts to describe the increasing number of distributed non-relational databases that have emerged lately. The first attempt to define the term 'NoSQL' is published in *NoSQL: Einstieg in die Welt nichtrelationaler Web-2.0-Datenbanken* (Edlich et al. 2010). In this publication, NoSQL is defined as a new generation of database systems which have at least some of the following properties:

1. The underlying data model is not relational.
2. The systems are designed from the start as horizontally and vertically scalable².
3. The system is open-source.
4. The system is schema-free or has only gentle schema restrictions.
5. Because of the distributed architecture, the system supports a simple data replication method.
6. The system provides a straightforward API.
7. The system generally uses a different consistency model (see §2.1.1), but not ACID.

These conditions give a good context in which the NoSQL field is located, but opinions differ. Sankar (2010) states that NoSQL should be seen as a feature set, now available in very different datastores, but in the future coming to more and more well-known databases.

NoSQL is not a homogeneous domain; the solutions all satisfy very different needs. Some systems, like the document oriented ones, gain an immense ease of use, while most of the key/value or column oriented ones make it easier to distribute data over clusters of computers. Basically, there are several different categories of NoSQL frameworks (Sankar 2010; nosql-databases.org 2011; Edlich et al. 2010):

- Key/value: store data in key/value pairs: very efficient for performance and highly scalable, but difficult to query and to implement real-world problems;
- Column: store data in tabular structures, but columns may vary in time and each row may have only a subset of the columns;
- Document oriented: like key/value, but lets you store more values for a key. A document value could be for example an xml or json fragment. This is a nice paradigm for programmers as it becomes easy, especially with script languages, to implement a one-to-one mapping relation between the code objects and the objects (documents) in the system;
- Graph: stores objects and relationships in nodes and edges of a graph. For situations that fit this model, like hierarchical data, this solution could be faster than the other ones.

Currently, little is known about how these datastores behave when they are used to manage spatial data.

2 Vertically scalable means increasing the capacity of the existing single system. Horizontally scalable means extending the architecture with multiple systems.

Some NoSQL solutions include support for geographical data either natively or with an extension. Others were not initially designed for geographical applications but have been improved to support geographical data.

2.1.1 Consistency, availability and partition tolerance

Many NoSQL systems are designed with the CAP theorem in mind. As introduced in paragraph 1.1, three core systemic requirements exist in a special relationship while designing a distributed environment (Brewer 2000; Gilbert and Lynch 2002):

- *Consistency* means transactions are ACID (Atomic, Consistent, Isolated, Durable). All nodes see the same data at the same time.
- *Availability* means all data must be available and all transactions must come to an end (within a reasonable time). If a node fails, the other nodes continue.
- *Partition-tolerance* is the fault-tolerance between individual components in a distributed environment. This means the system continues even if messages between components are lost.

A distributed system can satisfy only two of these guarantees at the same time but not all three of them. This is visualized in Appendix I. Lets discuss a simple proof of the theorem using the example and pictures of Browne (2009).

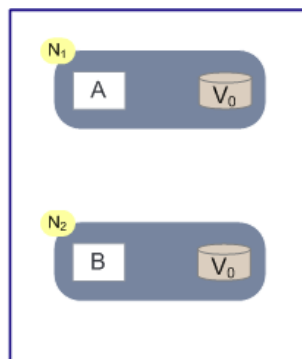


Figure 3: CAP | Two nodes in a network (Browne 2009)

Figure 3 shows two nodes in a network, N_1 and N_2 , both sharing data item V . Lets assume V represents the amount of one particular book title in stock, which has value V_0 . Node N_1 has an algorithm running called A which can be considered safe, bug free, predictable and reliable. The algorithm writes a new value of V , changing it to value V_1 . At the same time, an algorithm called B is running on node N_2 which reads the value of V .

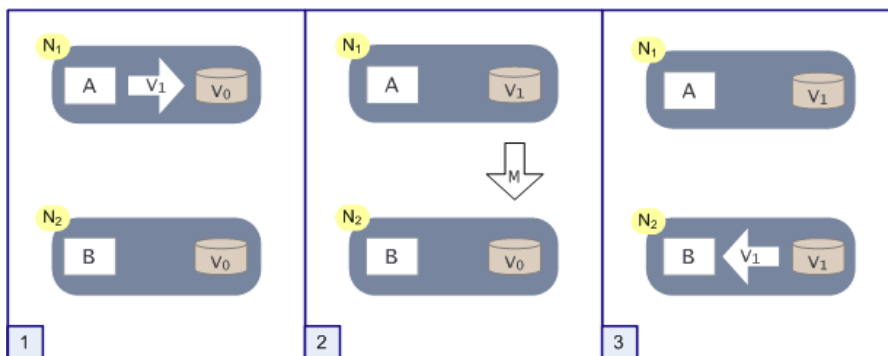


Figure 4: CAP | Two nodes in a network messaging (Browne 2009)

In a perfect world, A writes a new value to V , changing it to V_1 . Then a message M is passed from N_1

to N_2 , updating V . Now any read by B will return value V_1 as illustrated with Figure 4. However, when the network partitions, messages from N_1 to N_2 are not delivered. Node N_2 will contain an inconsistent value of V when step 3 occurs. This is illustrated in Figure 5. In an asynchronous model, it is impossible to provide consistent data: if M is asynchronous, then N_1 has no way of knowing whether N_2 gets the message. Even with guaranteed delivery of M , N_1 has no way of knowing if a message is delayed by a partition event or something failing in N_2 . In a synchronous model, a write by A on N_1 and the update event from N_1 to N_2 will be an atomic operation, which will result in latency issues. Gilbert and Lynch (2002) also prove that even in a partially-synchronous model, with ordered clocks on each node, atomicity cannot be guaranteed.

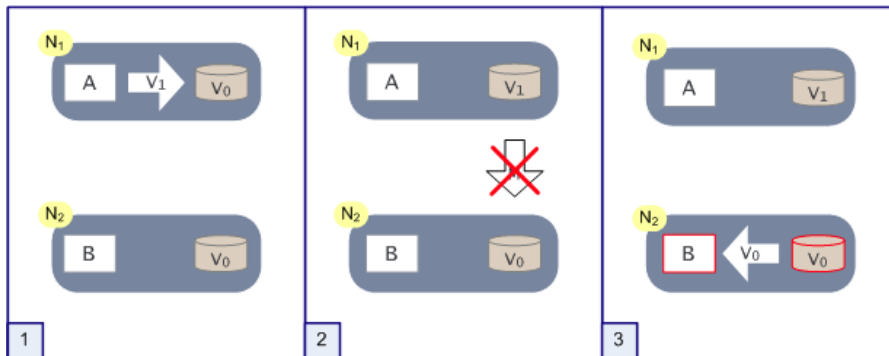


Figure 5: CAP | Two nodes in a network in practice (Browne 2009)

CAP rules do not have to be applied in an 'all or nothing' fashion. Different systems can choose various levels of consistency, availability or partition tolerance to meet their objective. For example, increasing the availability could reduce at the same time partition tolerance or consistency, accepting eventual consistency. As a result, different consistency models are applied. A well known architectural approach of this is known as BASE (Basically Available, Soft-state, Eventually consistent). (Pritchett 2008)

2.1.2 Querying

A NoSQL datastore is of no use if it is impossible to retrieve and show the data to end-users or web services. NoSQL systems do not provide a high-level declarative query language like SQL. Instead, querying or operating these systems is specific to the data model used. Many of the NoSQL platforms allow for RESTful (REpresentational State Transfer) interfaces to the data and others offer query APIs. Some query tools have been specifically developed to attempt to query a variety of NoSQL datastores, although they typically work across a single NoSQL category. Examples are SPARQL for graph stores and UnQL for document stores. (nosql-databases.org 2011)

2.2 Comparison

The first step in this research is to examine the technical and geographical capabilities of current NoSQL systems. One significant problem with the NoSQL family is the lack of a clear definition, which might result in confusion when comparing and discussing the various systems. Four elements are chosen to evaluate the many systems available: reliability, the current spatial awareness of the system, the ability to store schema-less data, and whether the project is open-source or not. The system that best meets these requirements is the PostGIS competitor.

Whereas the non-relational movement started from the open-source world, nowadays well-known companies like Microsoft (Azure Table Storage 2011) or Oracle (Oracle Big Data 2011) are scratching the surface of NoSQL. However, these initiatives are relatively new and closed source. They will not be considered in this comparison. Although the choices for non-spatial data seem to be infinite in the NoSQL world, only a few support spatial data. Currently, the following NoSQL projects support spatial data, at least

to a minor degree: (Kovacs 2010; Schutzberg 2011)

- CouchDB with the GeoCouch extension (Maintainer: Mische)
- MongoDB with geohashing (Maintainer: 10gen Inc.)
- Neo4j using the Neo4j-Spatial plugin (Maintainer: Neo Technology)

The next paragraphs will introduce each project.

2.2.1 CouchDB with GeoCouch extension

CouchDB is a document oriented datastore, currently stable at version 1.1.1 (October 2011) and licensed by Apache. It is written in Erlang and its main focus is data consistency and ease of use. Queries are communicated via the HTTP/REST protocol. (couchdb.org 2011, apache.org 2011)

Spatial data support comes from the GeoCouch spatial extension. GeoCouch has two different kinds of indices: B-tree for key/value queries and R-tree for spatial queries. Currently supported geometries are points, line-strings and polygons. Supported queries are limited to bounding box, polygon and radius searches. It primarily uses MapReduce (Dean and Ghemawat 2008) for querying; while it is easy to use, performance is low, especially for large spatial datasets. When performing spatial queries, the less optimized R-Tree index will be slower than the optimized GiST index in PostGIS. The combination of MapReduce and a less optimized R-Tree makes it highly unlikely GeoCouch is a competitor for PostGIS. (Mische 2011)

Reliability

CouchDB is user-friendly; the installation recently became easier for developers as well as for end-users. The software is available as pre-packed installation packages for many operating systems. GeoCouch is tightly integrated with CouchDB, and to set up a spatial system is as easy as installing the GeoCouch extension.

2.2.2 MongoDB with geohashing

MongoDB is also a document oriented datastore. The stable version is 2.0.1 (October 2011) and it is licensed by AGPL. It is written in C++ and its main focus lies on high performance and retaining some friendly properties of SQL. Queries are communicated via custom binary (BSON) protocol. (mongodb.org 2011)

Spatial data support comes from a spatial index called 'geohashing'. MongoDB supports two-dimensional geospatial indices limited to points only; line and polygon features are not supported at this time. A point-in-polygon search on a table containing points, however, is possible if the polygon is provided as part of the query. Indexing and standard queries in MongoDB are separated from the MapReduce interface, using MongoDB's dynamic query language results in very fast querying. Comparing to the relational reference system, the lack of support for simple spatial features in geohashing is a negative point. Because of this shortcoming, MongoDB will be no competitor for PostGIS. (paolocorti.net 2009, gissolved.blogspot.com 2009)

Reliability

MongoDB is available as pre-packed binary installation packages for different operating systems and the geohashing index is available directly after installation, but only for point data. The software is developed with commercial support available, thus ensuring great reliability.

2.2.3 Neo4j with the spatial plugin

Neo4j is quite different from the others in the sense that it is a graph datastore. The current stable version is 1.5 (November 2011) and the community edition is licensed by GPLv3. It is written in Java and its main focus lies on performance on graphs/relations. Queries are communicated via the HTTP/REST protocol or

directly in Java. (neo4j.org 2011)

Spatial data support comes from a plugin called Neo4j-Spatial, supporting geometry, point, line-string, polygon, multipoint, multi-linestring and multi-polygon simple features, as described by the OpenGIS specification. It makes use of the R-Tree for spatial queries. Neo4j has a unique data model, storing objects and relationships as nodes and edges in a graph. Bounding box queries are supposed to be slower than the optimized GiST index in PostGIS, but operations that fit Neo4j's model (e.g., hierarchical data) are very fast, regardless of the size of the data. Neo4j could have an advantage using certain types of analysis that fit the graph data model, like proximity searches or a route analysis. (Vicknair et al. 2010)

Reliability

Although Neo Technology provides pre-packed binary editions with commercial support and a well-written wiki documentation, spatial support has not yet made it to the pre-packed editions. The only way to use Neo4j with spatial data is to download the source code and compile it. Because of the current developmental stage, spatial queries are attached only partially to the HTTP/REST interface. For now, therefore, the only way to benefit from all spatial features is to use Java.³

2.2.4 Conclusion

While CouchDB and MongoDB have some promising features, their spatial abilities are doubtful. Neo4j-Spatial on the other hand has full-potential spatial advantages, but might not be ready for production. Because of the expected slow performance of CouchDB, the limitation of MongoDB only supporting points, and the potential abilities of the graph data model, the choice ultimately falls on Neo4j as a competitor to PostGIS.

3 Explained in: <http://lists.neo4j.org/pipermail/user/2011-October/012763.html>

3 Conceptual framework

Neo4j is established as a suitable NoSQL competitor for PostGIS, as explained in chapter 2 (page 12). When choosing a spatial database system, multiple aspects are of interest (Quak et al. 2008). This is why the evaluation methodology designed to compare the two involves both objective measurements and subjective measurements based on documentation and experience. The objective tests include processing speed based on a predefined set of queries, disk space requirements, and scalability. Subjective tests include maturity/level of support, stability and ease of use. The objective measurements are described here and start with the definition of the assessment system.

3.1 Assessment system

Although performance is not the only importance aspect, it is still a main criterion. A benchmark is part of the objective tests. Benchmarking can be defined as the process of running a specific program or workload on a specific machine or system, and measuring the resulting performance (Bui et al. 2007). There is a wide array of benchmarking models proposed by different authors. Some of the models have been developed uniquely for a particular type of benchmarking, while others are very generic. Apart from the different benchmarking models, there are also a plethora of classification schemes for benchmarking (Anand and Kodali 2008). Although the core of different benchmarking approaches is similar, most of the authors have tailored their methodology or models based on their own experience and practices (Partovi 1994).

Several studies have been performed on the benchmarking of spatial databases (Stonebraker et al. 1993). Some proposals narrow down a specific topic, such as spatial join algorithms (Hoel and Samet 1995), query performance (Z. Zhou et al. 2009) or indexing (Theodoridis et al. 1998). In the graph database field, benchmarks are discussed and improved as well (Dominguez-Sal et al. 2011). However, all current models lack a comprehensive benchmark framework for spatial vector data. The abilities offered by relational and non-relational databases differs significantly in terms of clustering, available features, spatial reference system support, spatial functions and indexing. A relational data model benchmark would not fit the non-relational datastore, nor would a dedicated graph datastore benchmark fit the relational datastore. Domain-specific benchmarks are a response to compare very diverse computer systems. A high-level assessment model defines the benchmark concepts. The development of a benchmark application requires mapping the performance-specific domain concepts to an implementation and producing complex technology and platform-specific code. The performance on various systems then gives a rough estimate of their relative performance on that problem domain (Gray 1992; Bui et al. 2007).

To be useful, a domain-specific benchmark must meet four important criteria (Gray 1992): relevant, portable, scaleable and simple. Lets discuss these criteria in detail.

Relevant

It must be closely connected or appropriate to the matter at hand, performing operations typical of that problem domain. Because there is no benchmark model currently available for the quantitative comparison in this research, a simple assessment is proposed based on the criteria from Gray (1992). First, the relevancy criterion. The system is based on the principles of many real-world GIS systems in which a client requests spatial queries via a communication protocol; a service executes the spatial queries on the database instance and submits the results to the client. A number of typical spatial queries have been selected to show the relevancy of this problem domain. These spatial queries are described in paragraph 3.3 (page 20).

Portable

It should be easy to implement the benchmark on many different systems and architectures. The portable part has been met by using a free and editable spatial dataset that can be easily implemented. OpenStreetMap provides a free map of the world that can be implemented on many different platforms.

Paragraph 3.2.2 (page 16) describes the dataset in detail.

Scaleable

The benchmark should apply to both small and large computer systems. It should be possible to scale the benchmark to larger systems, and to parallel computer systems, as computer performance and architecture evolve. Differently sized study areas have been chosen to met the scalability criterion, paragraph 3.3.1 (page 18) describes these scales in detail.

Simple

The benchmark must be understandable, otherwise it will lack credibility. The speed of a database system depends on the hardware configuration and the software algorithms used. However, the most important aspect of the efficiency of the database itself lies in the software algorithms rather than raw hardware speed. To eliminate the influence of hardware on the performance of the system, PostGIS is used as a reference system. To create such an experimental environment, the databases are deployed as one assessment pair <Neo4j, PostGIS>. The assessment pair will be configured containing the same spatial data with a client requesting identical spatial data operations, measuring the throughput and response time of every operation. The final result is a normalized value where reliance of databases on hardware is eliminated. This approach, using a well-known reference system, makes the assessment simple and understandable. Whenever needed, it is relatively easy to extend the assessment with other types of databases and on different hardware configurations, whether they are NoSQL or not. For example, extending the comparison with CouchDb would be another pair <CouchDb, PostGIS> on a different platform.

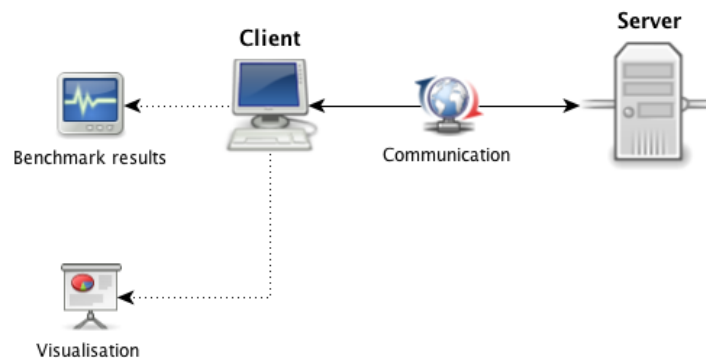


Figure 6: High level assessment system

Figure 6 shows the high-level assessment system that will be applied to the assessment pair. A client calls the database as directly as possible without the use of middleware software or drivers. A singleton database instance accepts tasks from the client, no attempts will be made to optimize the databases. Paragraph 3.2.1 (section 'architecture' at page 15) describes how this theoretical assessment system is applied for Neo4j and PostGIS.

3.2 Environment

3.2.1 Software

The software used in this research is office software, GIS software and development software. Preferably Open-Source Software (OSS) is used with the advantage that the source code and certain other rights normally reserved for copyright holders are provided under a software license that permits users to study, change, improve and also to distribute the software.

As office software used for the reporting part, the LibreOffice software suite is chosen. LibreOffice is a cross-platform open-source office application suite whose main components are word processing, spreadsheets, presentations, graphics and (simple) databases. It supports the OpenDocument Format (ODF) standard natively. References will be managed with the excellent Zotero (zotero.org 2012) research tool to generate citations and bibliographies. The tool integrates nicely with LibreOffice and the Firefox web browser. For creating figures and flow charts, the free yEd diagram editor from yWorks is used.

A download of the OSM dataset comes in an eXtensible Markup Language (XML) with the '.osm' extension. Different tools to edit, export, process and visualize the data exist. The command line Java application Osmosis (wiki.openstreetmap.org/wiki/osmosis 2011) is used for processing OSM data. The tool consists of a series of pluggable components that can be chained together to perform a more complex operation. For example, it has components for reading from and writing to databases and files, components for deriving and applying change sets to data sources and components for sorting data. It has been written so that it is easy to add new features without re-writing common tasks such as file or database handling.

For visualizing purposes, GeoServer (geoserver.org 2012) version 2.1.1 is used. It is an OSS server written in Java that allows users to share and edit geospatial data. Designed for interoperability, it publishes data from any major spatial data source using open standards. It is released under the GNU General Public License (GPL). GeoServer is able to visualize PostGIS data by default, visualizing Neo4j-Spatial has been enabled by updating an outdated GeoServer plugin from Neo Technologies. The source-code of this plugin is available online, at <http://github.com/bartbaas/gt-neo4j-spatial>.

Software development has been done with the open-source Netbeans Integrated Development Environment (IDE) (netbeans.org 2011). All the tools needed to create software with the Java platform are available. Licenses are offered under a dual license of the CDDL (Common Development and Distribution License) and GPL version 2. To simplify the build process, Apache Maven (maven.apache.org 2011) is used as a build tool. Based on the concept of a project object model (POM), Maven can manage a project's dependencies, build, reporting and documentation from a central piece of information. The tool VisualVM (visualvm.java.net 2012) has been used to provide a visual interface for viewing detailed information about Java applications while running on a Java Virtual Machine (JVM).

Neo4j architecture

Neo4j does not have a standard SQL interface; the standard interface language of the Neo4j database is with REST requests or directly in Java. Currently, only part of the full Java API is exposed to the REST API as a server plugin. Data operations for Neo4j are created in the Java language. The configuration of this environment is detailed in paragraph 5.1.1 (page 29).

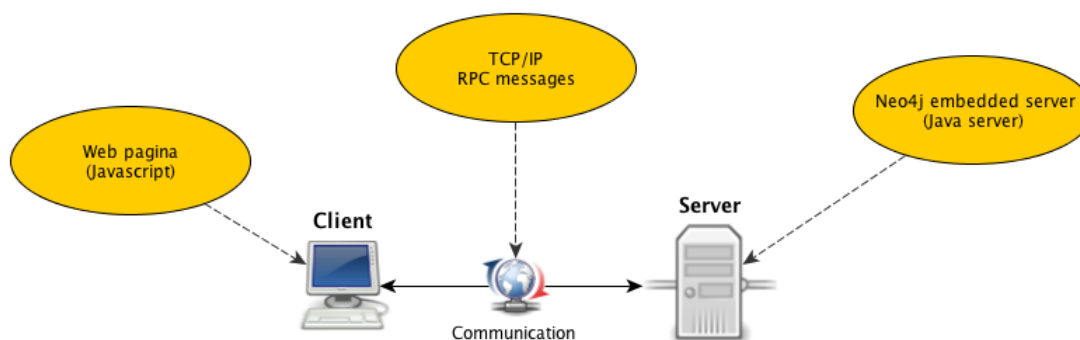


Figure 7: Neo4j architecture

The architecture for Neo4j is visualized in Figure 7.

PostGIS architecture

The relational model is represented by the open-source project PostGIS (postgis.refrations.net 2011), which adds support for geographic objects to the PostgreSQL relational database. It follows the Simple

Features for SQL specification from the Open Geospatial Consortium (OGC). The interface language of the PostgreSQL database is the standard SQL, which allows for inserts, updates and queries of data stored in relational tables. Data operations for PostgreSQL are created in SQL. The configuration of this environment is detailed in paragraph 5.1.2 (page 31).

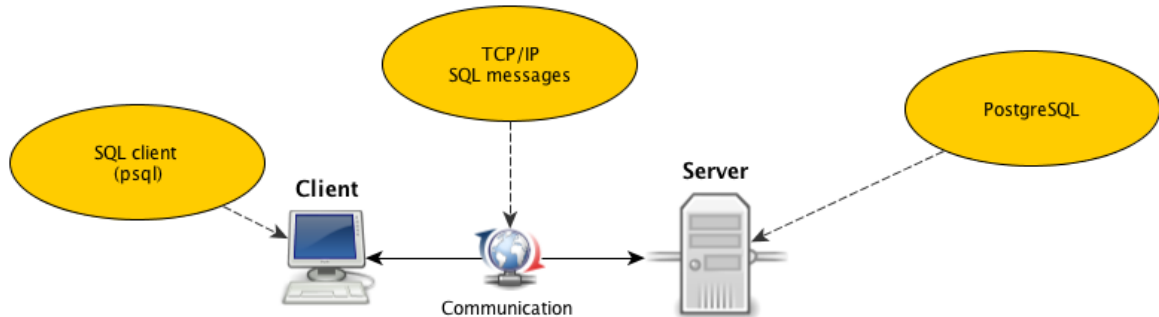


Figure 8: PostGIS architecture

The architecture for the reference system is visualized in Figure 8.

3.2.2 Datasets

The OpenStreetMap (OSM) dataset has been selected for the assessment. OSM creates and provides free geographic data such as street maps to anyone who wants it. The project was started because most maps have legal or technical restrictions, holding back people from using them in creative, productive, or unexpected ways. The license of the data itself is currently the Open Database License (ODBL). Goal of the dataset is to provide geographic data that is free and open to all to use. This means that OSM is a dataset which will enable programmers, researchers, cartographers and the like to fulfill their plans without being limited by any copyright. The dataset has a perfect fit for a research; numerous universities and schools worldwide are using OSM within the scope of research, with the advantage of generating useful documentation in the process. (wiki.openstreetmap.org 2011)

The OpenStreetMap data model consists of basic elements (or: data primitives) which are either a Node, Way or Relation. All of these can have one or more associated Tags and can be included as a member of one or more relations with an optional role. A Node defines a single geospatial point using a latitude and longitude. Nodes can be used to define a standalone point feature, to define the path of a Way, or both. The Way is the second primitive element, being an ordered list of between 2 and 2000 nodes. Ways can be

```
<?xml version='1.0' encoding='UTF-8'?>
<OSM version="0.6" generator="osmosis 0.40.1">
  <node id="34048220" version="2" timestamp="2012-01-28T12:09:51Z" uid="18149" user="Jan Klopper"
    changeset="408573" lat="52.7666808" lon="5.1134922">
    <tag k="created_by" v="Potlatch 0.4c"/>
    <tag k="description" v="DEK , sv, Medemblik NH 31-227-542804"/>
    <tag k="sport" v="soccer"/>
  </node>
  <way id="50689126" version="1" timestamp="2010-02-18T21:36:13Z" uid="195219" user="3dShapes"
    changeset="3912442">
    <nd ref="645521604"/>
    <nd ref="645521597"/>
    <nd ref="645521595"/>
    <nd ref="645520270"/>
    <nd ref="645521604"/>
    <tag k="3dshapes:ggmodelk" v="1"/>
    <tag k="building" v="yes"/>
    <tag k="source" v="3dShapes"/>
  </way>
  <....>
</xml>
```

Snippet 1: OpenStreetMap file

used to represent a linear feature or an area. A Relation consists of an ordered list of Nodes, Ways and

sometimes also other relations as member of the new relation. Relations are used to represent complex geometries (multi-polygons) or to group logical relationships between elements, for instance bus routes. The Relation can have Tags and each element can also optionally have a defined role within the Relation. A single element may appear multiple times in a Relation, and a Relation can be included as member of another. A Tag consists of two free-format textual fields, a 'key' and a 'value', each of which are Unicode strings of up to 255 characters. There are many conventions on how individual features are best described. By way of example, a residential road is defined using a key of 'highway' and a value of 'residential' in the tag `highway=residential`.

As explained at the OpenStreetMap wiki, the website only offers very small areas for downloading, although external sites allow larger amounts of data to be downloaded. One option is the OSM Extended API⁴ (or xapi, pronounced zappy). This is a read-only API based on a modified version of the standard API, which provides enhanced search and querying capabilities. It implements the standard map request and a number of additional ways of querying OSM data by tag values. Xapi uses a REST-style interface with X-path flavoring. Requesting a small area such as Medemblik is possible using wget:

```
wget -O medemblik.osm http://open.mapquestapi.com/xapi/api/0.6/*[bbox=5.0859,52.7566,5.1227,52.7756]
```

Xapi is a good tool to request OSM data, but requesting larger areas results in long waiting times and random http errors. As a result, requesting Amsterdam or larger areas with Xapi was practically impossible. A better option is to download a pre-generated OSM file of a continent or country and use a tool to extract a smaller area. A well-known tool is Osmosis, a command line Java application for processing OSM data. One of the basic use cases is extracting a subset of OSM XML from a large downloaded file. An OSM file of the Netherlands (`netherlands.osm`) is downloaded and a subset of our area of interest is extracted using a bounding box rectangle of the region. Requesting the Medemblik area as an OSM file:

```
osmosis --read-xml file=netherlands.osm --bounding-box top=52.7756 left=5.0859 bottom=52.7566 right=5.1227 --write-xml file=medemblik.osm
```

The binary OSM file of the Netherlands (`netherlands.osm.pbf`) is used as input, this will speed up the tool. The emphasis is on creating two test environments that provides equal results from OpenStreetMap data rather than a complete import. Osmosis has been given additional parameters to create a OSM file that is treated identically by the two database importers. As explained in paragraph 5.3 (page 40) incomplete geometries are trimmed off with the command parameter 'clipIncompleteEntities' and Relations are disregarded with the option 'reject-relations'. The resulting commands used to generate the OSM input files are:

Medemblik area (1) as an OSM file:

```
osmosis --read-pbf file=netherlands.osm.pbf --tf reject-relations --bounding-box top=52.7756 left=5.0859 bottom=52.7566 right=5.1227 clipIncompleteEntities=true --write-xml file=medemblik.osm
```

Amsterdam area (2) as an OSM file:

```
osmosis --read-pbf file=netherlands.osm.pbf --tf reject-relations --bounding-box top=52.4338 left=4.808 bottom=52.3137 right=4.993 clipIncompleteEntities=true --write-xml file=amsterdam.osm
```

North Holland area (3) as an OSM file:

```
osmosis --read-pbf file=netherlands.osm.pbf --tf reject-relations --bounding-box top=52.98 left=4.445 bottom=52.216 right=5.351 clipIncompleteEntities=true --write-xml file=north-holland.osm
```

A download for the Netherlands is available for free from geofabrik at <http://download.geofabrik.de>. This server has data extracts from the OpenStreetMap project which are normally updated every day.

4 <http://wiki.openstreetmap.org/wiki/XAPI>

3.2.3 Hardware

All the tests documented in this report were performed on a standard MacBook Pro, Apple's laptop. This computer's specifications are an Intel Core 2 Duo 2.80 GHz processor and 8 GB internal memory, clock rate at 1067 MHz.



Figure 9: About the hardware

The operating system is Mac OSX 10.7.3. OSX is a POSIX-compliant operating system (OS) built on top of the Mach kernel, with standard UNIX facilities available from the command line interface.

3.3 Tests

This paragraph explains the scale regions and the chosen operations (spatial queries) in detail; these operations and their input parameters are given in paragraphs 3.3.2, 3.3.3, 3.3.4 and 3.3.5. These are well-defined coordinates for which the correct output is known.

3.3.1 Scale regions

As explained in the first paragraph of this chapter, a benchmark should be scaleable. The dataset size for any particular study depends crucially on the scale of the problem and the granularity of the data, this can vary over many orders of magnitude. To capture this diversity, each of the operations are executed for differently sized study areas.

Local area (1): This area typifies the needs of local problems and an extra-small dataset. The geographical region in the benchmark is a 2,5 km x 2,5 km (6,25 km²) rectangle encompassing the small city of Medemblik. The WGS84 coordinates of this area are (longitude, latitude): 5.0859° E, 52.7756° N and 5.1227° E, 52.7566° N.

Urban area (2): This area typifies the needs of an urban problem and a small dataset. The geographical region in the benchmark is a 10 km x 10 km (100 km²) rectangle encompassing the city of Amsterdam. The WGS84 coordinates of this area are (longitude, latitude): 4.808° E, 52.4338° N and 4.993° E, 52.3137° N.

Province area (3): This area typifies a provincial scale problem and a medium dataset. The geographical region in the benchmark is a 60 km x 80 km (4800 km²) rectangle encompassing the province of North-Holland. The WGS84 coordinates of this area are (longitude, latitude): 4.445° E, 52.98° N and 5.351° E, 52.216° N.

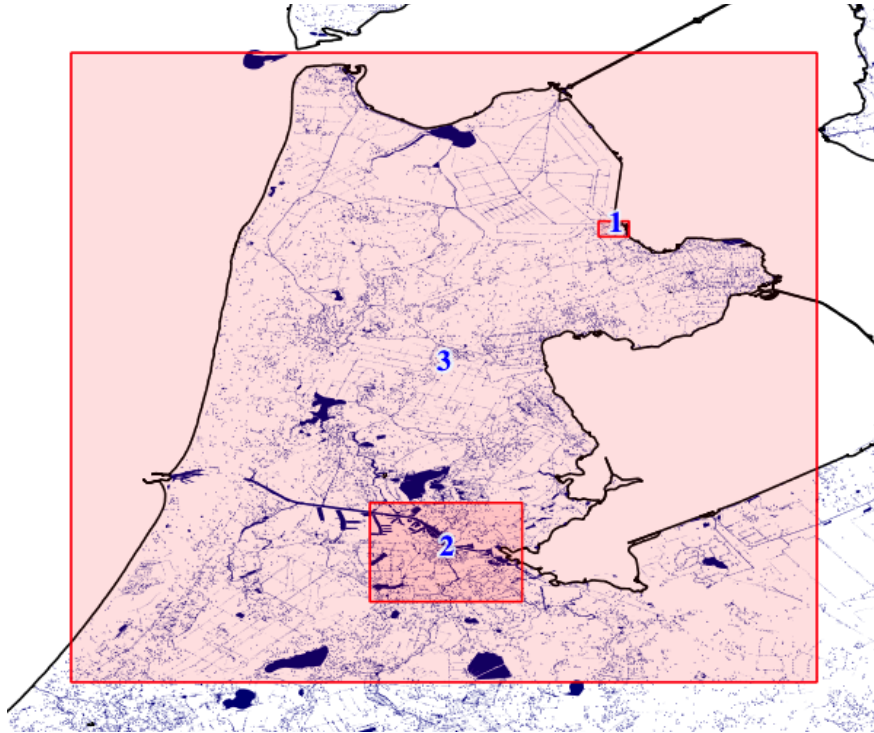


Figure 10: OpenStreetMap dataset boundaries

National area (4): This area typifies a national scale problem and a large dataset. The geographical region in the benchmark is a 270 km x 300 km (81000 km²) rectangle encompassing the Netherlands. The WGS84 coordinates of this area are (longitude, latitude): 3.27° E, 53.53° N and 7.30° E, 50.85° N.

Larger area's, such as a global scale, are not implemented for this research. Figure 10 shows the geographical location of study areas 1 to 3, it is assumed that the location of the Netherlands is known.

3.3.2 Spatial bounding box count operations (B)

Spatial bounding box count operations: Count the number of geometries inside a specific rectangular area. Given a rectangular area A and geometries B , the interior is the set of all objects B which are within A . Using the definition of Güting (1988), it is written as: $B \subseteq A$. Every dataset has input parameters that fits in the scale region.

Table 1: Input coordinates bounding box operation (WGS84)

Medemblik area		Amsterdam area		North-Holland area	
Top-left	Bottom-right	Top-left	Bottom-right	Top-left	Bottom-right
Coordinate WGS84	Coordinate WGS84	Coordinate WGS84	Coordinate WGS84	Coordinate WGS84	Coordinate WGS84
(longitude, latitude)	(longitude, latitude)	(longitude, latitude)	(longitude, latitude)	(longitude, latitude)	(longitude, latitude)
5.09623° E, 52.77227° N	5.10315° E, 52.76724° N	4.88557° E, 52.37674° N	4.91214° E, 52.36694° N	5.09623° E, 52.77227° N	5.10315° E, 52.76724° N
5.10528° E, 52.76338° N	5.10885° E, 52.76078° N	4.84779° E, 52.37478° N	4.86592° E, 52.36498° N	5.10528° E, 52.76338° N	5.10885° E, 52.76078° N
5.09492° E, 52.77134° N	5.11173° E, 52.76151° N	4.94322° E, 52.34834° N	4.96767° E, 52.33037° N	4.81342° E, 52.43037° N	4.98180° E, 52.32305° N
5.10725° E, 52.77271° N	5.11791° E, 52.76455° N	4.81342° E, 52.43037° N	4.98180° E, 52.32305° N	4.73166° E, 52.68332° N	4.98294° E, 52.37542° N

The results of these operations are included in Appendix XV. Corner point boundaries for the Netherlands dataset have not been established due to technical limitations. This is further explained in paragraph 6.1.1.

3.3.3 Spatial bounding box get operations (G)

Spatial bounding box get operations: Request the data of all geometries inside a specific rectangular area as GML (Geography Markup Language) file. This test is equal to the count operations (§3.3.2) except that it requests the data. The same input parameters are applied, see Table 1.

The results of these operations are included in Appendix XV. Corner point boundaries for the Netherlands dataset have not been established due to technical limitations. This is further explained in paragraph 6.1.1.

3.3.4 Closest point operations (C)

Closest point operations: find a node in the data which is closest to the coordinate and calculate the distance from point to coordinate. Given a set of points A in a metric space B and a query point $q \in B$ (Güting 1988), find the closest point in A to q . Every dataset has input parameters that fits in the scale region.

Table 2: Input coordinates closest point operation (WGS84)

Medemblik area	Amsterdam area	North-Holland area
Coordinate WGS84	Coordinate WGS84	Coordinate WGS84
(longitude, latitude)	(longitude, latitude)	(longitude, latitude)
5.09060° E, 52.76522° N	4.8594° E, 52.3576° N	4.87990° E, 52.39310° N
5.10949° E, 52.76080° N	4.8799° E, 52.3931° N	4.94650° E 52.39730° N
5.11160° E, 52.77358° N	4.9465° E, 52.3973° N	5.11160° E, 52.77358° N
5.10554° E, 52.76486° N	4.9412° E 52.3302° N	5.10554° E, 52.76486° N

The results of these operations are included in Appendix XV. Coordinates for the Netherlands dataset have not been established due to technical limitations. This is further explained in paragraph 6.1.1.

3.3.5 Shortest path operations (P)

Shortest path operations: find the shortest path between two known points based on Dijkstra (1959). Required is line network data with connected source and target nodes for every line, as discussed in paragraph 5.4 (page 42). Given a graph G , the length of the shortest path from node i to node j where the distance of every edge is k . It is written as: $G^{(k)}(i, j)$. Every dataset has input parameters that fits in the scale region.

Table 3: Input coordinates shortest path operation operation (WGS84)

Medemblik area		Amsterdam area		North-Holland area	
Start-point	End-point	Start-point	End-point	Start-point	End-point
Coordinate WGS84	Coordinate WGS84	Coordinate WGS84	Coordinate WGS84	Coordinate WGS84	Coordinate WGS84
(longitude, latitude)	(longitude, latitude)	(longitude, latitude)	(longitude, latitude)	(longitude, latitude)	(longitude, latitude)
5.09060° E, 52.76522° N	5.10949° E, 52.76080° N	4.8862° E, 52.3677° N	4.8594° E, 52.3576° N	4.88620° E, 52.36770° N	4.85940° E, 52.35760° N
5.11160° E, 52.77358° N	5.10554° E, 52.76486° N	4.8620° E, 52.3282° N	4.8873° E, 52.3257° N	4.86200° E, 52.32820° N	4.88730° E, 52.32570° N
5.09623° E, 52.77227° N	5.10315° E, 52.76724° N	4.9117° E, 52.4053° N	4.9489° E, 52.3884° N	5.09060° E, 52.76522° N	4.88557° E, 52.37674° N
5.10528° E, 52.76338° N	5.10885° E, 52.76078° N	4.9217° E, 52.3602° N	4.9412° E, 52.3302° N	5.09060° E, 52.76522° N	5.10949° E, 52.76080° N

The results of these operations are included in Appendix XV. Coordinates for the Netherlands dataset have not been established due to technical limitations. This is further explained in paragraph 6.1.1.

4 Neo4j internals

Neo4j is an open-source, embeddable graph database written in Java: 'embeddable' because it can be added to an application and used just like any other library, and 'graph database' because the data model it uses to express its data is a graph, storing nodes and relationships that connect them, supporting user defined properties on both constructs. Although the concept of a graph has been in existence since the late 19th century, it being one of the most generic of data structures, only in recent decades has there been a strong interest in graph ideas. A graph database has been defined by Angles and Gutierrez (2008):

“Graph database models can be defined as those in which data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors.”

Neo4j's internals are explained with these two characteristics in this chapter.

4.1 Graph theory

According to the definition from Angles and Gutierrez, the atomic entity is the graph as a whole. But what is a graph? The birth of the graph theory is ascribed, by many, to the Swiss mathematician Leonhard Euler (1707-1783) while solving the famous Seven Bridges of the Königsberg (now Kaliningrad) problem in 1736. In a graph, there are a set of vertices and a set of edges, where edges are undirected, connect two unique vertices, and no two edges exist between the same pair of vertices. Mathematically, this would be represented as: (West 2001)

$$G=(V, E) \text{ where } V=v_1, v_2, v_n \text{ and } E=e_1, e_2, e_n$$

This definition states that a graph G is composed of a set of vertices V and a set of edges E . Graphically, the vertex can be denoted by a dot and an edge can be denoted by a line. The structure formed by dots and lines is known as a graph. In this definition, the possibility that the two endpoints of an edge are the same vertex is not excluded – this is called a loop – nor are multiple edges ruled out, which is when more than one edge shares the same set of endpoints. The most common type of graph is the simple graph, having neither loops nor multiple edges. Figure 11 shows an example of a simple graph.

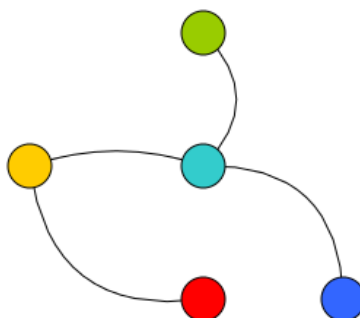


Figure 11: A simple graph

Different types of graphs are created by adding various characteristics to the primitive dots and lines to yield a more flexible or more expressive graph. Adding a label to a vertex to express its name renders the graph more meaningful. This type of graph is called a vertex-labeled graph. When an edge contains explicitly one-way directions, this is called a directed graph. Note that one or more of these characteristics may apply in any given case. A short summary of each graph type is provided in the table below. (Rodriguez and Neubauer 2010)

Table 4: Graph types (Rodriguez and Neubauer 2010)

Graph type	Description
Simple	Prototypical graph. An edge connects two vertices and no loops are allowed.
Undirected	Typical graph. Used when relationships are symmetric.
Multi	Allows multiple edges between the same two vertices.
Weighted	Represents strength of ties or transition probabilities.
Semantic	Models cognitive structures such as the relationship between concepts and the instances of those concepts.
Vertex-attributed	Allows non-relational meta-data to be appended to vertices.
Vertex-labeled	Allows vertices to have labels (e.g. identifiers).
Edge-attributed	Allows non-relational meta-data to be appended to edges.
Edge-labeled	Denotes the way in which two vertices are related (e.g. friendships).
Directed	Orders the vertices of an edge to denote edge orientation.
RDF	Resource Description Framework: graph standard, developed by W3C. Vertices and edges are denoted using Uniform Resource Identifiers (URI).
Half-edge	A unary edge (i.e. an edge connects to one vertex only).
Pseudo	Used to denote reflexive relationships.
Hyper-graph	An edge may connect an arbitrary number of vertices.

The list presented is not exhaustive of all graph types, nor are the terms generally accepted. Many of these structures have been rediscovered in different domains under different names. By not making use of vertex/edge attributes a semantic graph is generated, or by adding weight attributes to edges a weighted graph is generated. This process of creating new graph types by extending/restricting other graph types is referred to as 'graph type morphism' in Rodriguez and Neubauer (2010) as illustrated in Figure 12.

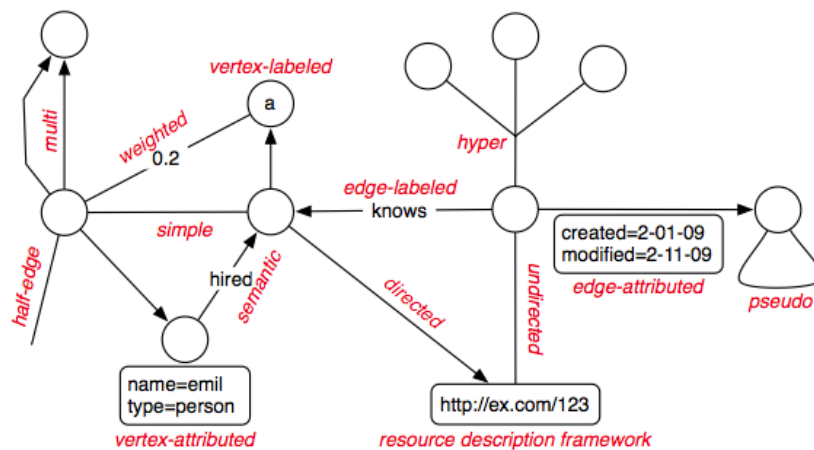


Figure 12: Graph type morphism (Rodriguez and Neubauer 2010)

A common graph type supported by most graph database systems is a combination of the directed-, labeled-, attributed- and multi-graph. This type of graph is also known as a 'property graph' and allows the representation of labeled vertices, labeled edges, and attribute data (or: properties) for both vertices and edges. The property graph is common because by simply abandoning or adding particular characteristics, other graph types can be expressed; for example, by removing the attributed characteristics, the graph

behaves as a labeled graph. Restricting the graph any further by not allowing loops, multiple edges, labels and directionality, a simple graph is created. This is visualized in Appendix II. Neo4j makes use of this property graph data model. To ensure optimal performance in graph traversals, the properties are linked to the data structure as explained in paragraph 4.2.3.

4.2 Neo4j

4.2.1 Nodes and Relationships

As explained in the previous paragraph, Neo4j is a graph database using the property graph data model. This model ensures that edges are directed, vertices/edges are labeled, vertices/edges have associated key/value pair data (i.e. properties), and that there can be multiple edges between any two vertices.

In Neo4j, attributed vertices are called 'nodes' and the directed attributed edges are named 'relationships'. The attributed values themselves are called 'properties'. The nodes, relationships and properties are the building blocks of the data model. A relationship connects two nodes, has a well-defined mandatory type, and, optionally, can be directed. Properties are key/value pairs that are attached to both nodes and relationships which can be either a primitive or an array of one primitive type. Combining nodes, the relationships between them and the properties of both nodes and relationships forms a node space – a coherent network representing the graph data. (Neo Technology 2006)

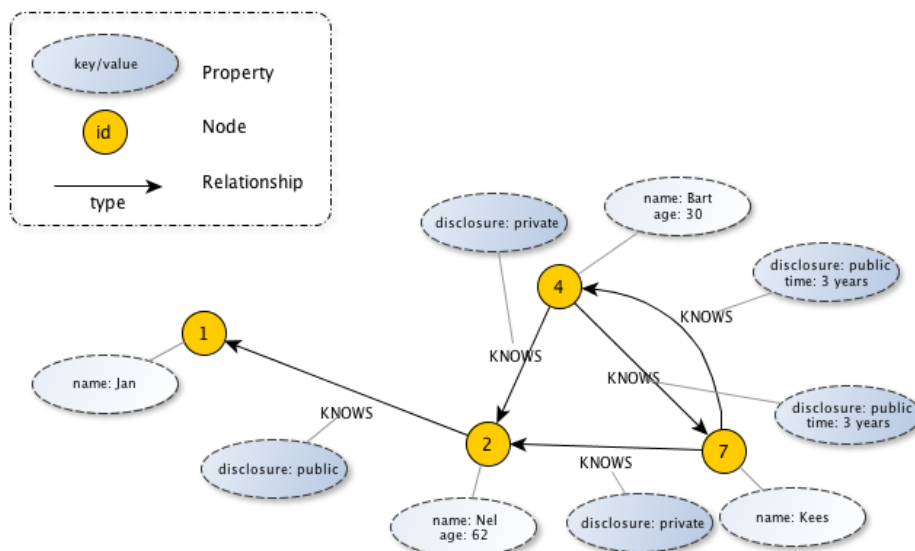


Figure 13: An example of a Neo4j graph

Figure 13 shows a simple property graph. Note how all nodes have ids and how all the relationships have a type. In this case, the two relationship types are KNOWS and BLOCKS, a directed relationship representing which person knows another. All nodes have a name property (seen in the light property boxes) and the relationships have properties describing for how long the people have known each-other and whether the acquaintance is secret (seen in the darker property boxes).

Now, let's create the first two nodes in Java. Neo4j their API documentation is available online (Neo4j-Spatial Components API 2012, Neo4j Community API 2012). Snippet 2 shows how to create a small graph consisting of two nodes, connected with one relationship and some properties. A complete code example can be found in Appendix II.

```
GraphDatabaseService graphDb = new EmbeddedGraphDatabase("var/base");
Transaction tx = graphDb.beginTx();

Node firstNode = graphDb.createNode();
Node secondNode = graphDb.createNode();
Relationship relationship = secondNode.createRelationshipTo(firstNode, MyRelationshipTypes.KNOWS);

firstNode.setProperty("name", "Jan");
secondNode.setProperty("name", "Nel");
secondNode.setProperty("age", "62");
relationship.setProperty("disclosure", "public");
tx.success();

tx.finish();
graphDb.shutdown();
```

Snippet 2: A small graph

Both nodes and relationships can hold properties. Properties are key/value pairs where the key is a string. Property values can be either a Java primitive or an array of one primitive type. For example String, int and int[] values are valid for properties.

4.2.2 Data operation

Unlike a relational database, Neo4j does not support declarative queries at its fundamentals. In database theory, Neo4j would be categorized as a navigational database which means that one navigates from a (given or arbitrary) start node via relationships to the nodes that match one's criteria. This is called traversing.

Traversing a graph means visiting its nodes, following relationships according to some rules making use of index-free, local traversals. In most cases only a subgraph is visited, as rules specify the interesting nodes and relationships to traverse. Typical questions for a traversal are: "How is Nel connected to Kees?", "Where's the nearest Starbucks?" or "What are the cheapest non-stop flights between Amsterdam and Moscow?". Neo4j comes with a callback-based traversal API which lets the programmer specify the traversal rules. A traversal result is a path: one or more nodes with connecting relationships.

Next comes a short explanation of all different methods that can modify or add to a traversal description.

- Expanders: define what to traverse, typically in terms of relationships' direction and type.
- Order: for example, depth-first or breadth-first.
- Uniqueness: visit nodes (relationships, paths) only once.
- Evaluator: decide what to return and whether to stop or continue traversal beyond the current position.
- Starting node: where the traversal will begin.

Traversals are not suitable when one wants to find a specific node or relationship according to a property it has. Rather than traversing the entire graph, an index is used to perform a look-up. The property graph type may still make use of indices to allow for the retrieval of elements from property values. However, the index is only used to retrieve start elements, from which point an index-free traversal is executed through the graph. The way the graph database queries OSM data is explained in paragraph 5.2.2 (page 32).

4.2.3 File storage

Making the data persistent, files are stored to disk. This information is not documented by Neo4j, most of the information in this paragraph is reverse engineered, analyzing the Neo4j files with a hex viewer. This is a simplification of how the system works; there are more files and directories holding the database which will not be mentioned here. The files described below are subject to change but give a good impression of how data is stored in Neo4j at bit level.

First, a word on ids. Ids are implemented as pointers that directly address the location of a record on disk; they are not stored. Ids are stored as longs, which is 8 bytes in Java. Their offset in the file defines their id and their id defines their offset in the file. As a general rule of thumb, the offset in the file can be calculated as: $id * record\text{-}size$.

Neo4j data is stored in one directory containing different files, all filenames starting with 'neostore.'. This is where the building blocks of the data model are made persistent as illustrated in Figure 14. First, the node (N) store will be discussed. This store is implemented at `org.neo4j.kernel.impl.nioneo.store.NodeStore` and creates the file 'neostore.nodestore.db'. Every created node is stored in a fixed size record of 9 bytes in total. The first byte is the use flag. The next 4 are an integer that is the id of the first of the relationships the node participates in. The final 4 bytes are the integer id of the first of the properties this node has.

The relationship (R) store is implemented by `org.neo4j.kernel.impl.nioneo.store.RelationshipStore` and uses the file 'neostore.relationshipstore.db'. It stores relationships (the edges), in a record of 33 bytes. First comes the used byte, with the second least significant bit used as a flag to indicate whether this relationship is directed (1) or not (0). Then 4 bytes, the id of the first node and another 4 bytes as the id of the second node. Next is another 4 bytes, an integer that is the id of the record that represents the type of this relationship. The next 4 integers are: the id of the previous relationship of the first node, the id of the next relationship of the first node, the id of the previous relationship of the second node and finally the id of the next relationship of the second node. The last 4 bytes are the id of the first property of this relationship.

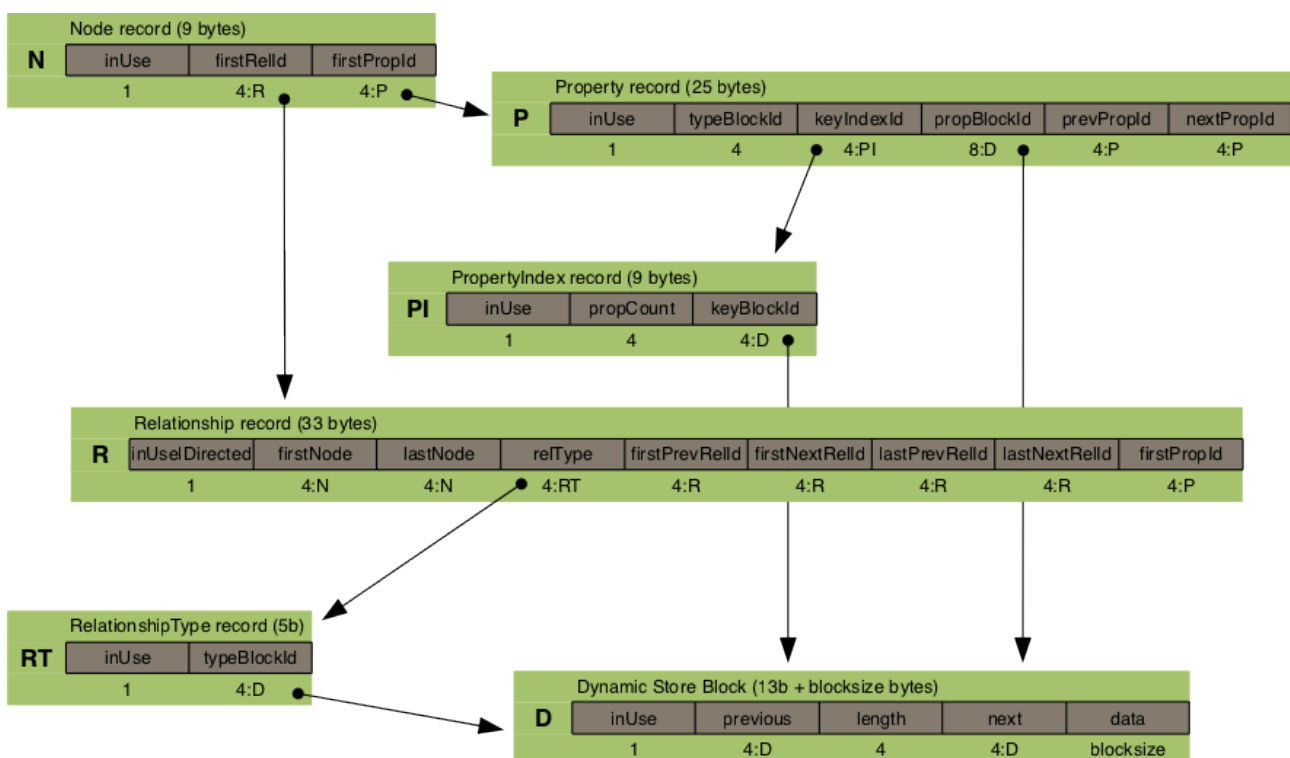


Figure 14: Neo4j file storage

The relationship type (RT) store is implemented by `org.neo4j.kernel.impl.nioneo.store.RelationshipTypeStore` and creates 'neostore.relationshiptypestore.db'. It also needs a place to store the relationship type name (which is a String and therefore of variable size), so it creates a private Dynamic (D) store which stores its data in 'neostore.relationshiptypestore.db.names'. The record size for the type is 5 bytes, the first as the `in_use` byte flag and the 4 remaining the id of the block that stores the String that is the name of the type.

The property (P) storage is primarily managed by `org.neo4j.kernel.impl.nioneo.store.PropertyStore`, which stores its data in 'neostore.propertystore.db'. Each property record begins with the `in_use` byte, then an

integer that stores the type of the property (Java primitives such as int, String, long[], etc, as defined in `org.neo4j.kernel.impl.nioneo.store.PropertyType`), an integer that is the id of the property index, a long that is an id to a DynamicStore (a DynamicStringStore or DynamicArrayStore, depending on the property Type, stored either in 'neostore.propertystore.db.strings' or 'neostore.propertystore.db.arrays') that stores the value or, depending on the Type field, the actual value and finally two integers that are of course ids of the previous and next property of the owning Primitive, for a total of 25 bytes.

The property index (PI) is stored in 'neostore.propertystore.db.index' and is managed by a `org.neo4j.kernel.impl.nioneo.store.PropertyIndexStore` created and held by PropertyStore. Its record starts with an `in_use` byte flag, an integer that keeps a property count and lastly another integer that is the id in a DynamicStore (a DynamicStringStore that keeps data in 'neostore.propertystore.db.index.keys') that keeps the property name. Different Neo primitives can have a common property with a private value if they have the same name. The `key_index_id` of the Property record points to a record in the PropertyIndexStore.

5 Implementation

“Most learning takes place in the process of building the model, rather than after the model is finished.” (Vennix et al. 1997, page 103)

Much effort has been put into creating two test environments that execute equivalent operations on the OpenStreetMap data. The complete source code of the assessment systems is available online:

Neo4j: <http://github.com/bartbaas/gima-neo4jtests>

PostGIS: <http://github.com/bartbaas/gima-postgistests>

5.1 Environment configuration

5.1.1 Neo4j dashboard

Currently, the spatial extension is not included in the main distribution. Neo4j-Spatial is available as a separate download, source code only. The standard interface language of the Neo4j database is with REST requests or directly in Java. Since not all Java API code from the spatial extension is exposed to the REST API, the embedded version of Neo4j is implemented, thus using the Neo4j Java API directly. Because of this limitation, Neo4j-Spatial currently is unfit for a client/server environment, at least 'out of the box'.

A Java application using Neo4j-Spatial has been created during this research. To make sure that the test environment behaves like a client/server system, the Google Web Toolkit (GWT) is used as application framework. GWT is a development toolkit for building browser-based applications in Java. The application runs as a server-side Java servlet while the client asynchronously requests tasks from the server from within a web browser.

A Java project with the spatial extension of Neo4j requires a lot of dependencies. To handle these, Maven is used as build tool. A first template application is created using:

```
mvn archetype:generate \
  -DarchetypeRepository=repo1.maven.org \
  -DarchetypeGroupId=org.codehaus.mojo \
  -DarchetypeArtifactId=gwt-maven-plugin \
  -DarchetypeVersion=2.4.0
```

This generated Maven project contains only GWT dependencies. Those specific to Neo4j-Spatial should be added to the project file 'pom.xml'. Instructions on setting this up are available online (rene-pickhardt.de 2011) and the entire pom file can be found in Appendix V. The resulting project depends on a number of libraries, these are:

- GWT, the Google Web Toolkit version 2.4.0
- Neo4j, generic libraries version 1.5
- Neo4j graphcollections, graph classes version 1.5
- Neo4j-Spatial, spatial support version 0.7
- Geotools, the Java GIS Toolkit version 8.0-M2
- Gremlin, support for pipes version 1.4
- Blueprints, Java reference libraries version 1.1

The operations on the Neo4j database are executed in a Java servlet. Apache Tomcat (tomcat.apache.org 2011) version 7.0.23 is used as Java servlet for all tests. The standard installation is extended with a few additional Java Virtual Machine (JVM) parameters, passed on the command line that starts the instance. JVMs offer a variety of standard and non-standard switches that tune memory allocation and garbage collection behavior. These parameters have been set as prescribed by the Neo4j team in their documentation (The Neo4j Manual v1.5 2011).

Table 5: Additional JVM parameters for a Neo4j Java servlet

JVM parameter	Name	Description
-64	64 bit	Use a 64-bit JVM
-server	Server mode	The server mode has been specially tuned to maximize peak operating speed. It is intended for executing long-running server applications.
-Xms 4G	Initial Java heap size	Initial Java heap size, reserving the maximum heap size.
-Xmx 4G	Maximum Java heap size	Maximum Java heap size, set to 4 Gigabyte.
-XX:+UseParallelGC	The Parallel Scavenger garbage collector	The recommended garbage collector to use when running Neo4j in production. The algorithm is tuned for gigabyte heaps on multi-CPU machines. This collection algorithm is designed to maximize throughput while minimizing pauses.

A control board has been developed to simplify the testing of the operations, called the Neo4j DashBoard. Figure 15 shows a screenshot of the GUI.

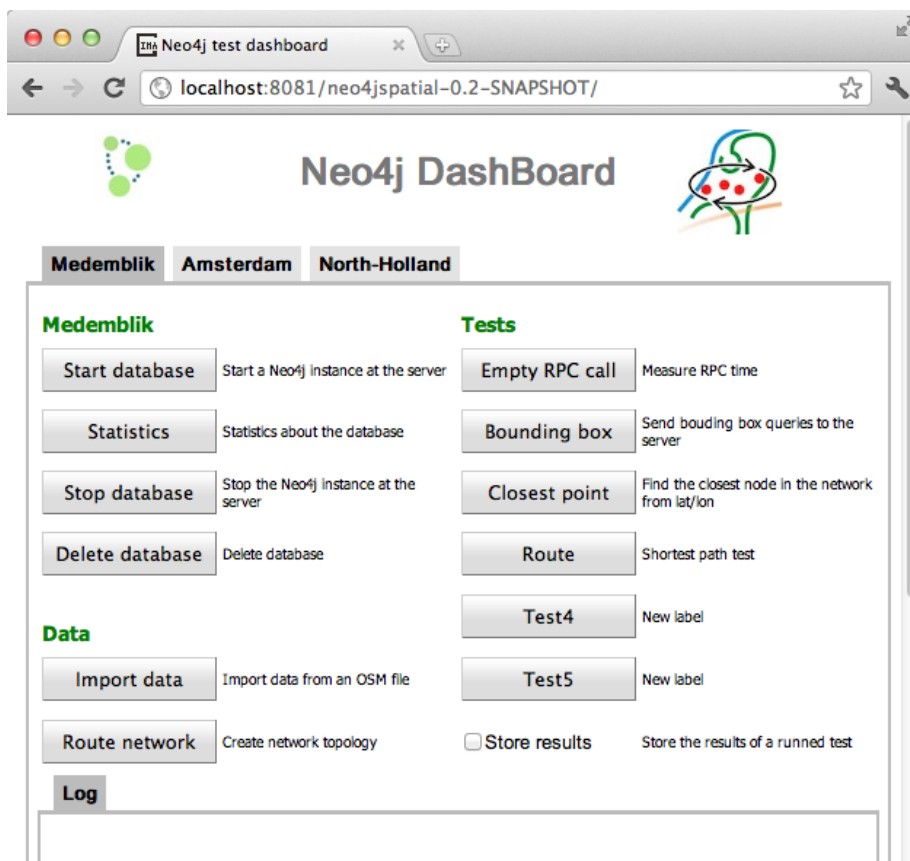


Figure 15: Neo4j dashboard

5.1.2 PostGIS dashboard

Installing the PostGIS database has been done with Homebrew ([mxcl.github.com/homebrew](https://github.com/mxcl/homebrew) 2012), a package management system that simplifies the installation of UNIX software on the Mac OSX operating system. Source-code is downloaded, compiled and installed by packages scripts called 'formulae'. The commands are all executed from a Terminal session:

```
brew install postgis
```

The PostGIS brew formula will also install PostgreSQL, Proj, Geos and some other dependencies. During the process, it will present an explainer about how to start the PostgreSQL server. Create an empty database with:

```
initdb /Users/<username>/data/pgdata
```

Make the database spatial aware with PostGIS (postgis.refractor.net 2011):

```
psql -d <databasename> -f /usr/local/share/postgis/postgis.sql
psql -d <databasename> -f /usr/local/share/postgis/spatial_ref_sys.sql
```

Add the routing engine pgRouting (pgrouting.org 2012):

```
brew install pgrouting
```

Make the database routing aware:

```
psql -d <databasename> -f /usr/local/share/postlbs/routing_core.sql
psql -d <databasename> -f /usr/local/share/postlbs/routing_core_wrappers.sql
psql -d <databasename> -f /usr/local/share/postlbs/routing_topology.sql
```

OSM data is imported into PostGIS using the Osm2pgsql program. Installation is also performed with Homebrew:

```
brew install osm2pgsql
```

The installed versions that were used for this research are:

- PostgreSQL version 9.1.3
- PostGIS version 1.5.3
- pgRouting version 1.05
- Osm2pgsql version 0.8

The operations on the PostgreSQL database are executed with a combination of Bash⁵ shell- and SQL-scripts. The interface language of the relational database is the standard SQL which allows for inserts, updates and queries of data stored in relational tables. A control board has been developed to simplify the testing of the operations, the PostGIS dashboard. Commands selected from the dashboard execute a corresponding bash script which will take care of the assessment and the execution of the SQL script. Figure 16 shows a screenshot of the GUI.

5 Bourne Again Shell, shell for UNIX systems

```

gima-postgistests — bash — 80x24
POSTGIS DASHBOARD - MEDEMBLIK
Spatial tests with Open Street Map data on PostGIS

A) Medemblik      B) Amsterdam      C) North-Holland

Instance
1. Start database
2. Database statistics
3. Stop database
4. Delete database

Data
5. Import osm data
6. Create route topology

Tests
7. Empty SQL call
8. Test1 - bounding box
9. Test2 - closest point
10. Test3 - route
11. Test4 - spatial join
12. Test5 - graph traversal

Note: enter 'cmd' for the psql commandline

Enter your choice [A-C] or [1-12] or q to exit: █

```

Figure 16: PostGIS dashboard

5.2 Data structure

5.2.1 Geography

OpenStreetMap stores the coordinates as a double-precision floating-point type representing the latitude and the longitude of each point, using the WGS84 standard. The basis for the OSM data is a sphere and spatial features are represented on "geographic" coordinates (sometimes called "geodetic" coordinates or "latitude/longitude"), expressed in angular units (degrees).

Geographies are universally accepted coordinates (Burrough and McDonnell 1998, pages 100–101), but the calculations on geographies (areas, distances, lengths, intersections, etc) must be performed on the sphere, using more complex mathematics. For more accurate measurements, the calculations must take the actual spheroidal shape of the world into account, and the mathematics become very complicated indeed. Geographies will correctly handle queries that cover the poles or the international date-line, while a projected dataset on a plane will not.

The OSM data model in Neo4j-Spatial, created by the OSMImporter class, is designed to mimic the complete contents of the XML files provided for OSM. As a result, coordinates in the Neo4j database are also spherical coordinates expressed in angular units (degrees). The import function is currently unable to project the data model on a plane.

Data is stored in both databases as geographies – WGS 84 longitude/latitude (SRID:4326) – using the values from OSM data without conversion. Note that calculations on a sphere are computationally far more expensive than cartesian calculations. For example, the cartesian formula for distance (Pythagoras) involves one call to `sqrt()`. The spherical formula for distance (Haversine) involves two `sqrt()` calls, an `arctan()` call, four `sin()` calls and two `cos()` calls. Geography functions are very costly, and spherical calculations involve a lot of them.

5.2.2 Neo4j

This paragraph describes how OSM data is stored in Neo4j. The way Neo4j-Spatial stores OSM data is not described in the documentation; most of the information in this paragraph has been reverse engineered. The Neo4j-Spatial source code has been analyzed with Netbeans IDE and the created datasets have been visualized with a software package called NeoClipse. A small 'fake' OSM dataset is created containing a tagged node, a railway and a building. This allows us to see the graph in NeoClipse.

Importing OSM data is built-in with the `org.neo4j.gis.spatial.osm.OSMImporter` class and runs in two phases. The first phase requires a batch inserter on the database, and the next phase creates an index on the data. The sequence here is: start batch inserter, import file, shutdown batch inserter, start database, reindex,

```

OSMImporter importer = new OSMImporter("layername");

BatchInserterImpl batchInserter = new BatchInserterImpl(dir, config);
importer.importFile(batchInserter, "data.osm", false);
batchInserter.shutdown();

GraphDatabaseService db = new EmbeddedGraphDatabase(dir);
importer.reIndex(db, 10000);
db.shutdown();

```

Snippet 3: Import OSM data

shutdown database, this is illustrated with Snippet 3.

The resulting data model is designed to mimic the complete contents of the XML files provided by OSM. Every basic OSM element is represented by a unique node. To gain some more insight in how the graph database can be queried, a high-level property graph of OpenStreetMap data in Neo4j is considered (Figure 17). Vertices in this graph represent a group of components, such as layers, OSM primitive or tags. The edges of the graph connect those components that are related. To get the properties of a LineString geometry, one needs to traverse from the geometry node to the way node and finally to the tags node (to get the tags). Thus, it is also possible to gain insight in which OSM ways are created in a particular change-set. This knowledge can be obtained by traversing the graph. For this purpose the Neo4j graph database provides a native Java API and the Gremlin (gremlin.tinkerpop.com 2012) graph programming language. Gremlin is a relatively simple and database-agnostic language with a syntax similar to XPath. Using Gremlin, graph queries can be expressed in a succinct way.

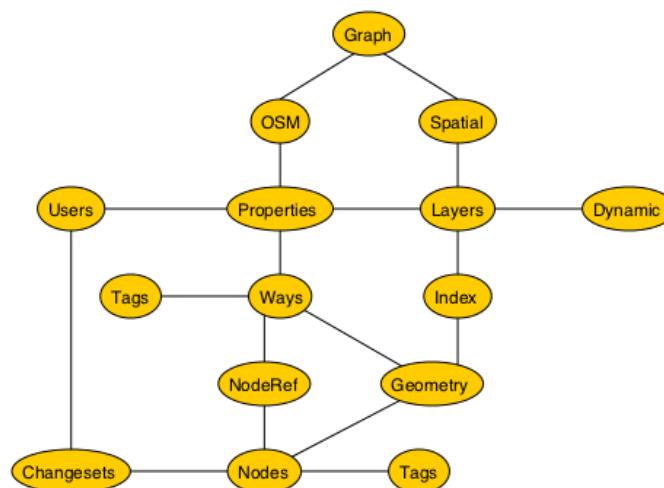


Figure 17: Neo4j's high-level data model for OSM data (simplified)

Using the graph model in a direct way would require precise knowledge on how the OSM data is modeled in Neo4j. Rather, it is better to use the OSM API provided in the `OSMLayer`, `OSMDataset` and `OSMGeometryEncoder` classes. In Neo4j-Spatial every geometry is represented by a unique node, but a node need not necessarily contain coordinates or tags. This is outlined by the `GeometryEncoder` class. The `GeometryEncoder` defines custom approaches to storing geometries in the database graph. There are two primary approaches:

- **In-node:** This approach makes use of properties of an individual node to store the geometry. The built-in WKT (Well-Known Text) and WKB (Well-Known Binary) encoders use this approach, but a custom encoder simply storing a `float[]` of coordinates of a `LineString` would also be classed here.
- **Sub-graph:** This approach makes use of a graph of nodes and relationships to describe a single geometry. This could be as simple as a chain of nodes representing a `LineString` or a complex nested graph like the OSM approach to `MultiPolygons`.

Extending the example above, `org.neo4j.gis.spatial.osm.OSMDataset` provides a method for getting a Way object from a node; the returned object can be queried for its geometries. Because of the nature of the OSM graph, most nodes do not represent point geometries, but are part of complex geometries (streets, regions, buildings, etc.).

The lat/long values are stored quite a bit deeper in the graph. In the case of a Way, there is a chain of reference nodes that runs from the first to the last node of the way. Each of these nodes has a relationship to another node (the OSM Node) that contains the location (lat/long value). The reason for the intermediate nodes is because the location nodes can exist in multiple ways. Most points in the OSM model are not exposed as point geometries in the spatial index. This is because most of them are intended as parts of larger geometries. For example, if someone created a lake in OSM, made of 100 points in a polygon, those 100 points would not be indexed in the spatial index, but the polygon would be. Using the spatial index to find any arbitrary point of the lake will not work, as only points that are tagged individually will appear in the spatial index.

Diving even further in Neo4j-Spatial shows the data types used to store OSM data. Figure 18, below, shows a data dump which is the raw fake OSM data with important parts of the graph annotated. A full resolution image of the graph is included in Appendix III.

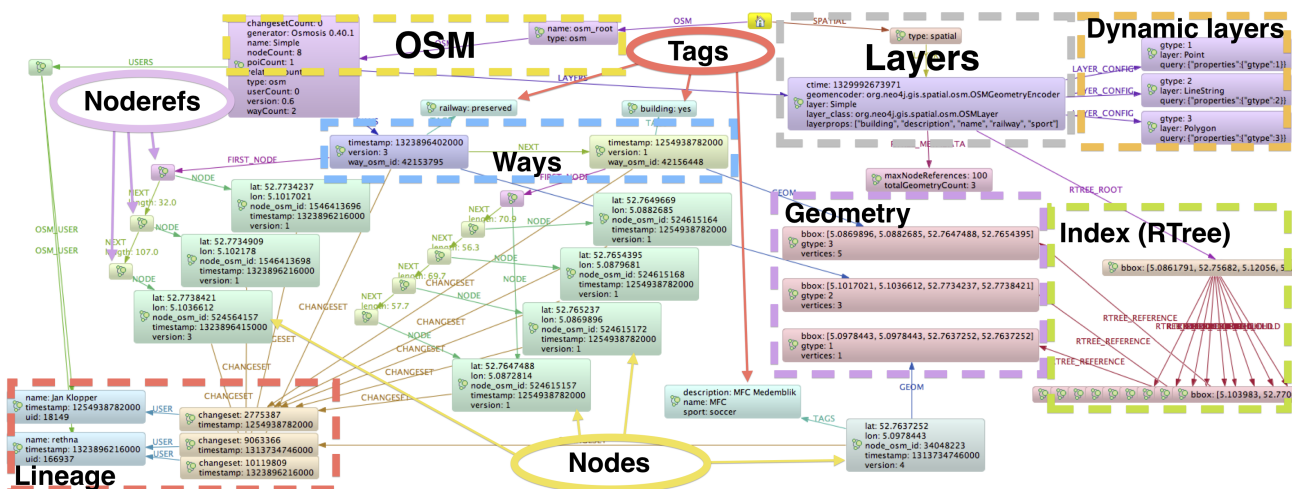



Figure 18: Neo4j raw data

The 'OSM in a graph' data structure is explained as a traversal, navigating from starting nodes to related nodes in the form of: "Nodes are organized by → Relationships, which also have → Properties".

As explained in paragraph 4.2.1, the data model is a property graph containing nodes and relations, and both of them can have key/value style properties. The Neo4j classes provided in `org.neo4j.gis.spatial.osm.OSMLayer`, `org.neo4j.gis.spatial.osm.OSMDataset` and `org.neo4j.gis.spatial.osm.OSMGeometryEncoder` ensure the structure of the graph. After importing the OSM file, as illustrated in Snippet 3, the components described below can be recognized.

An OSM graph database has a → Spatial entry, and also the → OSM-related data

Neo4j comes with a built-in reference node (visualized as  in the Appendix), which is a starting point in the node space. The start node contains the OpenStreetMap part and the spatial part of the data.

The spatial entry is organized by → Layers

This is the spatial part of the database. A spatial graph database can contain multiple layers.

Layers are organized by → Dynamic Layers, and also a reference to the → R-tree Index

This node has information about some basic spatial properties of a layer. Important is the reference to the two classes that determine how the database is constructed: OSMLayer and OSMGeometryEncoder.

Table 6: Neo4j-Spatial layers data structure

Key	Type	Description
ctime	long	Timestamp the layer is created in Unix time
geomencoder	string	The class that is used to create the geometries
name	string	The name of the layer
layer_class	string	The class that is used to create the layer
layerprops	string	All properties that exists in the layer
layercrs	string	Optional: Information about the Coordinate Reference System for this layer. Not defined here for an OSM layer, this is defined in the OSMLayer class

Dynamic Layers

Extends a layer to be able to express itself as several 'sub-layers'. Each dynamic layer is defined by adding filters to the original layer. The filters are configured in the query key. One key example of where this type of capability is valuable is for example when a layer contains geometries of multiple types and the consuming application (desktop or web application) can only express one type in one layer. Then one can use DynamicLayer to expose each of the different geometry types as a different layer.

Table 7: Neo4j-Spatial dynamic layers data structure

Key	Type	Description
gtype	integer	Specifies the geometry type using the OpenGIS geometry type numbers
layer	string	The name of the dynamic layer
query	string	Definition of the filter to the original layer

The R-tree index contains → Geometries

The current index is an R-tree index, using the Java Apache Lucene (lucene.apache.org 2012) software library version 3.1. Typically for an R-tree, every geometry is grouped and represented with their minimum bounding rectangle (double array) in the next-higher level of the tree. Appendix III shows a visualization of the R-tree index in Neo4j of the Amsterdam dataset. While the bounding box is stored as double values in an array, Lucene is originally designed as a word density search engine library. The spatial queries implemented using the R-tree are: contain, cover, covered by, cross, disjoint, intersect, intersect window, overlap, touch, within and within distance.

Geometries are connect to → Ways, or directly to → Nodes

These nodes represent the geometry of spatial entities. Storing OSM data, these nodes are part of other geometries and contain references to the OSM Nodes or Ways. Depending on the type of geometry, it is connected to a OSM Way (linestring, polygon) or an OSM Node (points).

Table 8: Neo4j-Spatial geometries data structure

Key	Type	Description
bbox	double[]	Holds the bounding box of the specific geometry
gtype	integer	Specifies the geometry type using the OpenGIS geometry type numbers
vertices	integer	Specifies the number of vertices of a geometry
wkb	byte[]	Optional: representing vector geometry objects in OGC well-known binary format (WKB)

Ways contain → References to Nodes, and could have → Tags

These nodes represent the OSM Ways from the OSM dataset. Administrative data is stored in a Tag; the geometry itself is defined with references.

Table 9: Neo4j-Spatial ways data structure

Key	Type	Description
timestamp	long	Timestamp as stored in the OSM file
version	string	Identifier as stored in the OSM file
way_osm_id	long	Version as stored in the OSM file

NodeReferences are directly connected to → Nodes

These nodes refer to the OSM Nodes to represent a linear feature or a plane. A closed Way has a start node and an end node referencing the same OSM Node. Note that the length of every segment is stored as a double property in the 'NEXT' relation.

Nodes are connected to → Change-sets, and could have → Tags

Represents the nodes from the OSM dataset and spatial coordinates.

Table 10: Neo4j-Spatial nodes data structure

Key	Type	Description
lat	double	Latitude spherical coordinate expressed in angular units (degrees)
lon	double	Longitude spherical coordinate expressed in angular units (degrees)
node_osm_id	long	Identifier as stored in the OSM file
timestamp	long	Timestamp as stored in the OSM file
version	string	Version as stored in the OSM file

Users contain → Change-sets

Describes the contributor name as a unique node. The user node makes it possible to determine who has made the edits for every individual node.

Table 11: Neo4j-Spatial users data structure

Key	Type	Description
name	string	User name as stored in the OSM file
timestamp	long	Timestamp as stored in the OSM file
uid	long	User identifier as stored in the OSM file

OSM-related data consists of → Users & Ways, and the reference to the → Spatial Layer

Provides non-geographic 'metadata' of the OpenStreetMap dataset.

Change-sets are connected to → Nodes

A group of edits made within a certain time by one user from the OSM dataset. The change-set node makes it possible to determine when edits were made for any given node. Describing the history of a dataset in the form of a note on the data sources and procedures used in the compilation is called lineage information (Burrough and McDonnell 1998, page 94). While the change-set data from OSM does not affect aspects of quality, such as positional accuracy or procedure information, it does show the capability of Neo4j storing lineage information for each and every individual geometry.

Table 12: Neo4j-Spatial change-set data structure

Key	Type	Description
changeset	long	Identifier as stored in the OSM file
timestamp	long	Timestamp as stored in the OSM file
tag	text	Optional: it is possible to extend the change-set with some more tags as described in the OSM standard

Tags

Consist of a 'Key' and a 'Value' as stored in the OSM file. One or more tags can be associated to a Node or a Way, values can contain free format textual fields (strings). Tags are for example used for rendering thematic maps.

5.2.3 PostGIS

This paragraph describes the structure of the PostgreSQL database produced by `osm2pgsql` and the topology script. The import is initialized by a bash script which adds extra spatial functions to PostgreSQL and executes the `osm2pgsql` tool that imports the OSM file.

After the `osm2pgsql` import, these tables are created in the database:

- `planet_osm_line`: contains all imported line ways.
- `planet_osm_point`: contains all imported nodes with tags.
- `planet_osm_polygon`: contains all imported polygon ways.
- `planet_osm_roads`: contains a subset of `planet_osm_line` suitable for rendering at low zoom levels (not used in this research).
- `planet_osm_nodes`: contains the raw OSM node data (not used in this research).
- `planet_osm_rels`: contains the raw OSM relation data (not used in this research).
- `planet_osm_way`: contains the raw OSM way data. This table is not used in the benchmark part of this research, but it is used by the functions that create the network topology.

The topology script (see §5.4 at page 42) adds another two tables to the database:

- `network`: contains the network line-string topology data.
- `vertices_tmp`: contains the network node data. This table is a temporary table for creating the network table.

```
#!/bin/sh
#One-off task to create an empty database
DBNAME=$1
IMPORTOSM=/Volumes/Data/Users/bartbaas/data/OSM/$1.osm

# if exists, remove the previous database
dropdb $DBNAME

# create routing database
createdb $DBNAME
createlang plpgsql $DBNAME

# add PostGIS functions and do this silent
echo "Adding PostGIS functions..."
psql -d $DBNAME -f /usr/local/share/postgis/postgis.sql > /dev/null 2>&1
psql -d $DBNAME -f /usr/local/share/postgis/spatial_ref_sys.sql > /dev/null 2>&1

# add pgRouting core functions and do this silent
echo "Adding pgRouting functions..."
psql -d $DBNAME -f /usr/local/share/postlbs/routing_core.sql > /dev/null 2>&1
psql -d $DBNAME -f /usr/local/share/postlbs/routing_core_wrappers.sql > /dev/null 2>&1
psql -d $DBNAME -f /usr/local/share/postlbs/routing_topology.sql > /dev/null 2>&1

echo "Importing $1 OSM file with osm2pgsql"
osm2pgsql --slim --database $DBNAME --keep-coastlines --style /usr/local/share/osm2pgsql/default.style --latlong
--host localhost --port 5432 --number-processes 2 $IMPORTOSM
psql -d $DBNAME -c "VACUUM"
```

Snippet 4: Bash script for OSM to PostGIS

The OSM raw data tables are present after an import in slim mode (please refer to the command switches of `osm2pgsql` in Snippet 4). They are used in the first import stage; geometry is still in OSM format (lat/lon for Nodes, Node references for Ways) and only minimal conversion from OSM format has taken place. Geometry uses coordinates in the EPSG:4326 coordinate system, as is explained in paragraph 5.2.1. Notice that relations are excluded from this research. Each table has a column containing the geometry ('way') for the object in the chosen coordinate system. Two indices are created for each table: one GiST index for the way column as indicated by '**' and one B-tree index for the `osm_id` column as indicated by '*'. As documented in the PostGIS manual, GiST indices are not a single kind of index, but rather an infrastructure within which different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the operator class). The standard distribution of PostgreSQL/PostGIS includes GiST operator classes equivalent to the R-tree operator classes. (PostGIS 1.5.3 Manual 2011, PostgreSQL 9.1.3 Documentation 2011)

The tables used in this research are described in detail in the coming sections. A full printout of the table structures can be found in Appendix IV.

planet_osm_point

This table contains all nodes with tags which were imported. Nodes without tags (as those whose only purpose is to define the position of a way) are not imported.

Table 13: PostGIS point data structure

Column	Type	Description
OSM_id*	integer	Identifier as stored in the OSM file
tag	text	One column for every tag as specified in the style file, contains the tag value
z_order	integer	Z order (if specified in style file), calculated automatically
way**	geometry	Node geometry (coordinates as WKB)

planet_osm_line

This table contains all non-closed ways which were imported.

Table 14: PostGIS linestring data structure

Column	Type	Description
OSM_id*	Integer	Identifier as stored in the OSM file
tag	text	One column for every tag as specified in the style file, contains the tag value
z_order	integer	Z order (if specified in style file), calculated automatically
way_area	real	Area (if specified in the style file), calculated automatically
way**	geometry	Way geometry (coordinates of all points as WKB)

planet_osm_polygon

This table contains all polygons (closed ways) which were imported.

Table 15: PostGIS polygon data structure

Column	Type	Description
OSM_id*	integer	Identifier as stored in the OSM file
tag	text	One column for every tag specified in the style file, contains the tag value
z_order	integer	Z order (if specified in style file), calculated automatically
way_area	real	Area (if specified in the style file), calculated automatically
way**	geometry	Way geometry (coordinates of all points as WKB)

network

This table contains the route network topology in a table format that is expected by pgRouting, as explained in paragraph 5.4 (page 42). PgRouting's `shortest_path` function needs the source and target nodes in integer format.

Table 16: PostGIS network data structure

Column	Type	Description
gid	serial	Unique identifier
OSM_id	integer	Identifier as stored in the OSM file
name	character	Name of the road as stored in the OSM file
the_geom	geometry	Linestring geometry (coordinates of all points as WKB), for visualization purposes
source	integer	Start node identifier as build by the <code>assign_vertex_id()</code> function
target	integer	End node identifier as build by the <code>assign_vertex_id()</code> function
length	double	Calculated length (or: cost) of the linestring geometry

As the pgRouting functions do rely only in the data in the table, no index is created for the table.

5.3 Geometry dissimilarities

Early tests importing OSM data showed dissimilarities in the number of geometries in the two databases. As explained in paragraph 3.2.2, an OSM XML file format is a list of instances of three data primitives (Nodes, Ways and Relations) associated with Tags. The software used for the import is `osm2pgsql` for

PostGIS and the OSMImport class that is included in Neo4j-Spatial. Initial results of the import tests showed differences in the number of generated geometries in the two databases while using the same (OSM) input file. The differences for the Medemblik dataset, grouped by geometry, are given in Table 17.

Table 17: Initial results importing OSM file

Input file (OSM)		Database		
Geometry	Count	Geometry	PostGIS	Neo4j
Nodes	17650	Points	156	156
Ways	2510	Lines	1000	2107
Relations	40	Polygons	1630	374

Using the same input file, the Neo4j import tool creates more lines in the database, while the PostGIS import tools generate more polygons. The differences are visualized for the Medemblik dataset using GeoServer, high-resolution illustrations are included in Appendix VI.

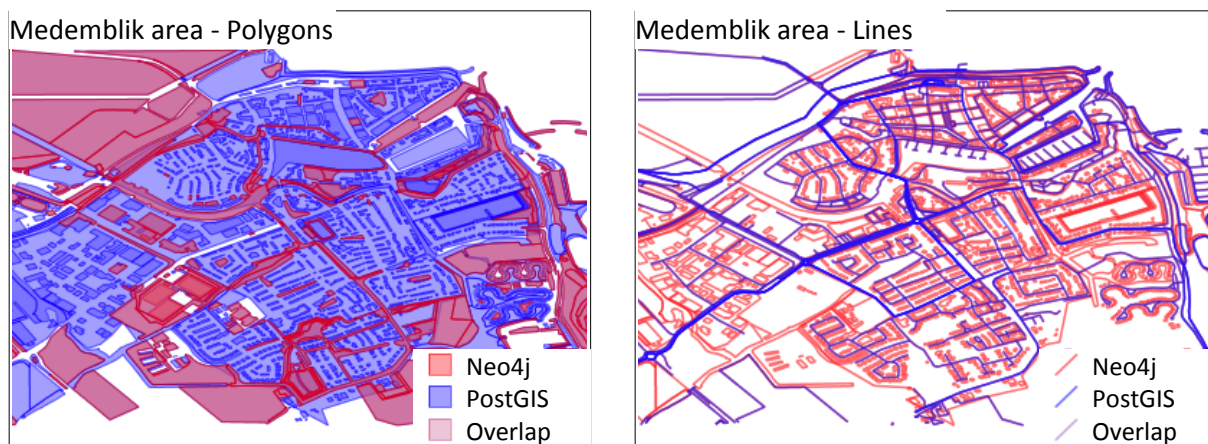


Figure 19: Initial import OSM data

As seen in the illustration above, most Ways are interpreted by PostGIS as a polygon while Neo4j assigns them as a line. The differences are a result of how the data primitives are modeled and categorized by the import tools. Two phenomena can be identified:

1. The extracted OSM files contains missing Nodes and Ways. These errors are managed in different ways by the import tools; and
2. The two import tools use their unique interpretation of the OSM data model.

Phenomenon 1

The first problem has to do with inconsistencies in the downloaded or generated XML file. The complete dataset contains the entire planet (planet.osm) which is a huge file. Naturally, a subset of a smaller area is downloaded or extracted, in this case a subset of the Netherlands. By default, performing a bounding box extraction with Osmosis preserves references in ways or relations inside the bounding box to nodes or members that lie outside the bounding box. Entities are referring to other entities that do not exist in the file. This is just how the software works: it finishes processing Nodes before it works on the Ways and Relations. (comments.gmane.org 2011)

There are several options available to fix this issue. The simplest solution (i) is to drop the constraint from the database after importing. This is more of a hack than a solution and might lead to other issues. Another possibility (ii) is to add the `clipIncompleteEntities=true` option to the bounding box task. This will modify the entities referring to non-existent entities to ensure referential integrity. Ways will be modified

to include only Nodes inside the area and Relations are modified to only include valid members. Obviously they will no longer be identical to the original ways from the input planet.osm file. To preserve the original geometries from the input file (iii), the `completeWays=true` and `completeRelations=true` options could be used instead. This will include all missing nodes outside the bounding box, but will slow down importing enormously. The second option (ii) is chosen (referred to as fix 1) fixing the first identified phenomenon when OSM files have missing nodes and ways.

Phenomenon 2

The second problem is that the OSM data model could be interpreted in different ways. Paragraph 3.2.2 – page 16 – gives an explanation of the OSM data primitives and their associations. For example, Ways may be 'open' where they do not share a first and last Node, or 'closed' where they do. In some cases a closed Way will be interpreted as a 'closed line' and in other instances as an area and in some cases as both a closed line and an area. It is necessary to interpret and review the Tags associated with the Way and in some cases the Tags associated with any Relations associated with the Way. Besides this, areas can also be described using a Relation.

Osm2pgsql distinguishes between ways using a configuration file called the 'default.style'. This file can be found in Appendix VIII. The last column of the import style file controls whether a Way is a line-string or an area. If any of the Tags on an object are marked as 'polygon', the Way is considered as a candidate for polygon rendering. If it is closed, it is treated as an area; if it is open, it is treated as a line-string. On top of that, the area tag controls this behavior directly: `area=yes` always produces an area. In some cases the decision is difficult: for example, the highway Tag has both areas (pedestrian areas) and closed linear features (roundabouts). The practice on the main map is to set the import style to linear and use `area=yes` with area objects. Relations are not controlled with a configuration file but are hard-coded in the software. OSMImporter has a simpler approach. All Ways are considered as a line-string except for the ones that are closed and not tagged with `highway=*`. This clarifies the geometry differences but not all of them: an error was found in the OSMImporter class for the closed line logic.

```
Node firstNode = getOSMNode(nd_ref, changesetNode);
Node prevNode = getOSMNode(nd_ref, changesetNode);

if (firstNode != null && prevNode == firstNode) {
    geometry = GTYPE_POLYGON;
}
```

Snippet 5: Closed line logic (improper)

```
Node firstNode = getOSMNode(nd_ref, changesetNode);
Node prevNode = getOSMNode(nd_ref, changesetNode);

if (prevNode.equals(firstNode) && wayNodes.size() >= 4) {
    geometry = GTYPE_POLYGON;
}
```

Snippet 6: Closed line logic (proper)

Because of the osm2pgsql advantages, it was decided to adjust the OSMImporter class to resemble the osm2pgsql tool. First, the bug in the importer class needs to be fixed. As seen in Snippet 5, the code uses an equality operator `prevNode == firstNode` to check whether the start node and end node are equal. When the objects `prevNode` and `firstNode` have the same value, this comparison would fail because the '==' operator is expected to check if the actual object instances are the same or not. A better option would be to check whether the nodes have the same value. The Java method `equals()` is present in the `java.lang.Object` class and is expected to check for the equivalence of the state of objects by value comparison. Hence, the comparison between `firstNode` and `prevNode` would pass if they have the same value. The enhanced code is displayed in Snippet 6.

The OSMImporter class needs some modifications to interpret the OSM data model in exactly the same way as Osm2pgsql. To achieve this, extra code is added to the OSMImporter class to use the configuration file 'default.style' from osm2pgsql. A new static class is created for this research called StyleReader; the source code is included in Appendix IX. This class reads the configuration file and creates three arraylists: a list of Tags that define the points, a list with Tags that define lines/polygons and a list with Tags that are candidates for polygons. During the import, the tags of every node or way are examined against these lists. If a tag appears in the arraylist, the node or way is processed. Snippet 7 shows the parts of the code

(referred to as fix 2) that distinguish between Nodes and Ways using the configuration file.

```
// Get tag-lists from the configuration file
ArrayList<String> pointTags = StyleReader.readNodes();
ArrayList<String> lineTags = StyleReader.readWays();
ArrayList<String> polyTags = StyleReader.readPolyCandidates();

/**
 * addOSMNode method */
boolean nomineePoint = inList(pointTags, currentNodeTags);
// Check if the node is a point
if (nomineePoint) {
    addNodeGeometry(currentNode, GTYPE_POINT, bbox, 1);
}

/**
 * createOSMWay method */
boolean nomineeLine = inList(lineTags, wayTags);
// Check if the way should be processed or exit
if (!nomineeLine) {
    return;
}

// Check if the way has an area tag
boolean hasAreaTag = false;
if (wayTags.containsKey("area")) {
    hasAreaTag = wayTags.get("area").equals("yes");
}

boolean nomineePoly = inList(polyTags, wayTags);
// Check if the way is a polygon
if (prevNode.equals(firstNode) && wayNodes.size() >= 4) {
    if (nomineePoly || hasAreaTag) {
        geometry = GTYPE_POLYGON;
    }
}
}
```

Snippet 7: OSMImporter modifications

The modified parts of the OSMImporter class can be found in Appendix X. The full class (see: [org.neo4j.gis.spatial.osm.OSMImporter](http://github.com/bartbaas/gima-neo4jtests)) is available online at <http://github.com/bartbaas/gima-neo4jtests>.

After correcting the issues described above, the number of generated geometries in the two databases while using the same (OSM) input file is exactly the same. Table 18 shows the results of the Medemblik dataset with the initial import, the correcting fixes and the final number of geometries. High-resolution illustrations are included in Appendix VII.

Table 18: Final results importing OSM file (Medemblik)

	Initial		Fix 1		Fix 2		Final (fix 1 + 2)	
	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j
Primitive								
Points	156	156	156	156	156	156	156	156
Lines	1000	2107	854	2103	1000	857	854	854
Polygons	1630	374	1617	369	1630	1622	1617	1617

Also in Appendix VII the results of all datasets and the corrected numbers are shown. The OSM data is now ready for the tests.

5.4 Routing topology

Routing engines require connected source and target nodes for every line in order to create a search for the shortest path. Creating this data on line networks involves creating a routing topology on that network; an interconnected set of lines representing possible paths for the movement of resources, people, or traffic from one location to another. The routing engine involves determination of an optimal path across the network from one location to another.

Equal line-networks are an important factor for the comparison; therefore, both systems are prepared to query the same routing network. The networks are created using the 'highway' property in OSM; expressed in ECQL (Extended Common Query Language) this query is:

```
highway is not null
and highway not in ('cycleway','footway','pedestrian','service')
and the_geom IS NOT NULL and geometryType(the_geom) = 'LineString'
```

The OSM data model in Neo4j-Spatial, created by the OSMImporter class, is designed to mimic the complete contents of the XML files provided for OSM. This is not ideal because it traces the complete set of nodes for the ways, while for routing a graph that connects each waypoint by a single relationship is needed. A Java class is developed for this research to create an overlap graph that has the waypoints connected; a full listing of the source-code is found in Appendix XI. This class is called NetworkGenerator and operates by two new layers: 1) adding vertices to a new point layer and checking whether these start nodes and end nodes are already added to the layer, 2) adding references to the line-strings in an edge layer. A 'network' relationship connects the nodes and line-strings. The class is called from a method which

```
// Java snippet to create a routing network
List<SpatialDatabaseRecord> list = OSMGeoPipeline
    .startOsm(osmLayer())
    .cqlFilter("highway is not null and highway not in ('cycleway','footway','pedestrian','service')
              and the_geom IS NOT NULL and geometryType(the_geom) = 'LineString'")
    .toSpatialDatabaseRecordList();

NetworkGenerator networkGenerator = new NetworkGenerator(netPointsLayer, netEdgesLayer, 0.002);
Transaction tx = graphService.beginTx();

Iterator<SpatialDatabaseRecord> it = list.iterator();
while (it.hasNext()) {
    tx = graphService.beginTx();
    try {
        int worked = 0;
        // Every transaction has 5000 spatialdatabaserecords
        for (int i = 0; i < 5000 && it.hasNext(); i++) {
            networkGenerator.add(it.next());
            worked++;
        }
        tx.success();
    } finally {
        tx.finish();
    }
}
```

Snippet 8: Create routing network in Java

performs an iteration on the selected line-strings. Neo4j has built-in route classes based on Dijkstra (1959) to do the shortest path calculation. The routing topology graph used is created by these methods.

Currently, PostGIS has no build-in support for shortest path calculations. A project called pgRouting (pgrouting.org 2012) extends the PostGIS/PostgreSQL geospatial database to provide geospatial routing functionality. Osm2pgsql is a lossy conversion utility. It only adds features that have certain tags, as defined in a config file (default.style), and it converts nodes and ways to line-strings and polygons. The tool is targeted to rendering a map and connections between line-strings are not stored. The data model created by Osm2pgsql is unsuitable for routing analyses. Some tools exist to create the topology. The tool Osm2pgrouting⁶ works well for creating a routing topology in PostGIS, but it is impossible to configure the tool to make a network that is equal to the network in Neo4j. A stored SQL procedure has been developed during this research to create a routing topology in the database. A full listing of this procedure is also to be found in Appendix XI.

⁶ <http://www.pgrouting.org/docs/tools/osm2pgrouting.html>

```

CREATE OR REPLACE FUNCTION create_network() RETURNS text AS $$
DECLARE
streetRecord record;
wayRecord record;
pointCount integer;
pointIndex integer;
geomFragment record;
BEGIN -- start the transaction
FOR streetRecord IN SELECT way, OSM_id, name FROM planet_osm_line
  WHERE highway IS NOT NULL AND highway NOT IN ('cycleway','footway','pedestrian','service') LOOP
  SELECT * FROM planet_osm_ways
  WHERE id = streetRecord.osm_id INTO wayRecord;
FOR pointIndex IN array_lower(wayRecord.nodes, 1)..array_upper(wayRecord.nodes,1)-1 LOOP
  SELECT st_makeline(st_pointn(streetRecord.way, pointIndex), st_pointn(streetRecord.way, pointIndex+1)) AS way
  INTO geomFragment;
  INSERT INTO network(OSM_id, name, the_geom, source, target, length)
  VALUES(streetRecord.osm_id,
    streetRecord.name,
    geomFragment.way,
    wayRecord.nodes[pointIndex],
    wayRecord.nodes[pointIndex+1],
    st_length(ST_GeogFromWKB(geomFragment.way),
    false));
  END LOOP;
END LOOP;
return 'Done';
END;
$$ LANGUAGE 'plpgsql';

```

Snippet 9: Create a routing network in SQL

Two new tables are created by the procedure: 1) an vertices table containing the unique topology nodes and a 2) line-string table containing the roads. The procedure `create_network()` works by creating a street segment for each pair of nodes from the `planet_osm_ways` table. The geometry is created by creating a new line from the points corresponding to the nodes (see the `st_makeline()` call). Once the topology is created, the `pgRouting` function `assign_vertex_id()` is called to compute the source and target node ids. After that, the table is ready for use by `pgRouting`.



Figure 20: Routing topology Amsterdam

Figure 20 shows an example of the routing network in the Amsterdam area. In Appendix XII, all created networks are visualized.

5.5 Tests

In this section, the functions that form the benchmark tests are presented. In each case, the query is expressed in Java and SQL language. This paragraph makes extensive use of the PostGIS Manual (PostGIS 1.5.3 Manual 2011) and the Neo4j-Spatial JavaDoc API documentation (Neo4j-Spatial Components API 2012). It is assumed that the reader understands basic query and aggregate functions on a PostgreSQL database as well as basic Java knowledge.

The complete source code of the test can be found online:

Neo4j: <http://github.com/bartbaas/gima-neo4jtests>

PostGIS: <http://github.com/bartbaas/gima-postgistests>

Some frequently used Java packages (a named collection of classes) are:

`java.*` - The most fundamental classes of the Java language. Any class in this package are referred by its simple name, for example, `java.lang.String` is typed as `String`.

`org.neo4j.*` - The Neo4j Java packages (Neo4j Community API 2012).

`org.neo4j.gis.spatial.*` - The Neo4j-Spatial Java packages (Neo4j Spatial Components API 2012)

`com.vividsolutions.jts.*` - Packages from the Java Topology Suite (JTS) (vividsolutions.com/jts 2012). As Neo4j-Spatial is based on GeoTools, the JTS libraries are used by GeoTools to provide an implementation of the geometry data structure.

5.5.1 Empty operation

One operation is included to measure the response time from a call to the server. This test aims to show that both client/server environments have a similar architecture and to indicate whether a test iteration is valid or not. A command not containing any tasks is sent.

Neo4j uses GWT RPC messages to sent an operation to the server. This call is sent using the `GwtService` class:

```
gwtService.SendTask(task, db, obj, store, callback);
```

```
// Java empty RPC call at server side
public class GwtServiceImpl extends RemoteServiceServlet implements GwtService {
    public String SendTask(Messages.Type type, Messages.Db db, double[][] obj, boolean store) {
        try {
            switch (type) {
                case TEST_EMPTY:
                    return "Nothing to do";
                default:
                    return "<div class=red>Not implemented yet.</red>";
            }
        } catch (Exception ex) {
            return (ex.toString());
        }
    }
}
```

```
-- SQL empty call (test_empty.sql)
-- Sends an empty sql file to the server
SELECT 'Nothing to do' AS TEST_EMPTY;
```

Snippet 10: Empty operation call: Java | SQL

The response time of the PostgreSQL database is measured using an sql file without any operations on the database. The call is sent using the psql command:

```
psql -d $DBNAME -f sql/test_empty.sql
```

5.5.2 Test B – bounding box count

This test has been designed to count all geometries within a specific rectangular area. Identical algorithms are developed for the graph database and relational database.

Neo4j uses the GeoPipeline class to calculate the geometries. In this case, this is a three-step operation:

GeoPipeline – Neo4j uses the TinkerPop (tinkerpop.com 2012) library to process data-flows. A geopipeline is a data-flow framework that enables the splitting, merging, filtering, and transformation of data from input to output. A geopipeline implements a simple computational step that can be composed with other (geo)pipe objects to create a larger computation. All steps have to be implicitly defined.

```
org.neo4j.gis.spatial.pipes.GeoPipeline GeoPipeline(org.neo4j.gis.spatial.Layer layer)
```

.startWithinSearch – Extracts Layer items that are within the given geometry and start a new pipeline with an iterator of SpatialDatabaseRecords.

```
GeoPipeline startWithinSearch(org.neo4j.gis.spatial.Layer layer,  
                             com.vividsolutions.jts.geom.Geometry geometry)
```

.copyDatabaseRecordProperties – Copies item node properties to item properties. Since extracting properties from database nodes can be expensive it must be done explicitly using this pipe. This pipe is useful if you want to process or filter by property values. If there are no keys specified, all keys are added to the result set.

```
GeoPipeline copyDatabaseRecordProperties(String[] keys)
```

.getGeometryType – Calculates geometry type for each item in the pipeline. The property that is used for the output defaults to 'GeometryType' if not set.

```
GeoPipeline getGeometryType(String resultPropertyName)
```

```
// Java bounding box query, bbox and layer are variables  
GeoPipeline pipeline = OSMGeoPipeline  
    .startWithinSearch(layer, layer.getGeometryFactory().toGeometry(bbox))  
    .copyDatabaseRecordProperties()  
    .getGeometryType();  
  
-- SQL bounding box query, bbox is a variable  
SELECT COUNT(*) FROM planet_osm_points WHERE ST_Within(way, ST_MakeEnvelope(:bbox));  
-- etcetera for lines and polygons...
```

Snippet 11: Bounding box count operations: Java | SQL

In PostgreSQL, the PostGIS functions are used to calculate the amount of geometries.

ST_Within – Returns true if the geometry A is completely inside geometry B.

```
boolean ST_Within(geometry A, geometry B)
```

ST_MakeEnvelope – Creates a rectangular polygon formed from the given minima and maxima. Input values must be in SRS specified by the SRID.

```
geometry ST_MakeEnvelope(double precision xmin, double precision ymin,  
                        double precision xmax, double precision ymax, integer srid)
```

5.5.3 Test G – bounding box get

This test has been designed to get all geometries within a specific rectangular area as an GML file. Identical algorithms are developed for the graph database and relational database.

Similar to the previously examined test B, Neo4j uses the GeoPipeline class to calculate the geometries. In this case, this is a two-step operation:

GeoPipeline – Create an empty pipeline.

```
org.neo4j.gis.spatial.pipes.GeoPipeline GeoPipeline(org.neo4j.gis.spatial.Layer layer)
```

.startWithinSearch – Extracts Layer items that are within the given geometry and start a new pipeline with an iterator of SpatialDatabaseRecords.

```
GeoPipeline startWithinSearch(org.neo4j.gis.spatial.Layer layer,
                             com.vividsolutions.jts.geom.Geometry geometry)
```

.createGML() – Encodes item geometry to GML. This pipe is useful to generate GML files from a pipe. At a later stage, the GML fragments are extracted from the pipe using the 'GML' property.

```
GeoPipeline createGML()
```

```
// Java bounding box query, bbox and layer are variables
GeoPipeline pipeline = OSMGeoPipeline
    .startWithinSearch(layer, layer.getGeometryFactory().toGeometry(bbox))
    .createGML();
```

```
-- SQL bounding box query, bbox is a variable
SELECT ST_AsGML(way,3) FROM planet_osm_point WHERE ST_Within(way, ST_MakeEnvelope(:bbox));
UNION ALL
SELECT ST_AsGML(way,3) FROM planet_osm_line WHERE ST_Within(way, ST_MakeEnvelope(:bbox));
-- etcetera for polygons...
```

Snippet 12: Bounding box get operations: Java | SQL

In PostgreSQL, PostGIS functions are used to create the GML fragments.

ST_Within – Returns true if the geometry A is completely inside geometry B.

```
boolean ST_Within(geometry A, geometry B)
```

ST_MakeEnvelope – Creates a rectangular polygon formed from the given minima and maxima. Input values must be in SRS specified by the SRID.

```
geometry ST_MakeEnvelope(double precision xmin, double precision ymin,
                        double precision xmax, double precision ymax, integer srid)
```

ST_AsGML – Returns the geometry as a GML element. The version parameter, if specified, may be either 2 or 3. If no version parameter is specified then the default is assumed to be 2. The 'options' argument is a bitfield but not used for this test.

```
text ST_AsGML(geometry geom, integer version, integer options=0)
```

5.5.4 Test C – closest node

This test is designed to find a node in the OSM data which is closest to the coordinate provided and calculate the distance from node to coordinate. Identical algorithms have been developed for the graph database and relational database.

Similar to the previously examined test B and G, Neo4j uses the GeoPipeline class to calculate the geometries. In this case, this is a three-step operation:

GeoPipeline – Create an empty pipeline.

```
org.neo4j.gis.spatial.pipes.GeoPipeline GeoPipeline(org.neo4j.gis.spatial.Layer layer)
```

.startNearestNeighborLatLonSearch – Calculates the distance between the node and Layer items nearest to it and starts a new pipeline with an iterator of SpatialDatabaseRecords. Note that this method does not necessarily return any results.

```
GeoPipeline startNearestNeighborLatLonSearch(org.neo4j.gis.spatial.Layer layer,
                                             com.vividsolutions.jts.geom.Coordinate point,
                                             double maxDistanceInKm)
```

.sort – Sort items in the pipeline comparing values of the given property. In this case, the orthodromic distance – the shortest distance between any two points on the surface of the earth – is the property on which will be sorted.

```
GeoPipeline sort(String property)
```

.getMin – Computes the minimum value of the specified property and discard items with a value greater than the minimum. Here, the orthodromic distance is also the property of interest.

```
GeoPipeline getMin(String property)
```

```
// Java closest point query, bbox and layer are variables
GeoPipeline pipe = GeoPipeline
    .startNearestNeighborLatLonSearch(layer, coordinate, 0.5)
    .sort("OrthodromicDistance")
    .getMin("OrthodromicDistance");
List<GeoPipeFlow> closests = pipe.toList();
return closests.get(0);
```

```
-- SQL closest point query, point is a variable
SELECT id, ST_Distance_Spheroid(pt, ST_ClosestPoint(geom,pt),
    'SPHEROID["WGS 84",6378137,298.257223563]') as dist
FROM (SELECT ST_GeomFromEWKT(:point)::geometry AS pt, way AS geom, OSM_id AS id
    FROM planet_osm_point) AS points ORDER BY dist LIMIT 1;
```

Snippet 13: Closest point operations: Java / SQL

Using PostGIS, a number of functions are applied:

ST_ClosestPoint – Returns the 2-dimensional node on geometry g_1 that is closest to geometry g_2 . This is the first point of the shortest line.

```
geometry ST_ClosestPoint(geometry g1, geometry g2)
```

ST_Distance_Spheroid – Returns the minimum distance between two lon/lat geometries given a particular spheroid.

```
float ST_Distance_Spheroid(geometry geom1lonlatA, geometry geom1lonlatB,
    spheroid measurement_spheroid)
```

ST_GeomFromEWKT – Constructs a PostGIS geometry object from an input point variable in the OGC Extended Well-Known Text (EWKT) format.

```
geometry ST_GeomFromEWKT(text EWKT)
```

5.5.5 Test P – shortest path

This test finds the shortest path between two known points based on Dijkstra (1959). Identical algorithms are developed for the graph database and relational database.

Neo4j uses the Dijkstra class to calculate the shortest path. A CostEvaluator is defined to use the length property ('_distance') stored in the route topology.

Dijkstra – This class can be used to perform shortest path computations between two nodes. The search is made simultaneously from both the start node and the end node. Note that by default, only one shortest path will be searched for, computation stops when a connected path is found. The search will be initiated when either the path (getPath) or the cost (getCost) is asked for. If at some later time getPath is called, the calculation is redone.

```
org.neo4j.graphalgo.impl.shortestpath.Dijkstra Dijkstra<CostType>(CostType startCost,
    Node startNode, Node endNode, CostEvaluator<CostType> costEvaluator,
    CostAccumulator<CostType> costAccumulator,
    Comparator<CostType> costComparator, Direction relationDirection,
    RelationshipType costRelationTypes)
```

.getPathAsNodes – A call to this method will run the algorithm to find a single shortest path, if not already done, and return it as a list of nodes.

```
List<Node> getPathAsNodes()
```

.getCost – A call to this method will run the algorithm to find the total cost for the shortest paths between the start node and the end node, if not calculated before.

```
CostType getCost()
```

CostEvaluator – A class to calculate the cost associated with choosing a certain path using the stored information in the graph. This algorithm accepts objects (data type 'T') for all relevant tasks regarding costs of paths. This allows the user to represent and calculate the costs in any possible way. Here, the '_distance' property attached to the edges representing road is retrieved.

```
org.neo4j.graphalgo.costevaluator CostEvaluator<T>
```

.getCost – This is the general method for looking up costs for relationships. This can do anything, like looking up a property or running some small calculation. The cost for a particular edge (or: relationship) is returned.

```
T getCost(Relationship relationship, Direction direction)
```

```
// Java shortest path query, startnode and endnode are variables
Dijkstra<Double> sp = new Dijkstra<Double>(0.0, startNode, endNode,

    new CostEvaluator<Double>() {
        public Double getCost(Relationship relationship, Direction direction) {
            Node startNd = relationship.getStartNode();
            if (direction.equals(Direction.INCOMING)) {
                return (Double) startNd.getProperty("_distance");
            } else {
                return 0.0;
            }
        }
    },
    new DoubleAdder(), new DoubleComparator(), Direction.BOTH,
    SpatialRelationshipTypes.NETWORK);

List<Node> pathNodes = sp.getPathAsNodes();
return "Length: " + sp.getCost().longValue() + ", Segments: " + pathNodes.size();

-- SQL shortest path query, start and end are variables
SELECT SUM(cost) AS distance FROM
    shortest_path('SELECT gid AS id, source, target, length AS cost FROM network',
        get_network_id(:start), get_network_id(:end), false, false);
```

Snippet 14: Shortest path operations: Java | SQL

Using the pgRouting functions, the shortest path is calculated:

shortest_path(sql) – The function returns a set of rows. There is one row for each crossed edge, and an additional one containing the terminal vertex. The columns of each row are:

```
path_results shortest_path(sql text, source_id integer, target_id integer, directed boolean,
    has_reverse_cost boolean)
```

sql – a sub-query returning a set of rows from the network edges' table with the following columns:

```
sql text(SELECT id, source, target, cost FROM edge_table)
```

get_network_id – A custom function created for this research. The code finds the closest point in the network topology, and functions very similar to the one explained in Query C. A more extensive code snippet can be found in Appendix XIV.

```
integer get_network_id(text EWKT)
```


6 Measurements

This chapter describes the results of the tests in the objective measurements and maturity and level of support, stability, and usability in the subjective section.

6.1 Objective measurements

Each query was run 10 times on each database and execution times were measured in seconds (s). The longest and shortest times were dropped and the remaining eight times were averaged. The results of the tests themselves can be found in Appendix XV and the measured times can be found in Appendix XVI. No attempts were made to optimize the Java VM, the PostgreSQL database service or the SQL queries. The benchmarks for Neo4j and PostgreSQL will be run 'out of the box', with a 'natural syntax', and 'as documented' for all types of queries.

Before operating on PostGIS or Neo4j, each database had a 'warming up' procedure run on it. In PostGIS, a 'SELECT * FROM geom_tables' was executed and all of the results were iterated through. In Neo4j, every vertex in the graph was iterated through and the outgoing edges of each vertex were retrieved. The tests were run in the following sequence: start operating system – start database – warming up – run test – shutdown operating system. This was done to ensure that any caching or system process activity would not affect the timings.

6.1.1 Storing OSM data

Four OSM datasets are imported in Neo4j-Spatial and PostGIS. Background information about the acquired datasets, the software used and the projection chosen is described in paragraphs 3.2.2, 3.2.1 and 5.2.1. The results of the import is summarized in Table 19. The import tool for PostGIS is optimized and operates quickly; the resulting storage space is acceptable. The importer for Neo4j is less optimized, less fast and the time to import OSM data takes longer as computer power is not efficiently utilized.

Table 19: Results importing OSM file

Name	XML	System		Time		Topology	
	OSM-file	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS
Entity	(MB)	(MB)	(MB)	(s)	(s)	(s)	(s)
Medemblik	4	8	15	9,8	4,0	3,75	1,0125
Amsterdam	106	219	129	600,5	47,9	351,125	429,75
North-Holland	1229	2337	1182	13865,5	663,1	8284,5	27190,375
Netherlands	10691	–	15360	–	28803,6	–	–

Importing the Netherlands dataset with Neo4j took more than 18 hours and resulted in a database that grew to more than 60 gigabyte (GB). With PostGIS, importing took approximately 4 hours, resulting in a database size of 15 GB. The size of the North-Holland dataset seems to be the limit for Neo4j, at least running in this hardware environment. Using larger datasets, a known problem with the Lucene index was encountered as discussed on their forums (Neo4j Community - More spatial questions 2011). The second stage of the import consists of reindexing the database and building a Lucene index. This process slows down exponentially for larger datasets, affecting the scalability of the importer.

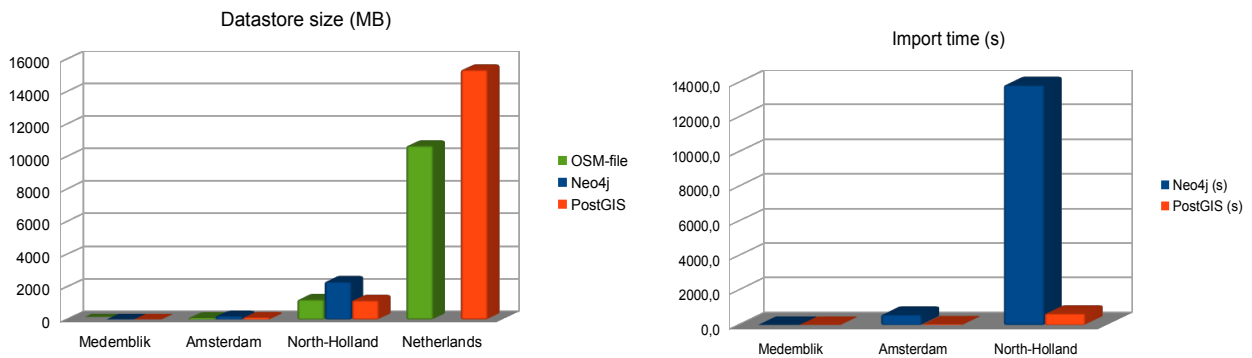


Figure 21: Measurements – importing OSM data

Paragraph 6.1.4 (page 54) will discuss in some more detail the correlation between the OSM file and the hardware needed. Because the import took so long and created an unacceptably sized database, the Netherlands dataset was not included in the other tests. When OSM file sizes remain smaller than 1 GB, both databases function acceptably as to size and speed. PostGIS has clear advantages importing larger OSM files: the import time is short and the database size is smaller.

6.1.2 Creating a routing network topology

As explained in paragraph 5.4 (page 42), a routing topology needs to be created for both databases. The result is the opposite of storing OSM data; Neo4j is faster in creating the same route network than PostGIS. Again, the differences are largest using the North-Holland dataset, and the Netherlands dataset is not included in the measurements.

As shown in Table 19, the smallest dataset from Medemblik shows an advantage for PostGIS. For larger datasets, Neo4j benefits from faster route network creating times. Figure 22 visualizes the time in seconds in a column chart.

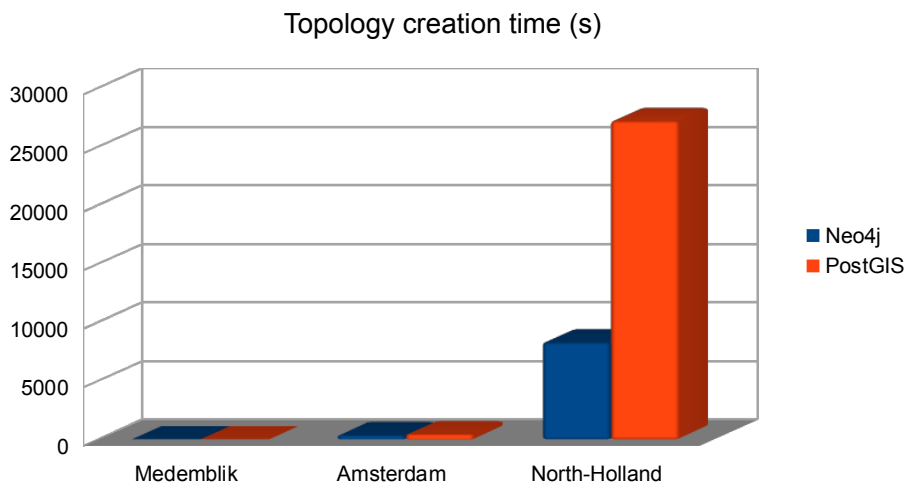


Figure 22: Measurements – creating a route topology

Apparently the graph database needs less time to find the corresponding start nodes and end nodes. The differences are found in the stage that finds the corresponding nodes. As explained in paragraph 5.4 (page 42), the pgRouting SQL procedure `assign_vertex_id()` fills in the source and target columns of the network table in such a way that connected geometries share vertex ids. This stage of the SQL script is thread-intensive and makes up 90% of processing time.

6.1.3 Spatial operations

The averaged results of the spatial operations are summarized in the table below. Raw results are presented in Appendix XVI. The following sections will discuss the results of every particular test in detail.

Table 20: Results spatial operations

Name	Empty call		Bbox count (B)		Bbox gml (G)		Closest point (C)		Shortest path(P)	
	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS
Entity	(s)	(s)	(s)	(s)	(s)	(s)	(s)	(s)	(s)	(s)
Medemblik	0,02	0,03	0,93	0,40	3,813	0,56	0,12	0,40	0,14	0,75
Amsterdam	0,02	0,03	14,75	1,01	111,075	10,70	0,53	0,51	1,28	3,54
North-Holland	0,02	0,03	57,13	2,69	701,875	41,96	2,01	0,93	5,18	29,48

Spatial bounding box operations

These operations are defined to find all the geometries within a specific rectangle area. Test B executes a count operation within a rectangle area while test G requests the data in GML format. The applied code is stated in paragraph 5.5.2, named Test B – bounding box count, and paragraph 5.5.3, named Test G – bounding box get. Undoubtedly, PostGIS outperforms the graph database with bounding box operations: the larger the dataset, the larger the differences in operation times. This test demonstrates the efficiency of the relational database. A query on a smaller dataset (Medemblik or Amsterdam) has comparable operation times to the same query on the larger North-Holland dataset.

Neo4j consistently performs slower using the same bounding boxes. However, in Neo4j, when a subsequent bounding box operation is executed after the initial run, the operation becomes much faster. For example, the first-called bounding box test in the Amsterdam area takes approximately 15 seconds. Running the same test again will take about 4 seconds. This topic will be discussed to some extent later in this paragraph.

The differences in boundary times are noteworthy. Both databases use an R-tree index for faster geometry selection. Some database optimizations will result in faster operation time, but the overall results should be roughly the same because of the R-tree index. The large differences in query time are due to the way in which Neo4j uses queries. Lucene is the index used for querying, which was originally designed as a word density search engine library. While newer versions do allow numeric queries, under the hood all data is treated as text. Executing a query on non-integer numeric data, such as doubles, results in even worse performance for Neo4j. The speed issues concerning searching for numbers in Neo4j's index are related to Lucene since conversions must be performed. This is a problem known to Neo4j's developers; an index dedicated to numeric values is supposed to result in faster boundary operations.

When boundary queries are the use case for an application, PostGIS clearly provides a more optimal solution.

Closest point operations

The following test is designed to find a node in the database which is closest to the coordinate provided, and to calculate the distance from node to coordinate. The applied code is explained in paragraph 5.5.4 and named Test C – closest node. PostGIS shows some overhead time in the Medemblik dataset; in general, the operation to find these nodes yields comparable results between the two systems.

Shortest path operations

Next are the shortest path operations based on Dijkstra (1959). The applied code is stated in paragraph

5.5.5 and named Test P – shortest path. For these queries, Neo4j was clearly faster, approximately by a factor of approximately 3, as detailed in Table 20 and Appendix XVI. A query on a smaller dataset (Medemblik or Amsterdam) has comparable operation times to the same query on the larger North-Holland dataset. Shortest path queries are the typical traversals Neo4j is optimized for, whereas relational databases are not designed to do traversals. Because graph databases maintain direct references between related data, they are most efficient when performing this type of local data analysis. Generally, relational databases use joins in order to move between tables that are linked by certain columns. Traversing a relational database must be inferred through a series of join operations on different tables. Where a subset of the data may be desired, join operations require all data in all queried tables to be examined in order to extract the desired subset. This limitation is solved by the data structure pgRouting uses. The network that used consists of one table containing all network data. The results of pgRouting are very good, but do not come near Neo4j's. This can be explained by the fact that pgRouting needs to perform lookups to the next possible node in the network, while traversing an edge in the graph database does not require lookups at all.

When traversals are the ultimate use case for a query, Neo4j clearly provides a more optimal solution.

6.1.4 Up-scaling

During the tests, three hardware parameters were monitored: CPU usage, disk I/O and RAM consumption. The main bottleneck turned out to be memory consumption. Whenever a memory shortage occurred during the tests on the graph database, performance was greatly reduced because of page swapping and high CPU rates. Neo4j manages its primitives (nodes, relationships and properties) adaptively depending on their level of use. Requested nodes or relationships will be loaded into memory and different caches exist for every primitive group. Objects that are no longer in use by the software are reclaimed by an automatic memory management system called Garbage Collection (GC). This memory management is handled by the JVM and controlled with two parameters: one controls the heap space size, while the other controls the stack space size. The heap space parameter is the most important one for Neo4j, since it governs how many objects can be allocated.

Table 21: Results primitives Neo4j

Name	OSM-file	Area	Primitives
Entity	(MB)	(km2)	(count)
Medemblik	4	6,3	198403
Amsterdam	106	100,0	5261177
North-Holland	1229	4800,0	61050667
Netherlands	10691	81000,0	553886890

When it comes to heap space the general rule is: the more primitives, the more heap space needed. If the heap space does not fit in the RAM of the computer, the heap is paged out to disk. At that moment, performance will degrade rapidly. In Appendix XVII, the number of primitives and memory allocated by the JVM is recorded.

This results in Figure 23, showing the relation between heap space and the stack space. The graph explains another cause for some measurement differences. The hardware used in the tests had 8 GB of internal memory. As seen in Figure 23, the heap space needed follows a logic curve. Based on the results, an expression can be derived to calculate heap space necessary for any OSM file size. The first element in the equation is the expected number of primitives that will be created in the database. The experiments show a ratio of approximately 50000 times the binary OSM file size. This is expressed as $p = x \cdot 50000$ where p is the number of primitives and x the OSM file size in megabytes. While the number of primitives grows with

\log_{10} , starting at 1, the amount of RAM that is required multiplies by 4 for every \log_{10} step, starting at the value of 0.0005 GB RAM. Thus, the ratio between the number of primitives p and the heap space x that is needed for the Java VM is: $\log_{10}(p) = 4^{(x-1)}$

The resulting simplified expression is: $h = \frac{4^{(\log_{10}(x \cdot 50000) - 1)}}{2000}$ where h is the heap space in GB and x is the OSM file size in MB. This expression is useful as a generic rule of thumb to calculate the Java heap space needed for any OSM input file.

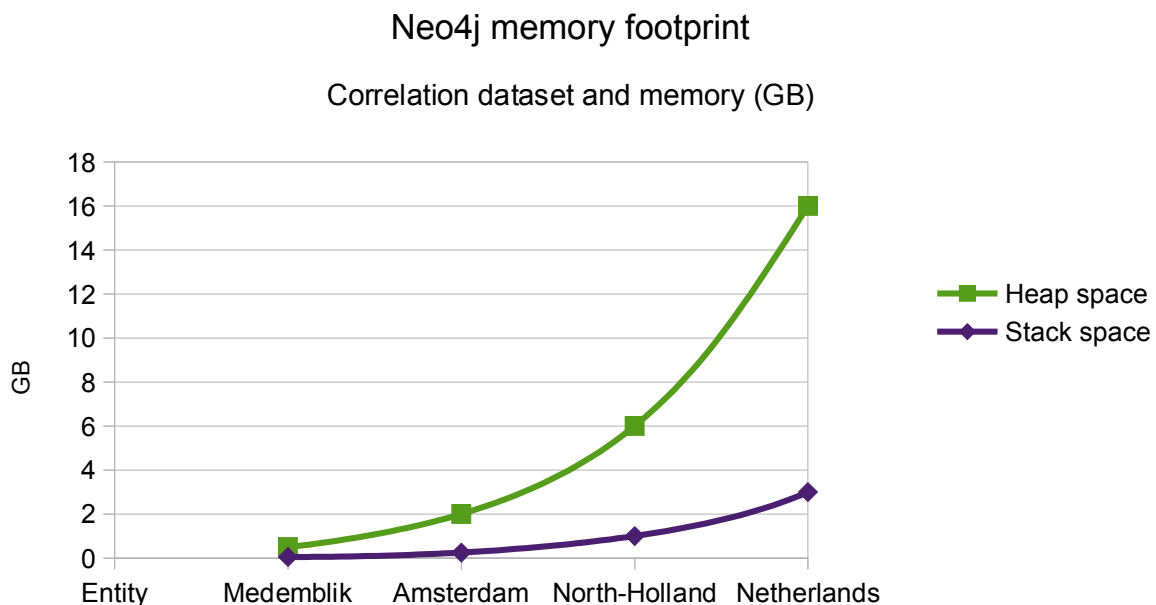


Figure 23: Neo4j memory footprint

6.2 Subjective measurements

The subjective measurements used in the comparison are: maturity and level of support, stability, and usability. While difficult to quantify, these criteria are important considerations in deciding which type of database to adopt.

6.2.1 Maturity and level of support

Maturity refers both to how old a particular system is and to how thoroughly tested it is. A higher level of testing means more time to ensure system stability, identify and solve bugs, and to have more questions answered. Obviously, a higher rate of adoption means more time to test the system. Maturity is usually proportional to level of support. A more mature piece of software will have more people using it, more people testing it in the field and more people talking about it on the web, in industry, and in academia.

The relational database, in general, is one of the most extensively used databases. The relational data model is used all over the world to support both commercial and academic pursuits and has spawned several commercial ventures. Relational databases benefit from having SQL available as a unified language through which to interact with the database. PostgreSQL and its PostGIS extension have extensive support, both from its parent company and its user community. Documentation is well-maintained and numerous guides are available to set up a spatial database quickly. This is not true for pgRouting, the shortest path extension. Both support and documentation are limited, resulting in time-consuming set-up of the extension.

Graph databases, in general, are quite new and currently less often used. They lack a unified language, which means that support for one implementation will not necessarily apply to other projects. Neo4j, specifically, is a commercial venture when used in a for-profit environment. It does have a reasonable amount of support from its parent company website. Most of its user support comes from a wiki on the Neo4j site. Support is limited from outside of the Neo4j site itself, which means that, if Neo4j ever collapses as a company, the majority support for it collapses as well. Serious effort is put in the documentation, but it is not always complete and up to date. The spatial extension documentation, in particular, needs more attention and guides are outdated. Setting up a Neo4j-Spatial can be time-consuming. While the Neo4j user community is small, it is vibrant and active, especially in comparison to that of relational databases. Community members are more than willing to help one setting up a graph database.

6.2.2 Stability

Stability refers to software running with a least as possible unexpected errors. It also refers to the way the system handles failures. Both Neo4j and PostGIS function solidly without any major issues.

During implementation and testing, numerous starts and stops were executed, as well as inserting lots of data. Whenever the development code locked up or suddenly stopped, the graph database managed the unresolved transaction. Neo4j uses two methods to manage invalid transactions. The first is a Write-Ahead Log (WAL) and the other is an implementation of a Wait-For graph that used to detect potential deadlocks before they happen. A WAL ensures that all modifications during a transaction are persisted on disk as they are requested. Modifications are performed on a temporary store, and are only performed upon the actual store after the transaction is committed to disk; subsequently, the transaction is removed from its temporary storing place. This ensures that, even if the system crashes during a commit, modifications can be recreated and replayed. The WAL is implemented by the `XaLogicalLog` class. This class manages a set of files that act as an intermediate store for all the commands and lifecycle changes of a transaction.

```
INFO: Dirty log: /Volumes/Data/Users/bartbaas/data/neo4j/amsterdam.gdb/index/lucene.log.1 now
closed. Recovery will be started automatically next time it is opened.
INFO: A valid shutdown command was received via the shutdown port. Stopping the Server instance.
...
INFO: Non clean shutdown detected on log
[/Volumes/Data/Users/bartbaas/data/neo4j/amsterdam.gdb/index/lucene.log.1]. Recovery started ...
INFO: Unresolved transactions found, recovery started ...
INFO: Recovery completed, all transactions have been resolved to a consistent state.
```

Snippet 15: Neo4j resolving a transaction

A deadlock is a situation in which two or more competing actions are waiting for each-other to finish, and thus neither ever does. This is detected with a Wait-For Graph (Mitchell and Merritt 1984) and implemented with three classes: `LockManager`, `RagManager` and `RWLock`. In a nutshell, the algorithm works like this: processes are represented as nodes and an edge indicates that one process is waiting on a resource held exclusively by another. An edge from process P_i to P_j implies P_j is holding a resource that P_i needs and thus P_i is waiting for P_j to release its lock on that resource. A deadlock exists if the graph contains any cycles. As a result, Neo4j supports transactions and maintains data integrity well. Snippet 15 shows a part of the log console while recovering an unresolved transaction. An environment with multiple user transactions has not been tested.

6.2.3 Usability

Usability is the ease of use and learnability of a technology. Neo4j may have a relatively steep learning curve compared to relational software, especially when a programmer has little or no experience programming in an object-oriented (OO) programming language.

Neo4j-Spatial is used as an embedded database in a GWT application using Java as programming language. Learning the Java language is like learning the rules of grammar for a spoken language. It's not hard to learn Java's syntax, but writing decent OO code takes some time to learn. A programmer already

experienced with an OO language will have little issues learning Java's syntax. The actual operations on spatial data is done using Neo4j's and Neo4j-Spatial's class libraries, which is a crucial learning step every new Neo4j developer has to take. Getting used to a new API applies to every Java program and is similar to learning a new SQL dialect for another database. The Neo4j-Spatial provides different ways to process spatial data. Although this can be achieved directly with the API, these operations are very limited. Another way to perform queries is to use CQL (or: ECQL, an extension that support spatial operations) an expressive language that is similar to SQL. CQL is also implemented in GeoServer and GeoTools, and is probably becoming a standard for spatial web queries. Because of the similarities with SQL, the query language is comfortable to use for every SQL migrator. The third method is using a 'Pipe', a data-flow framework developed by ThinkerPop and implemented in Neo4j-Spatial as a 'GeoPipe'. This is a new way to query data, and quite uncommon for spatial data. A GeoPipe implements a computational step that can be composed of other pipe objects to create a more complex computation. This allows, for example, starting a nearest neighbor pipe, sorting on the distance and retrieving the minimal value of the pipe as seen in Snippet 13 at page 48. Currently, pipes, in general, work for the following graph database vendors: Neo4j, OrientDB, DEX, and Sail-based RDF stores (e.g. Stardog). Java libraries easily can be extended with other Java code-based projects to provide extra functionality. This way, Neo4j-Spatial is extended for this research with a web interface from GWT to provide a dashboard for the tests. Every Java-based software can use the graph database in embedded mode to store and process spatial data. It is possible to create one Neo4j-Spatial project with different interfaces, such as a desktop GUI, an Android mobile interface and a web interface. All these options give great flexibility, but can be quite confusing for a programmer that has learn all these specific technologies to be able to create these different interfaces.

Personally, as a Java starter, it took a lot of time to get used to a new programming language and all the options available. Learning Java is a combination of learning generic computer programming and Java-specific things (Burd 2007). The language itself is straightforward, but dealing with dependencies and Neo4j API-specific calls takes a lot of time. Creating an identical SQL query for the (no less straightforward) PostGIS system therefore yielded quicker and more code-efficient results and, consequently, a significantly shorter development cycle compared to Neo4j-Spatial.

Neo4j has an easily mutable schema, while relational databases are less mutable. The relational database schema can be altered once the database is deployed, but doing so is a much more significant undertaking than with Neo4j. The OSM data model in Neo4j is not entirely schema-free; it is best described as schema-less due to the key/value store. Key/value allow the graph database to store any datatype of the programming language. Because of this, Neo4j has a large advantage compared to PostGIS storing schema-less data.

Neo4j is not partition-tolerant, meaning graph sharding is not possible. According to the CAP theorem (Brewer 2000), a distributed system can satisfy only two of the three criteria (Consistency, Availability and Partition-tolerance) at the same time. Neo4j falls under the category Consistency and Availability (CA), as does PostgreSQL. In this sense, they are comparable and neither of them has a distinct advantage.

6.3 Neo4j versus PostGIS

This paragraph aggregates the already discussed results of the implementation and the measurements in the previous sections.

As mentioned previously, the graph database is most beneficial when queries can be expressed as traversals over local regions of a graph. Queries that are well-suited to this approach are, for example, shortest path analyses or connectivity queries. This is reflected in the closest path searches: the graph database significantly outperformed the relational database. Bounding box queries were faster on the relational database while also delivering uniform results. The graph database has much different operation times and suffers from slow seeking times. The fact that the indexing mechanism used in the graph database is based on strings makes these queries less efficient. The larger the dataset, the more outspoken the differences become. Neo4j's main bottleneck is the memory it consumes. An OSM file up to one gigabyte seems to be the limit for Neo4j; importing larger datasets takes a long time and queries become slow.

Regarding the subjective measurements, Neo4j provides a lot of functionality. Transaction support is a welcome addition and the numerous ways to execute queries (for example using: Java, CQL or a geopipeline) are very nice as well. The data model is schema-less and allows additions or adjustments to the schema without any major impact on the data model. As a Java component, it lacks the overhead normally associated with server applications, yet it maintains its ability to perform well at server scale, at least to a certain limit (§6.1.4 at page 54); it is relatively easy to implement Neo4j-Spatial as an embedded component in any Java program. While describing Neo4j's data structure (§5.2.2 at page 32), a nice feature became apparent. The data model used for OSM information is able to store lineage information, the history of a dataset in the form of procedures used in the compilation. While the change-set data from OSM does not affect quality aspects, such as positional accuracy or procedure information, the possibilities of Neo4j storing lineage information might be an asset.

PostgreSQL showed its reliability as a well-known server application, operating at large-scale multi-user environments. It is efficient and extremely optimized, and by far the more mature database with a lot of documentation and support.

In picking the right system for a project the choice will depend on different aspects (Quak et al. 2008) and in some cases, Neo4j should be considered as an alternative for specific tasks. Nevertheless, relational databases will remain in use as non-relational systems are not suitable for each and every purpose. Edlich et al. (2010, pages 271–283) mention that a requirements analysis is necessary to decide on the basic database approach, relational or non-relational.

Using the results, a short Neo4j buyer's guide in the form of a flow diagram can be created for storing OSM data. This diagram is based on 1) the platform or hardware systems for the spatial database; 2) Java-based or SQL-based integration; 3) the type of spatial queries executed; and 4) how the system will scale in the future. The resulting flow diagram is displayed in Figure 24.

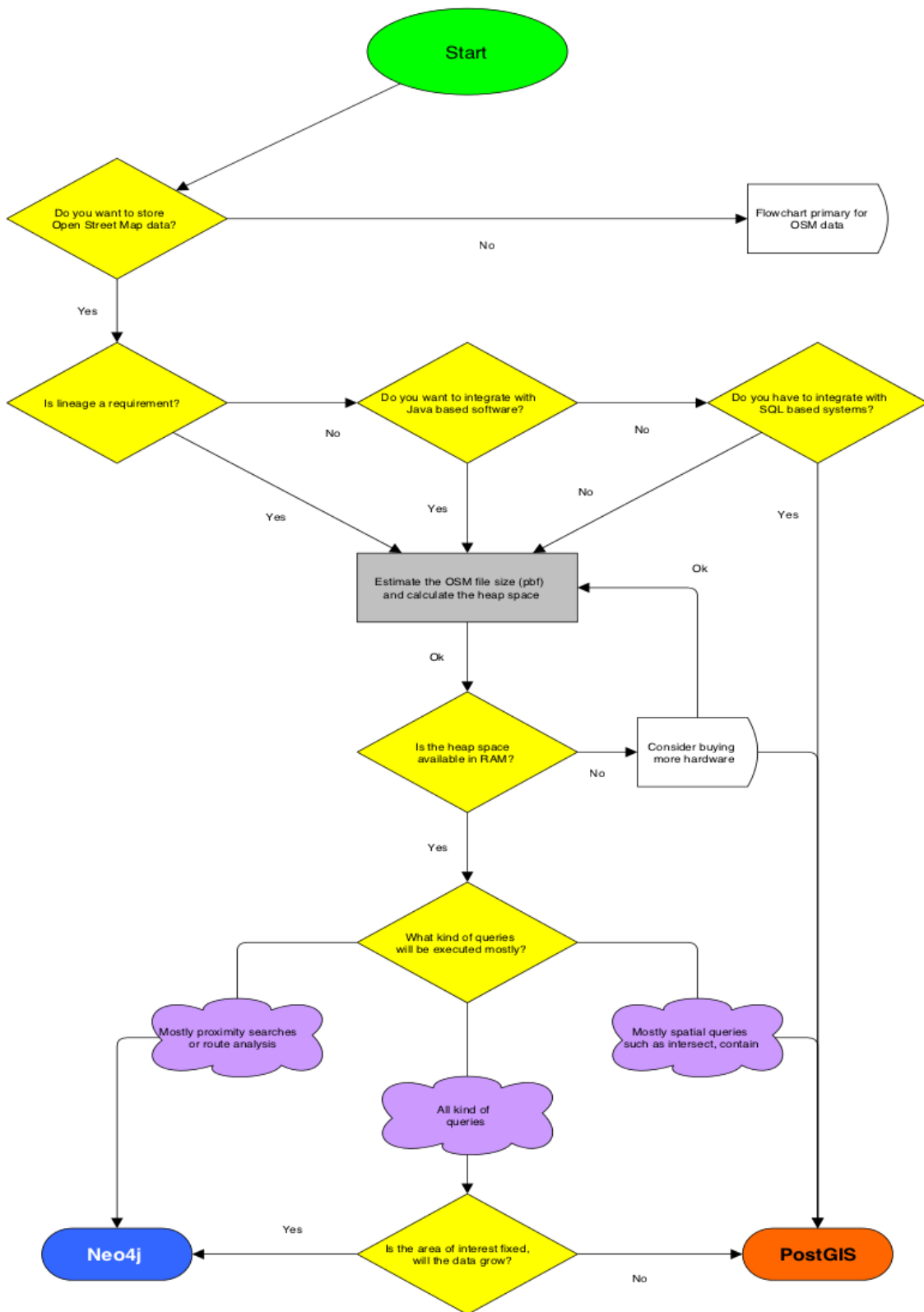


Figure 24: Neo4j versus PostGIS flow diagram

7 Conclusions

This work has presented spatial measurements on vector datasets that were subsequently used to evaluate both the functionality and performance of NoSQL systems, specifically Neo4j. The measurements include comparing the experience with the system and a range of typical spatial queries over the datasets. The experimental results provide useful guidance in database selection based on various requirements. Meanwhile, the benchmark framework could be used as a test suite for evaluating the performance and reliability of new spatial datastores.

The previous chapter gives an overview of the results that were obtained in the course of research. This chapter will return to the research questions (see §1.2.1 at page 3) with respect to the chosen NoSQL datastore, its implementation and the data applied.

7.1 Wrapping up

At the start of this research, the NoSQL movement was identified and the assumption was made that non-relational databases might have some advantages over the relational ones. This resulted in the target of this research: to discover some advantages of a NoSQL datastore as compared to a traditional relational database when storing and querying spatial vector data. The current definition of NoSQL is given, including several different categories of the NoSQL family. Next, the technical and geographical capabilities of current NoSQL systems were examined. At the time of this writing, only three NoSQL projects support spatial data: CouchDB, MongoDB and Neo4j. While all three have some promising and distinguishing features, Neo4j has the best spatial abilities, supporting the simple features specification from the Open Geospatial Consortium. Furthermore, by implementing a different approach to operate on spatial data, it could provide crucial advantages beyond the capabilities of relational systems. Thus, a competitor for PostGIS was found. This answers the first research question: *Which NoSQL project could be a good candidate to compare with PostGIS?*

An entire chapter is devoted to dive into the inner working of Neo4j as, previously, little was known of this NoSQL datastore. It is an embeddable property graph database written in Java and uses some interesting techniques to operate on OpenStreetMap data. Take, for example, the way the database identifies its primitives. Their id's are not stored, they are implemented as pointers that directly address the location of a record on disk. Also, data operations are performed differently compared to a relational database as the graph database uses traversals to answer queries. The OSM data model is built with a graph containing nodes as the vertices, and relationships as the edges of the graph. This answers the second research question: *What are the technical characteristics of the chosen NoSQL datastore?*

Regarding the third research question, *What are suitable methods to query this NoSQL datastore?:* the standard interface language of the Neo4j database is using REST requests, or is directly programmed in Java. Since not all Java API code is exposed to the REST API, the embedded version of Neo4j is used for this research. The Neo4j-Spatial API provides three ways to process spatial data. Although this can be achieved directly with the API itself, these operations are very limited. Another way to execute queries is by using CQL, an expressive language similar to SQL. The third method is a 'GeoPipe', a new way to query data and quite uncommon in the GI world. A geopipe implements a computational step that can be composed of other pipe objects to create a more complex computation.

Traversals are not suitable when one wants to find a specific node or relationship based on a property. Rather than traversing the entire graph, an index is used to perform a lookup in Neo4j. This also applies to a spatial lookup; the database uses an R-tree index structure for spatial queries. Every geometry is grouped and represented with their minimum bounding rectangle in the next higher level of the tree. The index is used only to retrieve the start elements; from that point onwards an index-free traversal is executed through the graph. In practice, while performing the benchmarks, the index seemed less efficient and demonstrated some serious performance penalties compared to PostGIS. The default Neo4j index (Lucene)

is optimized for text lookups. Numeric queries were a stumbling block for the index. This answers the fourth research question: *To what extent is an index structure supported for spatial data in the NoSQL datastore?*

Multiple aspects are of interest when choosing a spatial database system. The two databases were evaluated involving both objective and subjective measurements. A simple assessment system was proposed to evaluate the objective measurements. The subjective measurements are based on experiences during set-up and management of the two systems. A number of operations were developed for both databases, putting much effort into creating two test environments that execute identical operations and yield comparable results when applied on the OpenStreetMap data using Java for Neo4j and SQL for PostGIS. Also, each of the operations were executed for different study area sizes to determine the influence of scale: a local area (6,25 km²), an urban area (100 km²), a province area (4800 km²) and a national area (81000 km²). Initially, one OSM input file resulted in different numbers of geometries and different route networks in the databases. These problems were solved by adjusting the importing software and by creating custom tools that generate route topology. In the objective measurements, the graph database soon showed a major limitation: importing the national area resulted in an unacceptable database size and import times. While generating a route network topology, Neo4j benefits from its traversals resulting in shorter route network creating times.

By executing boundary box queries on the databases, the fifth research question was answered: *How do spatial bounding box operations perform on both databases?* When bounding box queries are the ultimate use case for an application, PostGIS provides faster queries for all study areas. The amount of heap space (Java RAM) available is trivial to Neo4j. By extension, a correlation between the different metrics was discovered and presented as a formal definition to calculate the amount of heap space (Java RAM) needed for a particular OSM file. The Dijkstra algorithm shows the speed of Neo4j-Spatial. While the relational database performed well in itself, the graph database comes in to its own when queries can be expressed as traversals over local regions of a graph. This answers the sixth research question: *How do shortest path operations perform on both databases?*

Besides objective measurements, subjective measurements were also a part of the assessment. While difficult to quantify, these criteria are significant in deciding which type of database to adopt. It is worth noting how easily the graph database integrates with other Java-based software. Every Java-based software can use the graph database in embedded mode to store and process spatial data. It is possible to create a project with different interfaces using Neo4j as storing backend. These options give great flexibility while still having atomicity and durability. This answers the seventh research question: *What are the advantages of storing spatial data in the chosen NoSQL datastore?* As a database server, PostGIS would be the better option, as it has been around for years and extensive support is provided; it is the more mature system of the two. PostGIS handles big datasets more efficiently and is better scalable.

NoSQL isn't just 'not using SQL'. It's a different storage paradigm, which comes with its own advantages and disadvantages. They will not replace relational databases, but instead will become a better option for certain types of projects. In some cases, Neo4j should be considered as an alternative for specific tasks. While relatively new in the spatial database field, it already has some advantages that are not in a relational database. Take for example the fast operations with shortest path analyses or the potentials to embed the software in other Java-based projects. Hopefully, future versions of the graph database will improve and eliminate the disadvantages it has now. Nevertheless, the two technologies, relational databases and non-relational database, will remain in usage side by side, each with the perfect fit for its own capabilities.

7.2 Future research

This section presents recommendations for further research based on the experiences from the research carried out in this research.

NoSQL is an umbrella term for a loosely defined class of non-relational datastores. The term is best described as 'Not only SQL' and grown into huge family of very different unconventional database systems. One significant problem with the NoSQL family is that there is no clear definition of the concept, no trademarks, no standard group, not even a manifesto. This raises all kinds of questions about the movement. Does any database that doesn't use SQL qualify? How about older database technologies that were used before Codd's definition? How about a relational system that does not have a SQL interface? What happens if someone manages to bolt a SQL interface onto MongoDB? To amplify the confusion, suggestions are made for NoSQL version 2.0 (slashdot.org 2012) as new systems are outperforming the current NoSQL datastores. While the technologies applied are very interesting, NoSQL is not a particularly accurate term, hampering comparison and discussion of the various products. Outlining these new technologies with a clear definition is not only necessary but also critical to further research.

The speed of a database system depends on the hardware configuration and the software algorithms used, but the efficiency of the database itself depends on software algorithms rather than raw hardware speed. The assessment approach used allows different assessment pairs with PostGIS as a reference system. The assessment pair will be configured containing the same spatial data with a client requesting identical spatial data operations, measuring the throughput and response time of every operation. It is relatively easy to extend the assessment with other types of databases, whether they are NoSQL or not. For example, extending the assessment with CouchDb would be another pair <CouchDb, PostGIS> on a different platform. It would be interesting to have other systems assessed using this framework and extend the results of this research.

This project started with an evaluation of the NoSQL movement, after which Neo4j was chosen as PostGIS competitor. The advantages of the Neo4j-Spatial graph database were presented, but not all features were evaluated. In the case of objective measurements, the tested implementations were very basic. In-depth research on the spatial abilities could give more detailed information. Interesting spatial queries are the spatial join or proximity searches.

In the process, a number of issues with Neo4j requiring special attention were discovered. The first issue was the Lucene index, originally designed as a word density search engine library. While newer versions allow numeric queries, all data is treated as text under the hood. This speed issues concerning searching for numbers in Neo4j's index are related to Lucene since conversions must be performed. A consideration for another spatial index, improving numeric queries on the index, compliments the functionality of the graph database. Fixing this issue would also improve the second stage of the import class, where the Lucene index is build. It slows down for larger datasets, affecting the scalability of the importer. The second issue is that the OSM data model could be interpreted in different ways. For example, Ways may be 'open' (when they do not share a first and last node) or 'closed' when they do. In some cases, a closed way will be interpreted as a 'closed line', and in other instances as an area, and in some cases as both a closed line and an area. In the future, it will be crucial to be able to exert control on this behavior as the importer currently provides no options to do so.

References

- Anand, G., and R. Kodali. 2008. Benchmarking the benchmarking models. *Benchmarking: An International Journal* 15 (3):257–291.
- Angles, R., and C. Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40 (1):1–39.
- apache.org. 2011. *Apache CouchDB: The Apache CouchDB Project*. <http://couchdb.apache.org/> (last accessed 19 November 2011).
- Azure Table Storage. 2011. *microsoft.com*. <http://www.microsoft.com/windowsazure/features/storage/> (last accessed 4 November 2011).
- Brewer, E. A. 2000. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 7–10.
- Browne, J. 2009. *julianbrowne.com*. *Brewer’s CAP Theorem*. <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem> (last accessed 8 June 2011).
- Bui, N. B. et al. 2007. Benchmark Generation Using Domain Specific Modeling. In *Proceedings of the 2007 Australian Software Engineering Conference*, 169–180. IEEE Computer Society.
- Burd, B. 2007. *Java for dummies*. Chichester: Wiley.
- Burrough, P., and R. McDonnell. 1998. *Principles of geographical information systems*. Oxford ;;New York: Oxford University Press.
- Chang, F. et al. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26 (2):1–26.
- Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13 (6):377–387.
- comments.gmane.org. 2011. *Osmosis development - Problems importing osm files to API bd*. <http://comments.gmane.org/gmane.comp.gis.openstreetmap.osmosis.devel/957> (last accessed 10 February 2012).
- Cook, W. R., and A. H. Ibrahim. 2006. Integrating programming languages and databases: What is the problem. *ODBMS. ORG, Expert Article*.
- couchdb.org. 2011. *CouchDB: The Definitive Guide*. <http://guide.couchdb.org/editions/1/en/index.html> (last accessed 11 October 2011).
- Dean, J., and S. Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51 (1):107–113.
- DeCandia, G. et al. 2007. Dynamo: amazon’s highly available key-value store. *ACM SIGOPS Operating Systems Review* 41 (6):205–220.
- Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1 (1):269–271.
- Dominguez-Sal, D. et al. 2011. A discussion on the design of graph database benchmarks. In *Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems*, 25–40. Singapore: Springer-Verlag.
- Edlich, S. et al. 2010. *NoSQL: Einstieg in die Welt nichtrelationaler Web-2.0-Datenbanken*. Hanser.
- Geek And Poke: NoSQL. 2011. *Geek And Poke: NoSQL*.

- <http://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html> (last accessed 8 July 2011).
- geoserver.org. 2012. *GeoServer*. <http://geoserver.org/display/GEOS/Welcome> (last accessed 10 March 2012).
- Gilbert, S., and N. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33 (2):51–59.
- gissolved.blogspot.com. 2009. *GIS Solved*. <http://gissolved.blogspot.com/2009/05/populating-mongodb-with-pois.html> (last accessed 23 May 2011).
- Gray, J. 1992. *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc.
- gremlin.tinkerpop.com. 2012. *Gremlin*. <https://github.com/tinkerpop/gremlin/wiki> (last accessed 20 May 2012).
- Güting, R. 1988. Geo-relational algebra: A model and query language for geometric database systems. *Advances in Database Technology—EDBT'88* :506–527.
- Heywood, D. et al. 2006. *An introduction to geographical information systems* 3rd ed. Harlow England ;;New York: Pearson Prentice Hall.
- Hoel, E. G., and H. Samet. 1995. Benchmarking spatial join operations with spatial output. In *Proceedings of The International Conference on Very Large Data Bases*, 606–618. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE).
- Kim, W., and F. H. Lochovsky. 1989. *Object-oriented concepts, databases, and applications*. New York: ACM Press.
- Kovacs, K. 2010. Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase comparison :: KKovacs. *KKovacs*. <http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis> (last accessed 15 August 2011).
- lucene.apache.org. 2012. <http://lucene.apache.org/> (last accessed 24 April 2012).
- Maier, D. 1990. Representing database programs as objects. In *Advances in database programming languages*, 377–386. ACM.
- maven.apache.org. 2011. *Apache Maven*. <http://maven.apache.org/> (last accessed 23 May 2012).
- McGee, W. C. 1976. On user criteria for data model evaluation. *ACM Transactions on Database Systems (TODS)* 1 (4):370–387.
- Mische. 2011. vimeo.com. *CouchDB and GeoCouch - Volker Mische on Vimeo*. <http://vimeo.com/19672716> (last accessed 11 October 2011).
- Mitchell, D. P., and M. J. Merritt. 1984. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, 282–284. ACM.
- mongodb.org. 2011. *MongoDB*. <http://www.mongodb.org/> (last accessed 19 November 2011).
- mxcl.github.com/homebrew. 2012. <http://mxcl.github.com/homebrew/> (last accessed 11 May 2012).
- Neo Technology. 2006. The Neo Database – A Technology Introduction. <http://dist.neo4j.org/neo-technology-introduction.pdf> (last accessed 18 November 2011).
- Neo4j Community - More spatial questions. 2011. *Neo4j Community Discussions - [Neo4j] More spatial questions*. <http://neo4j-community-discussions.438527.n3.nabble.com/Neo4j-More-spatial-questions-td3083509.html#a3195189> (last accessed 7 March 2012).
- Neo4j Community API. 2012. <http://components.neo4j.org/neo4j/1.5/apidocs/> (last accessed 22 February 2012).
- Neo4j Spatial Components API. 2012. <http://components.neo4j.org/neo4j-spatial/0.7/apidocs> (last accessed 21

- February 2012).
- neo4j.org. 2011. *neo4j: World's Leading Graph Database*. <http://neo4j.org/> (last accessed 20 March 2012).
- netbeans.org. 2011. *NetBeans NetBeans Platform Showcase*. <http://platform.netbeans.org/screenshots.html> (last accessed 23 May 2011).
- NOSQL meetup. 2009. <http://nosql.eventbrite.com/> (last accessed 11 July 2011).
- nosql-databases.org. 2011. *NoSQL databases*. <http://nosql-databases.org/> (last accessed 28 May 2011).
- Oracle Big Data. 2011. *wired.com*. <http://www.wired.com/wiredenterprise/2011/10/oracle-nosql-database/> (last accessed 4 November 2011).
- paolocorti.net. 2009. *paolocorti.net*. <http://www.paolocorti.net/2009/12/06/using-mongodb-to-store-geographic-data/> (last accessed 23 May 2011).
- Partovi, F. Y. 1994. Determining what to benchmark: an analytic hierarchy process approach. *International Journal of Operations & Production Management* 14 (6):25–39.
- pgrouting.org. 2012. *pgRouting Project*. <http://www.pgrouting.org/> (last accessed 22 May 2012).
- PostGIS 1.5.3 Manual. 2011. <http://postgis.refractions.net/docs/> (last accessed 21 February 2012).
- postgis.refractions.net. 2011. *PostGIS: Home*. <http://postgis.refractions.net/> (last accessed 2 April 2012).
- PostgreSQL 9.1.3 Documentation. 2011. *PostgreSQL: Documentation: Manuals: PostgreSQL 9.1.3 Documentation*. <http://www.postgresql.org/docs/9.1/static/index.html> (last accessed 27 March 2012).
- Pritchett, D. 2008. Base: An acid alternative. *Queue* 6 (3):48–55.
- Quak, W. et al. 2008. A spatial DBMS buyer's guide Inge Netterberg and Serena Coetzee (Eds.). <http://kennis.rgi.nl/?page=publications&type=publications&sub=print&id=389&col=bestand>.
- rene-pickhardt.de. 2011. *How to combine Neo4j with GWT and Eclipse*. <http://www.rene-pickhardt.de/how-to-combine-neo4j-with-gwt-and-eclipse/> (last accessed 22 February 2012).
- Rodriguez, M. A., and P. Neubauer. 2010. Constructions from Dots and Lines. *CoRR* abs/1006.2361.
- Sankar, K. 2010. Building a NoSQL Data Cloud. <http://my.safaribooksonline.com/video/databases/9781449396428> (last accessed 24 June 2011).
- Schutzberg, A. 2011. NoSQL Databases: What Geospatial Users Need to Know. *Directions magazine*. <http://www.directionsmag.com/articles/nosql-databases-what-geospatial-users-need-to-know/164635> (last accessed 23 May 2011).
- Silberschatz, A. et al. 1996. Data models. *ACM Computing Surveys (CSUR)* 28 (1):105–108.
- slashdot.org. 2012. *Is It Time For NoSQL 2.0? - Slashdot*. <http://hardware.slashdot.org/story/12/02/22/1732221/is-it-time-for-nosql-20> (last accessed 17 March 2012).
- Stonebraker, M. et al. 1993. The SEQUOIA 2000 storage benchmark. In *ACM SIGMOD Record*, 2–11. ACM.
- Strozzi, C. 1998. NoSQL Relational Database Management System: Home Page. http://www.strozzi.it/cgi-bin/CSA/tw7/l/en_US/nosql/Home%20Page (last accessed 10 July 2011).
- The Neo4j Manual v1.5. 2011. <http://docs.neo4j.org/chunked/1.5.2/index.html> (last accessed 6 November 2011).
- Theodoridis, Y. et al. 1998. Specifications for efficient indexing in spatiotemporal databases. In *Scientific and Statistical Database Management, 1998. Proceedings. Tenth International Conference on*, 123–132. IEEE.

- tinkerpop.com. 2012. *TinkerPop*. <http://tinkerpop.com/> (last accessed 22 May 2012).
- tomcat.apache.org. 2011. *Apache Tomcat*. <http://tomcat.apache.org/> (last accessed 2 March 2012).
- Vennix, J. A. M. et al. 1997. Foreword: Group model building, art, and science. *System Dynamics Review* 13 (2):103–106.
- Vicknair, C. et al. 2010. A Comparison of a Graph Database and a Relational Database. In *Proceedings of the 48th annual Southeast regional conference*.
- visualvm.java.net. 2012. *VisualVM*. <http://visualvm.java.net/> (last accessed 19 May 2012).
- vividsolutions.com/jts. 2012. *JTS Topology Suite*. <http://www.vividsolutions.com/jts/jtshome.htm> (last accessed 22 May 2012).
- Weglarz, G. 2004. Two Worlds Data-Unstructured and Structured. *DM REVIEW* 14:19–23.
- West, D. B. 2001. *Introduction to graph theory*. Prentice Hall Upper Saddle River, NJ.:
- wiki.openstreetmap.org. 2011. *OpenStreetMap Wiki*. http://wiki.openstreetmap.org/wiki/Legal_FAQ (last accessed 3 August 2011).
- wiki.openstreetmap.org/wiki/osmosis. 2011. *Osmosis*. <http://wiki.openstreetmap.org/wiki/Osmosis> (last accessed 3 August 2011).
- wikipedia.org. 2011. *Wikipedia - NoSQL*. <http://en.wikipedia.org/wiki/Nosql> (last accessed 11 July 2011).
- Zhou, Z. et al. 2009. Evaluating query performance on object-relational spatial databases. *Computer Science and Information Technology, International Conference on* 0:489–492.
- zotero.org. 2012. *Zotero*. <http://www.zotero.org/> (last accessed 27 March 2012).

Appendices

Appendix I, Brewer's (CAP) Theorem

Appendix II, Graph internals

Appendix III, Neo4j OSM data structure

Appendix IV, PostGIS OSM table structure

Appendix V, Maven pom file

Appendix VI, OSM import differences

Appendix VII, OSM equal geometries

Appendix VIII, Default.style file

Appendix IX, Neo4j StyleReader class

Appendix X, Neo4j OSMImporter (improved)

Appendix XI, Creating route topology

Appendix XII, Networks

Appendix XIII, Source code, GUI's part

Appendix XIV, Source code tests

Appendix XV, Results of the operations

Appendix XVI, Tables of measurements

Appendix XVII, Memory measurements Neo4j

Appendix I

Brewer's (CAP) Theorem

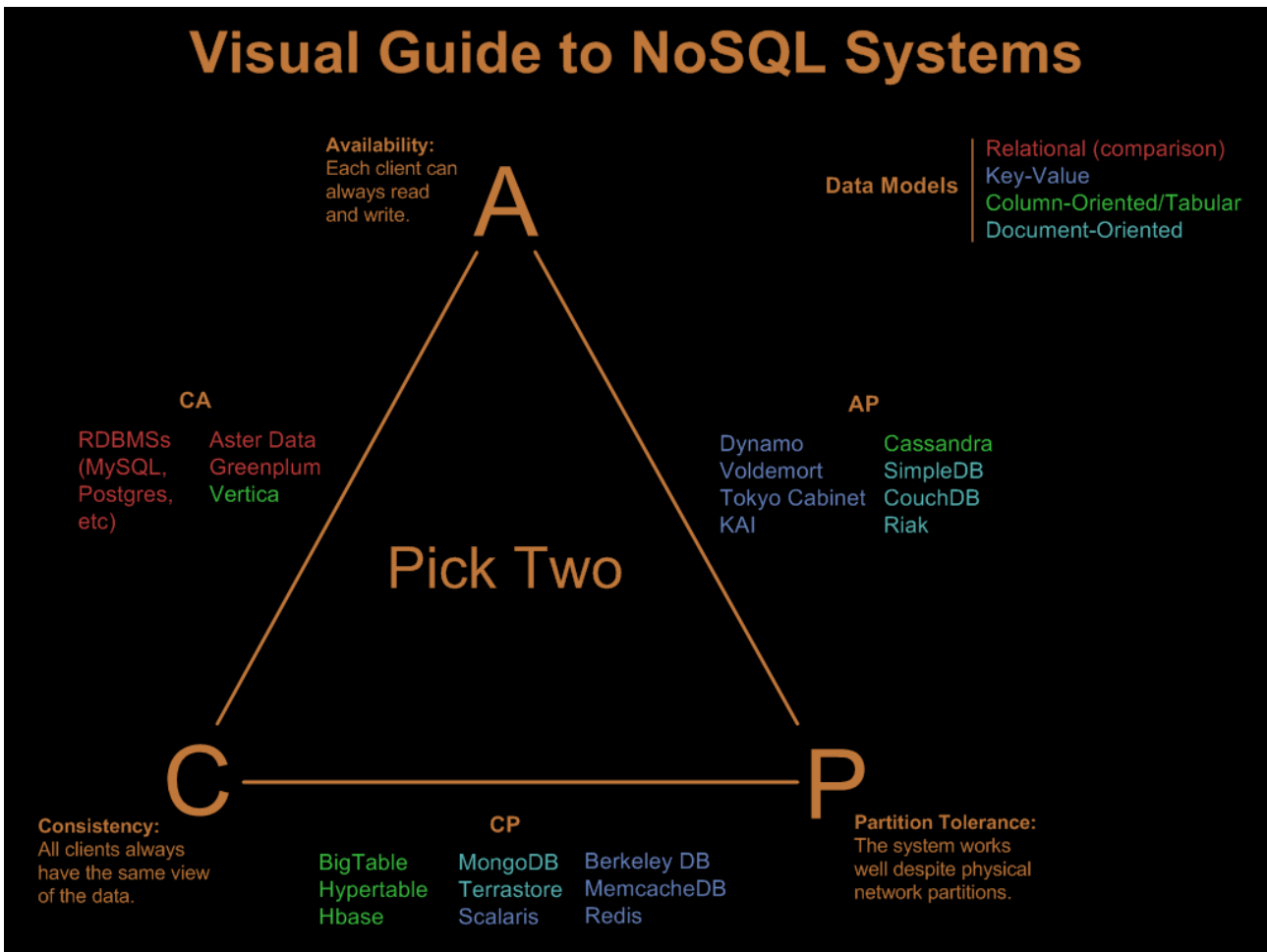
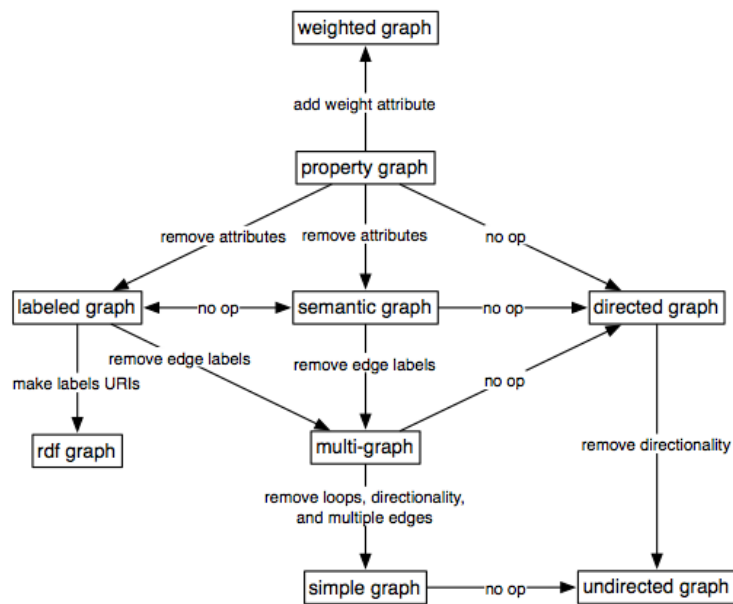


Figure adopted from <http://blog.nahurst.com/visual-guide-to-nosql-systems>

Appendix II Graph internals

Graph data models



Simple Neo4j database

```
import org.neo4j.graphdb.*;
import org.neo4j.kernel.EmbeddedGraphDatabase;

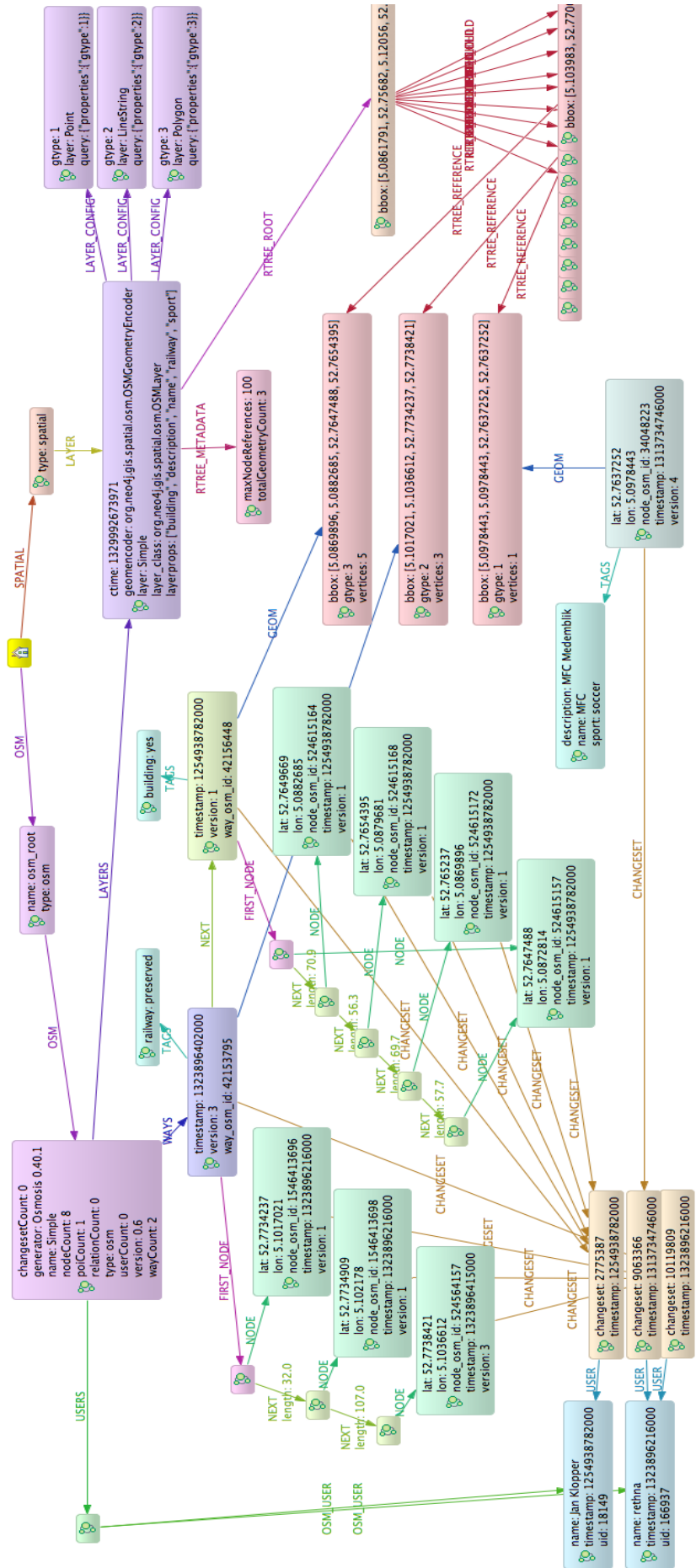
/**
 * Example class that constructs a simple graph with message attributes
 */
public class HelloNeo4j {

    public enum MyRelationshipTypes implements RelationshipType {
        KNOWS
        BLOCKS
    }

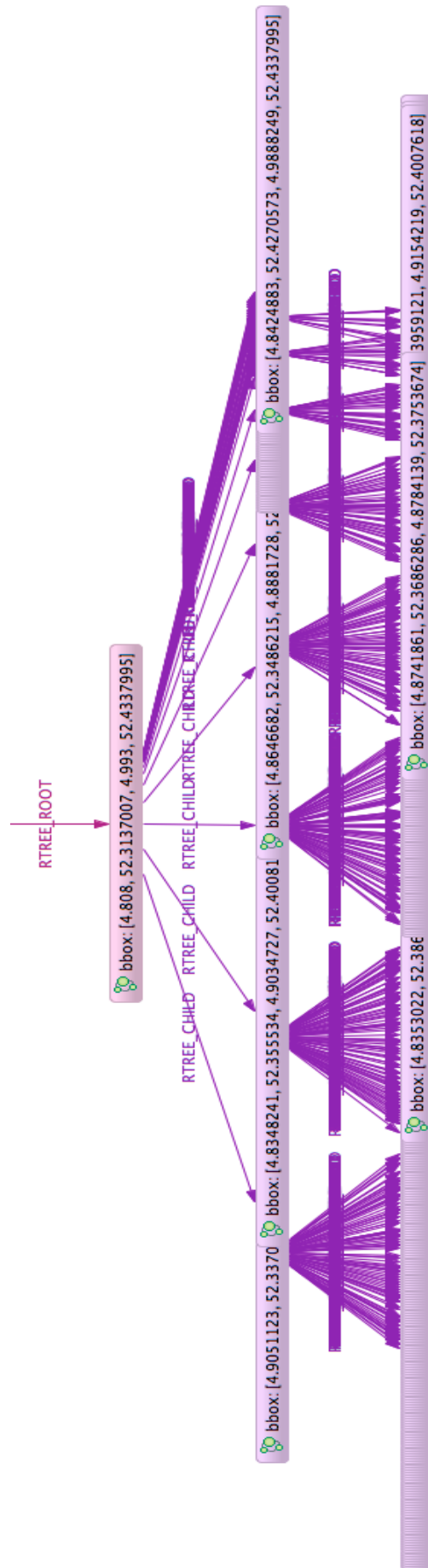
    public static void main(String[] args) {
        GraphDatabaseService graphDb = new EmbeddedGraphDatabase("var/base");
        Transaction tx = graphDb.beginTx();
        try {
            Node firstNode = graphDb.createNode();
            Node secondNode = graphDb.createNode();
            Relationship relationship =
                secondNode.createRelationshipTo(firstNode, MyRelationshipTypes.KNOWS);

            firstNode.setProperty("name", "Jan");
            secondNode.setProperty("name", "Nel");
            secondNode.setProperty("age", "62");
            relationship.setProperty("disclosure", "public");
            tx.success();
        } finally {
            tx.finish();
            graphDb.shutdown();
        }
    }
}
```

Appendix III Neo4j OSM data structure



Neo4j R-tree visualized



(Amsterdam dataset)

Appendix IV PostGIS OSM table structure

List of relations					
Schema	Name	Type	Owner	Size	Description
public	geography_columns	view	bartbaas	0 bytes	
public	geometry_columns	table	bartbaas	48 kB	
public	network	table	bartbaas	11 MB	
public	network_gid_seq	sequence	bartbaas	8192 bytes	
public	planet_osm_line	table	bartbaas	5888 kB	
public	planet_osm_line_pid_seq	sequence	bartbaas	8192 bytes	
public	planet_osm_nodes	table	bartbaas	20 MB	
public	planet_osm_point	table	bartbaas	1064 kB	
public	planet_osm_point_pid_seq	sequence	bartbaas	8192 bytes	
public	planet_osm_polygon	table	bartbaas	11 MB	
public	planet_osm_polygon_pid_seq	sequence	bartbaas	8192 bytes	
public	planet_osm_rels	table	bartbaas	8192 bytes	
public	planet_osm_roads	table	bartbaas	1112 kB	
public	planet_osm_ways	table	bartbaas	17 MB	
public	spatial_ref_sys	table	bartbaas	3000 kB	
public	vertices_tmp	table	bartbaas	2424 kB	
public	vertices_tmp_id_seq	sequence	bartbaas	8192 bytes	

Table "public.planet_osm_point"				
Column	Type	Modifiers	Storage	Description
OSM_id	integer		plain	
access	text		extended	
addr:housename	text		extended	
addr:housenumber	text		extended	
addr:interpolation	text		extended	
admin_level	text		extended	
aerialway	text		extended	
aeroway	text		extended	
amenity	text		extended	
area	text		extended	
barrier	text		extended	
bicycle	text		extended	
brand	text		extended	
bridge	text		extended	
boundary	text		extended	
building	text		extended	
capital	text		extended	
construction	text		extended	
covered	text		extended	
culvert	text		extended	
cutting	text		extended	
denomination	text		extended	
disused	text		extended	
ele	text		extended	
embankment	text		extended	
foot	text		extended	
generator:source	text		extended	
harbour	text		extended	
highway	text		extended	
historic	text		extended	
horse	text		extended	
intermittent	text		extended	
junction	text		extended	
landuse	text		extended	
layer	text		extended	
leisure	text		extended	
lock	text		extended	
man_made	text		extended	
military	text		extended	
motorcar	text		extended	
name	text		extended	
natural	text		extended	

oneway	text	extended
operator	text	extended
poi	text	extended
population	text	extended
power	text	extended
power_source	text	extended
place	text	extended
railway	text	extended
ref	text	extended
religion	text	extended
route	text	extended
service	text	extended
shop	text	extended
sport	text	extended
surface	text	extended
toll	text	extended
tourism	text	extended
tower:type	text	extended
tunnel	text	extended
water	text	extended
waterway	text	extended
wetland	text	extended
width	text	extended
wood	text	extended
z_order	integer	plain
way	geometry	main

indices:

"planet_osm_point_index" gist (way)

"planet_osm_point_pkey" B-tree (OSM_id)

Has OIDs: no

Table "public.planet_osm_line"

Column	Type	Modifiers	Storage	Description
OSM_id	integer		plain	
access	text		extended	
addr:housename	text		extended	
addr:housenumber	text		extended	
addr:interpolation	text		extended	
admin_level	text		extended	
aerialway	text		extended	
aeroway	text		extended	
amenity	text		extended	
area	text		extended	
barrier	text		extended	
bicycle	text		extended	
brand	text		extended	
bridge	text		extended	
boundary	text		extended	
building	text		extended	
construction	text		extended	
covered	text		extended	
culvert	text		extended	
cutting	text		extended	
denomination	text		extended	
disused	text		extended	
embankment	text		extended	
foot	text		extended	
generator:source	text		extended	
harbour	text		extended	
highway	text		extended	
historic	text		extended	
horse	text		extended	
intermittent	text		extended	
junction	text		extended	
landuse	text		extended	
layer	text		extended	
leisure	text		extended	
lock	text		extended	
man_made	text		extended	
military	text		extended	

motorcar	text		extended
name	text		extended
natural	text		extended
oneway	text		extended
operator	text		extended
population	text		extended
power	text		extended
power_source	text		extended
place	text		extended
railway	text		extended
ref	text		extended
religion	text		extended
route	text		extended
service	text		extended
shop	text		extended
sport	text		extended
surface	text		extended
toll	text		extended
tourism	text		extended
tower:type	text		extended
tracktype	text		extended
tunnel	text		extended
water	text		extended
waterway	text		extended
wetland	text		extended
width	text		extended
wood	text		extended
z_order	integer		plain
way_area	real		plain
way	geometry		main

indices:

"planet_osm_line_index" gist (way)
"planet_osm_line_pkey" B-tree (OSM_id)

Has OIDs: no

Table "public.planet_osm_polygon"

Column	Type	Modifiers	Storage	Description
OSM_id	integer		plain	
access	text		extended	
addr:housename	text		extended	
addr:housenumber	text		extended	
addr:interpolation	text		extended	
admin_level	text		extended	
aerialway	text		extended	
aeroway	text		extended	
amenity	text		extended	
area	text		extended	
barrier	text		extended	
bicycle	text		extended	
brand	text		extended	
bridge	text		extended	
boundary	text		extended	
building	text		extended	
construction	text		extended	
covered	text		extended	
culvert	text		extended	
cutting	text		extended	
denomination	text		extended	
disused	text		extended	
embankment	text		extended	
foot	text		extended	
generator:source	text		extended	
harbour	text		extended	
highway	text		extended	
historic	text		extended	
horse	text		extended	
intermittent	text		extended	
junction	text		extended	
landuse	text		extended	
layer	text		extended	

leisure	text	extended
lock	text	extended
man_made	text	extended
military	text	extended
motorcar	text	extended
name	text	extended
natural	text	extended
oneway	text	extended
operator	text	extended
population	text	extended
power	text	extended
power_source	text	extended
place	text	extended
railway	text	extended
ref	text	extended
religion	text	extended
route	text	extended
service	text	extended
shop	text	extended
sport	text	extended
surface	text	extended
toll	text	extended
tourism	text	extended
tower:type	text	extended
tracktype	text	extended
tunnel	text	extended
water	text	extended
waterway	text	extended
wetland	text	extended
width	text	extended
wood	text	extended
z_order	integer	plain
way_area	real	plain
way	geometry	main

indices:

```
"planet_osm_polygon_index" gist (way)
"planet_osm_polygon_no_building_index" gist (way) WHERE building IS NULL
"planet_osm_polygon_pkey" B-tree (OSM_id)
```

Has OIDs: no

Table "public.network"

Column	Type	Modifiers	Storage	Description
gid	integer	not null default nextval('network_gid_seq'::regclass)	plain	
OSM_id	integer		plain	
name	character varying		extended	
the_geom	geometry		main	
source	integer		plain	
target	integer		plain	
length	double precision		plain	

Has OIDs: no

Table "public.vertices_tmp"

Column	Type	Modifiers	Storage	Description
id	integer	not null default nextval('vertices_tmp_id_seq'::regclass)	plain	
the_geom	geometry		main	

indices:

```
"vertices_tmp_idx" gist (the_geom)
```

Check constraints:

```
"enforce_dims_the_geom" CHECK (st_ndims(the_geom) = 2)
"enforce_geotype_the_geom" CHECK (geometrytype(the_geom) = 'POINT'::text OR the_geom IS NULL)
"enforce_srid_the_geom" CHECK (st_srid(the_geom) = 4326)
```

Has OIDs: no

Appendix V Maven pom file

Maven pom file as explained in paragraph 5.1.1, page 29.

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  <!-- POM file generated with GWT webAppCreator -->
  <modelVersion>4.0.0</modelVersion>
  <groupId>gima.neo4j.testsuite</groupId>
  <artifactId>neo4jspatial</artifactId>
  <packaging>war</packaging>
  <version>0.2-SNAPSHOT</version>
  <name>Neo4j GWT TestSuite</name>
  <description>Testing environment for Neo4j-Spatial</description>

  <properties>
  <!-- Convenience property to set the GWT version -->
    <gwtVersion>2.4.0</gwtVersion>
  <!-- Neo4j versions -->
    <!--neo4j.version>1.4</neo4j.version-->
    <neo4j.version>1.5</neo4j.version>
    <neo4j.graphcollections.version>1.5-SNAPSHOT</neo4j.graphcollections.version>
    <neo4j.spatial.version>0.7-SNAPSHOT</neo4j.spatial.version>
    <!--geotools.version>2.7-RC1</geotools.version-->
    <geotools.version>8.0-M2</geotools.version>
    <gremlin.version>1.4</gremlin.version>
    <blueprints.version>1.1</blueprints.version>
  <!-- GWT needs at least java 1.5 -->
    <webappDirectory>${project.build.directory}/${project.build.finalName}</webappDirectory>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <netbeans.hint.deploy.server>gfv3ee6</netbeans.hint.deploy.server>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-servlet</artifactId>
      <version>${gwtVersion}</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.google.gwt</groupId>
      <artifactId>gwt-user</artifactId>
      <version>${gwtVersion}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>javax.validation</groupId>
      <artifactId>validation-api</artifactId>
      <version>1.0.0.GA</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.validation</groupId>
      <artifactId>validation-api</artifactId>
      <version>1.0.0.GA</version>
      <classifier>sources</classifier>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j</artifactId>
      <version>${neo4j.version}</version>
      <exclusions>
        <exclusion>
          <groupId>org.neo4j</groupId>
          <artifactId>neo4j-udc</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>org.neo4j</groupId>
```

```

        <artifactId>neo4j-graph-collections</artifactId>
        <version>${neo4j.graphcollections.version}</version>
    </dependency>
    <dependency>
        <groupId>org.neo4j</groupId>
        <artifactId>neo4j-graphviz</artifactId>
        <version>${neo4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.neo4j</groupId>
        <artifactId>Neo4j-Spatial</artifactId>
        <version>${neo4j.spatial.version}</version>
        <!--scope>system</scope>
        <systemPath>${basedir}/src/main/resources/Neo4j-Spatial/Neo4j-Spatial-0.7-SNAPSHOT.jar</systemPath-->
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-main</artifactId>
        <version>${geotools.version}</version>
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-shapefile</artifactId>
        <version>${geotools.version}</version>
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-process</artifactId>
        <version>${geotools.version}</version>
    </dependency>
    <dependency>
        <groupId>org.geotools</groupId>
        <artifactId>gt-render</artifactId>
        <version>${geotools.version}</version>
        <exclusions>
            <exclusion>
                <groupId>it.geosolutions.imageio-ext</groupId>
                <artifactId>imageio-ext-tiff</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>com.tinkerpop.gremlin</groupId>
        <artifactId>gremlin-groovy</artifactId>
        <version>${gremlin.version}</version>
        <type>jar</type>
        <exclusions>
            <!-- Sail support not needed -->
            <exclusion>
                <groupId>com.tinkerpop.blueprints</groupId>
                <artifactId>blueprints-sail-graph</artifactId>
            </exclusion>
            <!-- Maven support in groovy not needed -->
            <exclusion>
                <groupId>org.codehaus.groovy.maven</groupId>
                <artifactId>gmaven-plugin</artifactId>
            </exclusion>
            <!-- "readline" not needed - we only expose gremlin through webadmin -->
            <exclusion>
                <groupId>jline</groupId>
                <artifactId>jline</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>com.tinkerpop.gremlin</groupId>
        <artifactId>gremlin-java</artifactId>
        <version>${gremlin.version}</version>
        <type>jar</type>
        <exclusions>
            <!-- Sail support not needed -->
            <exclusion>
                <groupId>com.tinkerpop.blueprints</groupId>
                <artifactId>blueprints-sail-graph</artifactId>
            </exclusion>
            <!-- Maven support in groovy not needed -->
            <exclusion>
                <groupId>org.codehaus.groovy.maven</groupId>
                <artifactId>gmaven-plugin</artifactId>
            </exclusion>
            <!-- "readline" not needed - we only expose gremlin through webadmin -->
            <exclusion>
                <groupId>jline</groupId>
                <artifactId>jline</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

```

```

</exclusions>
</dependency>
<dependency>
  <groupId>com.tinkerpop.blueprints</groupId>
  <artifactId>blueprints-neo4j-graph</artifactId>
  <version>${blueprints.version}</version>
  <exclusions>
    <exclusion>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
</dependency>
<dependency>
  <groupId>commons-collections</groupId>
  <artifactId>commons-collections</artifactId>
  <version>3.2</version>
  <type>jar</type>
</dependency>
</dependencies>
<repositories>
  <repository>
    <id>osgeo</id>
    <name>Open Source Geospatial Foundation Repository</name>
    <url>http://download.osgeo.org/webdav/geotools/</url>
  </repository>
  <repository>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
    <id>opengeo</id>
    <name>OpenGeo Maven Repository</name>
    <url>http://repo.opengeo.org</url>
  </repository>
  <repository>
    <id>neo4j-public-repository</id>
    <name>Publically available Maven 2 repository for Neo4j</name>
    <url>http://m2.neo4j.org</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>tinkerpop-snapshot-repository</id>
    <name>Tinkerpop snapshot repo</name>
    <url>http://tinkerpop.com/maven2</url>
    <snapshots>
      <enabled>>true</enabled>
    </snapshots>
    <releases>
      <enabled>>true</enabled>
    </releases>
  </repository>
</repositories>

<build>
<!-- Generate compiled stuff in the folder used for developing mode -->
  <outputDirectory>${webappDirectory}/WEB-INF/classes</outputDirectory>

  <plugins>
    <!-- GWT Maven Plugin -->
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>gwt-maven-plugin</artifactId>
      <version>2.4.0</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>test</goal>
            <goal>generateAsync</goal>
          </goals>
        </execution>
      </executions>
    <!-- Plugin configuration. There are many available options, see
    gwt-maven-plugin documentation at codehaus.org -->
    <configuration>
      <runTarget>tests.html</runTarget>
      <hostedWebapp>${webappDirectory}</hostedWebapp>
    </configuration>
  </plugins>
</build>

```

```
    </plugin>
  <!-- Copy static web files before executing gwt:run -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.1.1</version>
    <executions>
      <execution>
        <phase>compile</phase>
        <goals>
          <goal>exploded</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <webappDirectory>${webappDirectory}</webappDirectory>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.3.2</version>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins>
</build>
</project>
```

Appendix VI OSM import differences

Initial import Medemblik

Points



Lines



Polygons



Appendix VII OSM equal geometries

Import after correction Medemblik

Points



Lines



Polygons



Results in a table:

Medemblik								
	Initial		Fix 1		Fix 2		Final (fix 1 + 2)	
Primitive	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j
Points	156	156	156	156	156	156	156	156
Lines	1000	2107	854	2103	1000	857	854	854
Polygons	1630	374	1617	369	1630	1622	1617	1617

Amsterdam								
	Initial		Fix 1		Fix 2		Final (fix 1 + 2)	
Primitive	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j
Points	12483	12473	12483	12721	12483	12473	12483	12483
Lines	36386	42497	34858	42315	36386	34903	34858	34858
Polygons	39737	32221	39627	32155	39737	39696	39627	39627

North-Holland								
	Initial		Fix 1		Fix 2		Final (fix 1 + 2)	
Primitive	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j	PostGIS	Neo4j
Points	39503	39226	39503	39459	39503	39226	39503	39503
Lines	200045	261635	192731	261497	200045	214984	192731	192731
Polygons	508039	433920	502646	433825	508039	535043	502646	502646

Appendix VIII Default.style file

```
# This is the style file that matches the old version of osm2pgsql, which
# did not make distinctions between tags for nodes and for ways. There are a
# number of optimisations that can be applied here. Firstly, certain tags
# only apply to only nodes or only ways. By fixing this we reduce the amount
# of useless data loaded into the DB, which is a good thing. Possible
# optimisations for the future:

# 1. Generate this file directly from the mapnik XML config, so it's always
# optimal

# 2. Extend it so it can understand that highway=tertiary is for ways and
# highway=bus_stop is for nodes

# Flags field isn't used much yet, expect if it contains the text "polygon"
# it indicates the shape is candidate for the polygon table. In the future I
# would like to be able to add directives like "nocache" which tells
# osm2pgsql that it is unlikely this node will be used by a way and so it
# doesn't need to be stored (eg coastline nodes). While in essence an
# optimisation hack, for --slim mode it doesn't matter if you're wrong, but
# in non-slim you might break something!

# Also possibly an ignore flag, for things like "note" and "source" which
# can simply be deleted. (In slim mode this is, does not apply to non-slim
# obviously)

# OsmType Tag          DataType  Flags
node,way  note             text      delete # These tags can be long but are useless for rendering
node,way  source           text      delete # This indicates that we shouldn't store them
node,way  created_by       text      delete

node,way  access           text      linear
node,way  addr:house       text      linear
node,way  addr:house       text      linear
node,way  addr:interpolat text      linear
node,way  admin_level      text      linear
node,way  aerialway        text      linear
node,way  aeroway          text      polygon
node,way  amenity          text      nocache,polygon
node,way  area             text      # hard coded support for area=1/yes => polygon is in osm2pgsql
node,way  barrier          text      linear
node,way  bicycle          text      nocache
node,way  brand            text      linear
node,way  bridge           text      linear
node,way  boundary         text      linear
node,way  building         text      polygon
node      capital      text      linear
node,way  construction     text      linear
node,way  covered          text      linear
node,way  culvert          text      linear
node,way  cutting          text      linear
node,way  denomination     text      linear
node,way  disused          text      linear
node      ele         text      linear
node,way  embankment       text      linear
node,way  foot             text      linear
node,way  generator:source text      linear
node,way  harbour          text      polygon
node,way  highway          text      linear
node,way  historic         text      polygon
node,way  horse           text      linear
node,way  intermittent     text      linear
node,way  junction         text      linear
node,way  landuse         text      polygon
node,way  layer           text      linear
node,way  leisure         text      polygon
node,way  lock            text      linear
node,way  man_made        text      polygon
node,way  military         text      polygon
node,way  motorcar        text      linear
node,way  name            text      linear
node,way  natural         text      polygon # natural=coastline tags are discarded by a hard coded rule
node,way  oneway          text      linear
node,way  operator        text      linear
node      poi         text      linear
node,way  population       text      linear
node,way  power           text      polygon
node,way  power_source    text      linear
node,way  place           text      polygon
node,way  railway         text      linear
node,way  ref            text      linear
node,way  religion        text      nocache
node,way  route          text      linear
node,way  service         text      linear
```

```

node,way shop text polygon
node,way sport text polygon
node,way surface text linear
node,way toll text linear
node,way tourism text polygon
node,way tower:type text linear
way tracktype text linear
node,way tunnel text linear
node,way water text polygon
node,way waterway text polygon
node,way wetland text polygon
node,way width text linear
node,way wood text linear
node,way z_order int4 linear # This is calculated during import
way way_area real # This is calculated during import

# If you're interested in bicycle routes, you may want the following fields
# To make these work you need slim mode or the necessary data won't be remembered.
#way lcn_ref text linear
#way rcn_ref text linear
#way ncn_ref text linear
#way lcn text linear
#way rcn text linear
#way ncn text linear
#way lwn_ref text linear
#way rwn_ref text linear
#way nwn_ref text linear
#way lwn text linear
#way rwn text linear
#way nwn text linear
#way route_pref_color text linear
#way route_name text linear

# The following entries can be used with the --extra-attributes option
# to include the username, userid, version & timestamp in the DB
#node,way OSM_user text
#node,way OSM_uid text
#node,way OSM_version text
#node,way OSM_timestamp text

```

Appendix IX Neo4j StyleReader class

```
package gima.neo4j.testsuite.osmcheck;

public class StyleReader {
    public static final String DEFAULT = "/usr/local/share/osm2pgsql/default.style";
    public static final String LINE = "linear";
    public static final String POLYGON = "polygon";

    public static ArrayList<String> readWays() {
        String line = ""; ArrayList<String> data = new ArrayList<String>();
        try {
            FileReader fr = new FileReader(DEFAULT);
            BufferedReader br = new BufferedReader(fr);
            while ((line = br.readLine()) != null) {
                if (!line.startsWith("#")) {
                    String[] theline = split(line, " ", true);
                    if (theline.length > 3) {
                        if (theline[0].contains("way") & !theline[3].contains("delete")) {
                            data.add(theline[1]);
                        }
                    }
                }
            }
        } catch (IOException e) { System.out.println(e); }
        return data;
    }

    public static ArrayList<String> readNodes() {
        String line = ""; ArrayList<String> data = new ArrayList<String>();
        try {
            FileReader fr = new FileReader(DEFAULT);
            BufferedReader br = new BufferedReader(fr);
            while ((line = br.readLine()) != null) {
                if (!line.startsWith("#")) {
                    String[] theline = split(line, " ", true);
                    if (theline.length > 3) {
                        if (theline[0].contains("node") & !theline[3].contains("delete")) {
                            data.add(theline[1]);
                        }
                    }
                }
            }
        } catch (IOException e) { System.out.println(e); }
        return data;
    }

    public static ArrayList<String> readPolyCandidates() {
        String line = ""; ArrayList<String> data = new ArrayList<String>();
        try {
            FileReader fr = new FileReader(DEFAULT);
            BufferedReader br = new BufferedReader(fr);
            while ((line = br.readLine()) != null) {
                if (!line.startsWith("#")) {
                    String[] theline = split(line, " ", true);
                    if (theline.length > 3) {
                        if (theline[0].contains("way") & theline[3].contains(POLYGON)) {
                            data.add(theline[1]);
                        }
                    }
                }
            }
        } catch (IOException e) { System.out.println(e); }
        return data;
    }

    private static String[] split(String str, String delimiter, boolean removeEmpty) {
        final int len = (str == null) ? 0 : str.length();
        if (len == 0) { return new String[0]; }
        final List<String> result = new ArrayList<String>();
        String elem = null;
        int i = 0, j = 0;
        while (j != -1 && j < len) {
            j = str.indexOf(delimiter, i);
            elem = (j != -1) ? str.substring(i, j) : str.substring(i);
            i = j + 1;
            if (!removeEmpty || !(elem == null || elem.length() == 0)) {
                result.add(elem);
            }
        }
        return result.toArray(new String[result.size()]);
    }
}
```

Appendix X Neo4j OSMImporter (improved)

Relevant modified parts

```
ArrayList<String> pointTags = StyleReader.readNodes();
ArrayList<String> lineTags = StyleReader.readWays();
ArrayList<String> polyTags = StyleReader.readPolyCandidates();

private void addOSMNodeTags(boolean allPoints, LinkedHashMap<String, Object> currentNodeTags) {
    currentNodeTags.remove("created_by");

    if (inList(pointTags, currentNodeTags)) {
        if (allPoints || currentNodeTags.size() > 0) {
            Map<String, Object> nodeProps = getNodeProperties(currentNode);
            Envelope bbox = new Envelope();
            double[] location = new double[] {(Double) nodeProps.get("lon"), (Double) nodeProps.get("lat")};
            bbox.expandToInclude(location[0], location[1]);
            addNodeGeometry(currentNode, GTYPE_POINT, bbox, 1);
            poiCount++;
        }
    }
    addNodeTags(currentNode, currentNodeTags, "node");
}

private boolean inList(ArrayList<String> list, LinkedHashMap<String, Object> currentNodeTags) {
    Iterator it = list.iterator();
    while (it.hasNext()) {
        String tag = (String) it.next();
        if (currentNodeTags.containsKey(tag)) {
            return true;
        }
    }
    return false;
}
```

```

protected void createOSMWay(Map<String, Object> wayProperties, ArrayList<Long> wayNodes, LinkedHashMap<String,
Object> wayTags) {
    RoadDirection direction = isOneway(wayTags);
    String name = (String) wayTags.get("name");
    int geometry = GTYPE_LINESTRING;
    boolean isRoad = wayTags.containsKey("highway");
    boolean nomineeLine = inList(lineTags, wayTags);
    boolean nomineePoly = inList(polyTags, wayTags);
    boolean hasAreaTag = false;
    if (wayTags.containsKey("area")) {
        hasAreaTag = wayTags.get("area").equals("yes");
    }

    if (!nomineeLine) { return; }

    if (isRoad) {
        wayProperties.put("oneway", direction.toString());
        wayProperties.put("highway", wayTags.get("highway"));
    }
    if (name != null) {
        wayProperties.put("name", name);
    }

    T changesetNode = getChangesetNode(wayProperties);
    T way = addNode(INDEX_NAME_WAY, wayProperties, "way_osm_id");
    createRelationship(way, changesetNode, OSMRelation.CHANGESET);
    if (prev_way == null) {
        createRelationship(OSM_dataset, way, OSMRelation.WAYS);
    } else {
        createRelationship(prev_way, way, OSMRelation.NEXT);
    }
    prev_way = way;
    addNodeTags(way, wayTags, "way");
    Envelope bbox = new Envelope();
    T firstNode = null;
    T prevNode = null;
    T prevProxy = null;
    Map<String, Object> prevProps = null;
    LinkedHashMap<String, Object> relProps = new LinkedHashMap<String, Object>();
    HashMap<String, Object> directionProps = new HashMap<String, Object>();
    directionProps.put("oneway", true);
    for (long nd_ref : wayNodes) {
        T pointNode = getOSMNode(nd_ref, changesetNode);
        if (pointNode == null) {
            missingNode(nd_ref);
            continue;
        }
        T proxyNode = createProxyNode();
        if (firstNode == null) {
            firstNode = pointNode;
        }
        if (prevNode == pointNode) {
            continue;
        }
        createRelationship(proxyNode, pointNode, OSMRelation.NODE, null);
        Map<String, Object> nodeProps = getNodeProperties(pointNode);
        double[] location = new double[]{(Double) nodeProps.get("lon"), (Double) nodeProps.get("lat")};
        bbox.expandToInclude(location[0], location[1]);
        if (prevProxy == null) {
            createRelationship(way, proxyNode, OSMRelation.FIRST_NODE);
        } else {
            relProps.clear();
            double[] prevLoc = new double[]{(Double) prevProps.get("lon"), (Double) prevProps.get("lat")};
            double length = distance(prevLoc[0], prevLoc[1], location[0], location[1]);
            relProps.put("length", length);
            createRelationship(prevProxy, proxyNode, OSMRelation.NEXT, relProps);
        }
        prevNode = pointNode;
        prevProxy = proxyNode;
        prevProps = nodeProps;
    }
    if (prevNode.equals(firstNode) & wayNodes.size() >= 4) {
        if (nomineePoly || hasAreaTag) {
            geometry = GTYPE_POLYGON;
        }
    }
    if (prevNode.equals(firstNode) & wayNodes.size() == 2) {
        return;
    }
    if (wayNodes.size() < 2) {
        return;
    }
    addNodeGeometry(way, geometry, bbox, wayNodes.size());
    this.wayCount++;
}

```

Appendix XI Creating route topology

Creating network topology in Java

Main class initiating the iteration

```
public void MakeTopology() {
    NetworkGenerator networkGenerator = null;
    Transaction tx = graphService.beginTx();
    try {
        List<SpatialDatabaseRecord> list = OSMGeoPipeline
            .startOsm(osmLayer())
            .cqlFilter("highway is not null
                and highway not in ('cycleway','footway','pedestrian','service')
                and the_geom IS NOT NULL and geometryType(the_geom) = 'LineString'")
            .toSpatialDatabaseRecordList();

        int listCount = list.size();
        results = results + "<br>Found highway linestrings: " + listCount;

        EditableLayer netPointsLayer = spatialService
            .getOrCreateEditableLayer(osmLayer().getName() + " - network points");
        netPointsLayer.setCoordinateReferenceSystem(osmLayer()
            .getCoordinateReferenceSystem());

        EditableLayer netEdgesLayer = spatialService
            .getOrCreateEditableLayer(osmLayer().getName() + " - network edges");
        netEdgesLayer.setCoordinateReferenceSystem(osmLayer().getCoordinateReferenceSystem());

        networkGenerator = new NetworkGenerator(netPointsLayer, netEdgesLayer, 0.002);
        tx.success();
        tx.finish();

        Iterator<SpatialDatabaseRecord> it = list.iterator();
        while (it.hasNext()) {
            tx = graphService.beginTx();
            try {
                int worked = 0;
                for (int i = 0; i < 5000 && it.hasNext(); i++) {
                    networkGenerator.add(it.next());
                    worked++;
                }

                tx.success();
            } finally {
                tx.finish();
            }
        }
    } catch (Exception ex) {
        return (ex.toString());
    } finally {
    }
}
```

NetworkGenerator class

```
package gima.neo4j.testsuite.server;
import ...

public class NetworkGenerator {
    private EditableLayer pointsLayer; private EditableLayer edgesLayer;
    private Double buffer; private int edgePointCounter;

    public NetworkGenerator(EditableLayer pointsLayer, EditableLayer edgesLayer) {
        this(pointsLayer, edgesLayer, null);
    }

    public NetworkGenerator(EditableLayer pointsLayer, EditableLayer edgesLayer, Double buffer) {
        this.pointsLayer = pointsLayer;
        this.edgesLayer = edgesLayer;
        this.buffer = buffer;
    }

    public int edgePointCounter() {
        return edgePointCounter;
    }

    public void add(SpatialDatabaseRecord record) {
        Geometry geometry = record.getGeometry();
        if (geometry instanceof MultiLineString) {
            add((MultiLineString) geometry, null);
        } else if (geometry instanceof LineString) {
            add((LineString) geometry, null);
        } else {
            throw new IllegalArgumentException();
        }
    }

    protected void add(MultiLineString line, SpatialDatabaseRecord record) {
        for (int i = 0; i < line.getNumGeometries(); i++) {
            add((LineString) line.getGeometryN(i), record);
        }
    }

    protected void add(LineString line, SpatialDatabaseRecord edge) {
        if (edge == null) {
            edge = edgesLayer.add(line);
        }
        edge.setProperty("_distance", distance(line));
        addEdgePoint(edge.getGeomNode(), line.getStartPoint());
        addEdgePoint(edge.getGeomNode(), line.getEndPoint());
    }

    protected void addEdgePoint(Node node, Geometry edgePoint) {
        Iterator<SpatialDatabaseRecord> results = GeoPipeline
            .startNearestNeighborLatLonSearch(pointsLayer, edgePoint.getCoordinate(), buffer.doubleValue())
            .toSpatialDatabaseRecordList().iterator();
        if (!results.hasNext()) {
            SpatialDatabaseRecord point = pointsLayer.add(edgePoint);
            node.createRelationshipTo(point.getGeomNode(), SpatialRelationshipTypes.NETWORK);
            edgePointCounter++;
        } else {
            while (results.hasNext()) {
                node.createRelationshipTo(results.next().getGeomNode(), RelationshipTypes.NETWORK);
            }
        }
    }

    private double distance(LineString line) {
        double length = 0.0;
        for (int i = 0; i < line.getNumPoints() - 1; i++) {
            length = length + distance(line.getPointN(i), line.getPointN(i + 1));
        }
        return length;
    }

    private double distance(final Point point1, final Point point2) {
        DefaultEllipsoid WGS84 = DefaultEllipsoid.WGS84;
        return WGS84.orthodromicDistance(
            point1.getCoordinate().x, point1.getCoordinate().y, //lonA, latA
            point2.getCoordinate().x, point2.getCoordinate().y); //lonB, latB
    }
}
```


Creating network topology in SQL

```
-- create network topology from OSM data
-- clean up existing tables
DROP TABLE IF EXISTS network CASCADE;
DROP TABLE IF EXISTS vertices_tmp CASCADE;
CREATE TABLE network(gid serial, OSM_id INTEGER, name VARCHAR, the_geom GEOMETRY, source INTEGER, target INTEGER,
length FLOAT);

CREATE OR REPLACE FUNCTION create_network() RETURNS text AS $$
DECLARE
streetRecord record;
wayRecord record;
pointCount integer;
pointIndex integer;
geomFragment record;
BEGIN -- start the transaction
FOR streetRecord IN SELECT way, OSM_id, name FROM planet_osm_line
WHERE highway IS NOT NULL AND highway NOT IN ('cycleway','footway','pedestrian','service') LOOP
SELECT * FROM planet_osm_ways
WHERE id = streetRecord.osm_id INTO wayRecord;
FOR pointIndex IN array_lower(wayRecord.nodes, 1)..array_upper(wayRecord.nodes,1)-1 LOOP
SELECT st_makeline(st_pointn(streetRecord.way, pointIndex), st_pointn(streetRecord.way, pointIndex+1)) AS way
INTO geomFragment;
INSERT INTO network(OSM_id, name, the_geom, source, target, length)
VALUES(streetRecord.osm_id,
streetRecord.name,
geomFragment.way,
wayRecord.nodes[pointIndex],
wayRecord.nodes[pointIndex+1],
st_length(ST_GeogFromWKB(geomFragment.way),
false));
END LOOP;
END LOOP;
return 'Done';
END;
$$ LANGUAGE 'plpgsql';

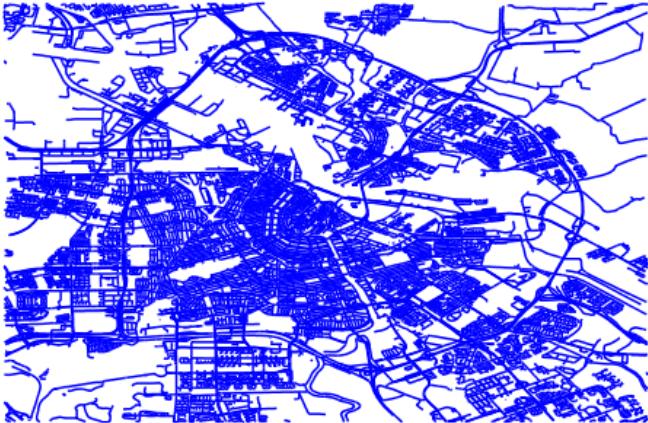
SELECT * FROM create_network();
-- clean up null values
DELETE FROM network WHERE LENGTH IS NULL;
-- fill in topology table's geometry column
INSERT INTO geometry_columns(f_table_catalog, f_table_schema, f_table_name, f_geometry_column, coord_dimension,
srid, "type")
SELECT '', 'public', 'network', 'the_geom', ST_CoordDim(the_geom), ST_SRID(the_geom), GeometryType(the_geom)
FROM network LIMIT 1;
SELECT assign_vertex_id('network', 0.00002, 'the_geom', 'gid');
```

Appendix XII Networks

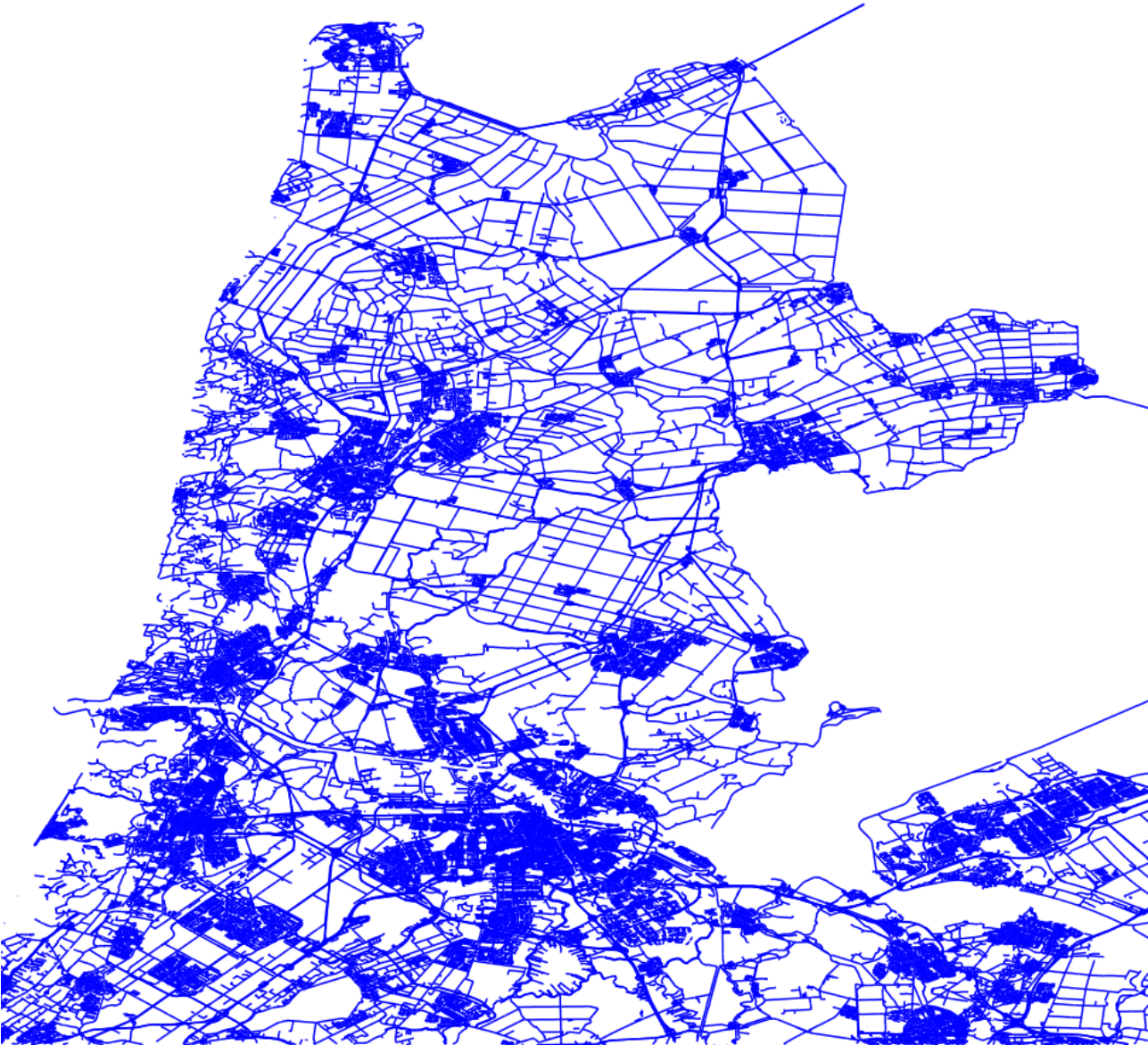
Medemblik



Amsterdam



North-Holland



Appendix XIII Source code, GUI's part

The complete source code of the assessment systems is available online:

Neo4j: <http://github.com/bartbaas/gima-neo4jtests>

PostGIS: <http://github.com/bartbaas/gima-postgistests>

Neo4j dashboard (Java)

```
package gima.neo4j.testsuite.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.AbsolutePanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TabPanel;
import com.google.gwt.user.client.ui.FlexTable;
import com.google.gwt.user.client.ui.HasHorizontalAlignment;
import com.google.gwt.user.client.ui.Image;
import com.google.gwt.user.client.ui.VerticalPanel;
import gima.neo4j.testsuite.shared.Messages;

/**
 * Entry point classes define onModuleLoad().
 */
public class tests implements EntryPoint {

    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {
        initRootPanel();
    }

    /**
     * This method build the components of the DashBoard.
     */
    private void initRootPanel() {
        RootPanel rootPanel = RootPanel.get();

        VerticalPanel verticalPanel = new VerticalPanel();
        verticalPanel.setHorizontalAlignment(HasHorizontalAlignment.ALIGN_CENTER);
        rootPanel.add(verticalPanel, 10, 10);
        verticalPanel.setSize("600px", "600px");

        AbsolutePanel absolutePanel = new AbsolutePanel();
        absolutePanel.setSize("600px", "80px");
        verticalPanel.add(absolutePanel);

        Image imgNeo = new Image("img/neo4j_logo.jpg");
        absolutePanel.add(imgNeo, 50, 0);

        Image imgGima = new Image("img/gima_logo.jpg");
        absolutePanel.add(imgGima, 450, 0);

        Label lblMainLabel = new Label("Neo4j DashBoard");
        absolutePanel.add(lblMainLabel);
        lblMainLabel.setStyleName("gwt-MainLabel");
        lblMainLabel.setSize("400px", "100");

        TabPanel tabDashboard = new TabPanel();
        verticalPanel.add(tabDashboard);

        VerticalPanel tabPanelMedemblik = new VerticalPanel();
        VerticalPanel tabPanelAmsterdam = new VerticalPanel();
        VerticalPanel tabPanelNL = new VerticalPanel();
    }
}
```

```

GridTable tableMedemblik = new GridTable();
tableMedemblik.name = "Medemblik";
tableMedemblik.description = "Spatial tests on Open Street Map data from the Medemblik area";
tableMedemblik.db = Messages.Db.MEDEMBLIK;
tableMedemblik.routes = new double[][]{
    {5.09060, 52.76522, 5.10949, 52.76080},
    {5.11160, 52.77358, 5.10554, 52.76486},
    {5.09623, 52.77227, 5.10315, 52.76724},
    {5.10528, 52.76338, 5.10885, 52.76078}};
tableMedemblik.bboxes = new double[][]{
    {5.09623, 52.77227, 5.10315, 52.76724}, //Meerlaan
    {5.10528, 52.76338, 5.10885, 52.76078},
    {5.09492, 52.77134, 5.11173, 52.76151},
    {5.10725, 52.77271, 5.11791, 52.76455}};
tableMedemblik.points = new double[][]{
    {5.09060, 52.76522},
    {5.10949, 52.76080},
    {5.11160, 52.77358},
    {5.10554, 52.76486}};
tableMedemblik.setSize("100%", "330px");
tabDashboard.addTab(tabPanelMedemblik, tableMedemblik.Name(), false);
tableMedemblik.SetupGui();
tabPanelMedemblik.add(tableMedemblik);
tabPanelMedemblik.add(tableMedemblik.LogPanel());

GridTable tableAmsterdam = new GridTable();
tableAmsterdam.name = "Amsterdam";
tableAmsterdam.description = "Spatial tests on Open Street Map data from the Amsterdam area";
tableAmsterdam.db = Messages.Db.AMSTERDAM;
tableAmsterdam.routes = new double[][]{
    {4.8862, 52.3677, 4.8594, 52.3576}, //4.8949 52.3692
    {4.8620, 52.3282, 4.8873, 52.3257},
    {4.9117, 52.4053, 4.9489, 52.3884},
    {4.9217, 52.3602, 4.9412, 52.3302}};
tableAmsterdam.bboxes = new double[][]{
    {4.88557, 52.37674, 4.91214, 52.36694}, //Centre of Amsterdam
    {4.84779, 52.37478, 4.86592, 52.36498}, //Old West
    {4.94322, 52.34834, 4.96767, 52.33037},
    {4.81342, 52.43037, 4.98180, 52.32305}};
tableAmsterdam.points = new double[][]{
    {4.8594, 52.3576},
    {4.8799, 52.3931},
    {4.9465, 52.3973},
    {4.9412, 52.3302}};
tableAmsterdam.setSize("100%", "330px");
tabDashboard.addTab(tabPanelAmsterdam, tableAmsterdam.Name(), false);
tableAmsterdam.SetupGui();
tabPanelAmsterdam.add(tableAmsterdam);
tabPanelAmsterdam.add(tableAmsterdam.LogPanel());

GridTable tableNL = new GridTable();
tableNL.name = "North-Holland";
tableNL.description = "Spatial tests on Open Street Map data from North-Holland";
tableNL.db = Messages.Db.NH;
tableNL.routes = new double[][]{
    {4.8862, 52.3677, 4.8594, 52.3576},
    {4.8620, 52.3282, 4.8873, 52.3257},
    {5.09060, 52.76522, 4.88557, 52.37674}, //Medemblik to Amsterdam
    {5.09060, 52.76522, 5.10949, 52.76080}};
tableNL.bboxes = new double[][]{
    {5.09623, 52.77227, 5.10315, 52.76724}, //Meerlaan
    {5.10528, 52.76338, 5.10885, 52.76078},
    {4.81342, 52.43037, 4.98180, 52.323}, //Amsterdam
    {4.73166, 52.68332, 4.98294, 52.37542}}; // Big part of Nort-Holland
tableNL.points = new double[][]{
    {4.8799, 52.3931},
    {4.9465, 52.3973},
    {5.11160, 52.77358},
    {5.10554, 52.76486}};
tableNL.setSize("100%", "330px");
tabDashboard.addTab(tabPanelNL, tableNL.Name(), false);
tableNL.SetupGui();
tabPanelNL.add(tableNL);
tabPanelNL.add(tableNL.LogPanel());

tabDashboard.selectTab(0);
}
}

```

PostGIS dashboard (bash)

```
#!/bin/bash
python -c'import time; print repr(time.time()''

DATASET=$1

if [ -z "$DATASET" ]; then          # -n tests to see if the argument is non empty
    DATASET="medemblik"
fi

function reload {
    exec ./dashboard.sh $1
}

function backtodashboard {
    read -p "Press ENTER for the dashboard or q to exit: " choice

    case $choice in
    q) echo "quit";;
    *) exec ./dashboard.sh $DATASET;;
    esac
}

function timer {
    STARTTIME=$(python -c'import time; print repr(time.time()''')
    # do something
    # start your script work here
    echo "Executing:" $1 "for dataset" $DATASET
    ./$1 $DATASET
    # your logic ends here
    ENDTIME=$(python -c'import time; print repr(time.time()''')
    ECHO "Elapsed time: " $(bc -l <<< $ENDTIME-$STARTTIME)
    backtodashboard
}

# clear the screen
tput clear

# Move cursor to screen location X,Y (top left is 0,0)
tput cup 1 5

# Set reverse video mode
tput rev
echo "PostGIS DashBoard -" $DATASET | tr a-z A-Z
tput sgr0

# Set a foreground colour using ANSI escape
tput setaf 7
tput cup 2 5
echo "Spatial tests with Open Street Map data on PostGIS"
tput sgr0

# Set a foreground colour using ANSI escape
tput setaf 7
tput setab 6
tput cup 4 5
echo "A) Medemblik"
tput cup 4 25
echo "B) Amsterdam"
tput cup 4 45
echo "C) North-Holland"
tput sgr0

# Set a foreground colour using ANSI escape
tput setaf 1
tput cup 6 5
echo "Instance"
tput cup 12 5
echo "Data"
tput cup 6 45
echo "Tests"
tput sgr0
```

```

tput cup 7 5
echo "1. Start database"
tput cup 8 5
echo "2. Database statistics"
tput cup 9 5
echo "3. Stop database"
tput cup 10 5
echo "4. Delete database"
tput cup 13 5
echo "5. Import osm data"
tput cup 14 5
echo "6. Create route topology"
tput cup 7 45
echo "7. Empty SQL call"
tput cup 8 45
echo "8. Test1 - bounding box"
tput cup 9 45
echo "9. Test2 - closest point"
tput cup 10 45
echo "10. Test3 - route"
tput cup 11 45
echo "11. Test4 - bounding box gml"
tput cup 12 45
echo "12. <..>"

tput cup 16 5
echo "Note: enter 'cmd' for the psql commandline"
# Set bold mode
tput bold
tput cup 18 5
read -p "Enter your choice [A-C] or [1-12] or q to exit: " choice
tput sgr0

#call the required shell or set QUIT and continue the loop, or continue the loop on any other input
case $choice in
A) reload "medemblik";;
B) reload "amsterdam";;
C) reload "north-holland";;
1) timer scripts/postgis_start.sh;;
2) timer scripts/statistics.sh;;
3) timer scripts/postgis_stop.sh;;
4) timer scripts/postgis_drop.sh $DATASET;;
5) timer scripts/import_osm.sh $DATASET;;
6) timer scripts/create_topology.sh $DATASET;;
7) timer scripts/test_empty.sh $DATASET;;
8) timer scripts/test_bbox_$DATASET.sh;;
9) timer scripts/test_closepoint_$DATASET.sh;;
10) timer scripts/test_route_$DATASET.sh;;
11) timer scripts/test_gml_$DATASET.sh;;
12) echo "not implemented yet";;
cmd) psql -d $DATASET;;
q) exit;;
*) reload;;
esac

```

Appendix XIV Source code tests

The complete source code of the assessment systems is available online:

Neo4j: <http://github.com/bartbaas/gima-neo4jtests>

PostGIS: <http://github.com/bartbaas/gima-postgistests>

Java: Count geometries inside bounding box

```
Envelope bbox = new Envelope(coords[i][0], coords[i][2], coords[i][3], coords[i][1]);

GeoPipeline pipeline = OSMGeoPipeline
    .startWithinSearch(layer, layer.getGeometryFactory().toGeometry(bbox))
    .copyDatabaseRecordProperties()
    .getGeometryType();

for (GeoPipeFlow flow : pipeline) {
    if (flow.getProperties().get("GeometryType") == "LineString") {
        numLine++;
    } else if (flow.getProperties().get("GeometryType") == "Point") {
        numPoint++;
    } else if (flow.getProperties().get("GeometryType") == "Polygon") {
        numPoly++;
    } else {
        System.out.println(flow.getProperties().get("GeometryType"));
    }
    numTotal++;
}
```

SQL: Count geometries inside bounding box

```
SELECT *, lines+points+polygons AS total FROM
    (SELECT (
        SELECT COUNT(*)
        FROM planet_osm_point WHERE ST_Within(way, ST_MakeEnvelope(:bbox))
    ) AS points,
    (
        SELECT COUNT(*)
        FROM planet_osm_line WHERE ST_Within(way, ST_MakeEnvelope(:bbox))
    ) AS lines,
    (
        SELECT COUNT(*)
        FROM planet_osm_polygon WHERE ST_Within(way, ST_MakeEnvelope(:bbox))
    ) AS polygons
    ) AS bbox;
```

Java: Get geometries GML inside bounding box

```
Envelope bbox = new Envelope(coords[i][0], coords[i][2], coords[i][3], coords[i][1]);

GeoPipeline pipeline = OSMGeoPipeline
    .startWithinSearch(layer, layer.getGeometryFactory().toGeometry(bbox))
    .createGML();

for (GeoPipeFlow flow : pipeline) {
    out.println(flow.getProperties().get("GML"));
}
```

SQL: Get geometries GML inside bounding box

```
-- bounding box GML query for all OSM geometry columns
\pset pager
SELECT ST_AsGML(way,3) FROM planet_osm_point WHERE ST_Within(way, ST_MakeEnvelope(:bbox))
UNION ALL
SELECT ST_AsGML(way,3) FROM planet_osm_line WHERE ST_Within(way, ST_MakeEnvelope(:bbox))
UNION ALL
SELECT ST_AsGML(way,3) FROM planet_osm_polygon WHERE ST_Within(way, ST_MakeEnvelope(:bbox));
```


Java: Shortest path

```
public String dijkstra() {
    String result;
    GraphDatabaseService databaseService = spatialDatabaseService.getDatabase();
    Transaction tx = databaseService.beginTx();
    try {
        // point <- edge -> point <- edge -> point
        Dijkstra<Double> sp = new Dijkstra<Double>(
            0.0,
            startNode,
            endNode,
            new CostEvaluator<Double>() {
                public Double getCost(Relationship relationship, Direction direction) {
                    Node startNd = relationship.getStartNode();
                    if (direction.equals(Direction.INCOMING)) {
                        return (Double) startNd.getProperty("_distance");
                    } else {
                        return 0.0;
                    }
                }
            },
            new DoubleAdder(),
            new DoubleComparator(),
            Direction.BOTH,
            SpatialRelationshipTypes.NETWORK);

        List<Node> pathNodes = sp.getPathAsNodes();
        result = "Length: " + sp.getCost().longValue() + ", Segments: " + pathNodes.size();
        tx.success();
    } catch (Exception e) {
        System.out.println(e.getMessage());
        result = "Error finding Path, Unable to find Path";
    } finally {
        tx.finish();
    }
    return result;
}
```

SQL: Shortest path

```
-- shortest path based on node id
CREATE OR REPLACE FUNCTION get_network_id (text) RETURNS integer AS $$
DECLARE
    -- Declare aliases for user input.
    ewkt ALIAS FOR $1;
    -- Declare a variable to hold the customer ID number.
    id INTEGER;
BEGIN
    -- Retrieve the node ID closest to the input-coordinate
    Select source_id, ST_Distance_Spheroid(pt, ST_ClosestPoint(line,pt), 'SPHEROID["WGS
84",6378137,298.257223563]') AS dist INTO id
FROM (SELECT ST_GeomFromEWKT(ewkt)::geometry AS pt, the_geom AS line, source AS source_id from network) AS Points
ORDER BY dist LIMIT 1;
    -- Return the ID number
    RAISE NOTICE 'Found closest node: %',id;
    RETURN id;
END;
$$ LANGUAGE 'plpgsql';

SELECT SUM(cost) AS distance FROM shortest_path('SELECT gid AS id, source, target, length AS cost FROM network',
get_network_id(:start), get_network_id(:end),false,false);
```

Appendix XV Results of the operations

Spatial bounding box count operations (B)

Results for the Medemblik dataset in WGS84 (longitude, latitude):

```
Input: 5.09623° E, 52.77227° N & 5.10315° E, 52.76724° N
points | lines | polygons | total
-----+-----+-----+-----
      8 |    78 |    133 |   219
```

```
Input: 5.10528° E, 52.76338° N & 5.10885° E, 52.76078° N
points | lines | polygons | total
-----+-----+-----+-----
      1 |    33 |     37 |    71
```

```
Input: 5.09492° E, 52.77134° N & 5.11173° E, 52.76151° N
points | lines | polygons | total
-----+-----+-----+-----
     56 |   356 |    637 |  1049
```

```
Input: 5.10725° E, 52.77271° N & 5.11791° E, 52.76455° N
points | lines | polygons | total
-----+-----+-----+-----
     47 |   138 |    345 |   530
```

Results for the Amsterdam dataset in WGS84 (longitude, latitude):

```
Input: 4.88557° E, 52.37674° N & 4.91214° E, 52.36694° N
points | lines | polygons | total
-----+-----+-----+-----
   864 |  1159 |    475 |  2498
```

```
Input: 4.84779° E, 52.37478° N & 4.86592° E, 52.36498° N
points | lines | polygons | total
-----+-----+-----+-----
   840 |   641 |    158 |  1639
```

```
Input: 4.94322° E, 52.34834° N & 4.96767° E, 52.33037° N
points | lines | polygons | total
-----+-----+-----+-----
   137 |   867 |   1253 |  2257
```

```
Input: 4.81342° E, 52.43037° N & 4.98180° E, 52.32305° N
points | lines | polygons | total
-----+-----+-----+-----
 11795 | 30259 |   32180 | 74234
```

Results for the North-Holland dataset in WGS84 (longitude, latitude):

```
Input: 5.09623° E, 52.77227° N & 5.10315° E, 52.76724° N
points | lines | polygons | total
-----+-----+-----+-----
      8 |    78 |    133 |   219
```

```
Input: 5.10528° E, 52.76338° N & 5.10885° E, 52.76078° N
points | lines | polygons | total
-----+-----+-----+-----
      1 |    33 |     37 |    71
```

```
Input: 4.81342° E, 52.43037° N & 4.98180° E, 52.32305° N
points | lines | polygons | total
-----+-----+-----+-----
```

```
11795 | 30253 | 32180 | 74228
```

```
Input: 4.73166° E, 52.68332° N, 4.98294° E, 52.37542° N
```

```
points | lines | polygons | total
```

```
-----+-----+-----+-----  
7073 | 38347 | 105825 | 151245
```

Spatial bounding box get operations (G)

Results for the Medemblik dataset:

```
medemblik.gml, 410 KB
```

Results for the Amsterdam dataset:

```
amsterdam.gml, 16,2 MB
```

Results for the North-Holland dataset:

```
north-holland.gml, 54,8 MB
```

Closest point operations (C)

Results for the Medemblik area in WGS84 (longitude, latitude):

Input: 5.09060° E, 52.76522° N	id	dist
	599541523	136.206156318246

Input: 5.10949° E, 52.76080° N	id	dist
	1379310095	341.492231778163

Input: 5.11160° E, 52.77358° N	id	dist
	521258227	50.7270316696754

Input: 5.10554° E, 52.76486° N	id	dist
	524425089	54.878090084591

Results for the Amsterdam area in WGS84 (longitude, latitude):

Input: 4.8594° E, 52.3576° N	id	dist
	540202609	4.16239051099858

Input: 4.8799° E, 52.3931° N	id	dist
	1119939508	112.90413901513

Input: 4.9465° E, 52.3973° N	id	dist
	599588715	166.40219536549

Input: 4.9412° E 52.3302° N	id	dist
	567903853	16.6314691560348

Results for the North-Holland area in WGS84 (longitude, latitude):

Input: 4.87990° E, 52.39310° N	id	dist
	1119939508	112.90413901513

Input: 4.94650° E 52.39730° N	id	dist
	599588716	166.40219536549

Input: 5.11160° E, 52.77358° N	id	dist
	521258227	50.7270316696754

Input: 5.10554° E, 52.76486° N	id	dist
	524425089	54.878090084591

Shortest path operations (P)

Results for the Medemblik area in WGS84 (longitude, latitude):

Start point: 5.09060° E, 52.76522° N; end point: 5.10949° E, 52.76080° N
distance: 1733.93155080236

Start point: 5.11160° E, 52.77358° N; end point: 5.10554° E, 52.76486° N
distance: 1362.82008637301

Start point: 5.09623° E, 52.77227° N; end point: 5.10315° E, 52.76724° N
distance: 806.468832837435

Start point: 5.10528° E, 52.76338° N; end point: 5.10885° E, 52.76078° N
distance: 431.153474449167

Results for the Amsterdam area in WGS84 (longitude, latitude):

Start point: 4.8862° E, 52.3677° N; end point: 4.8594° E, 52.3576° N
distance: 2601.44336740279

Start point: 4.8620° E, 52.3282° N; end point: 4.8873° E, 52.3257° N
distance: 2080.60715699327

Start point: 4.9117° E, 52.4053° N; end point: 4.9489° E, 52.3884° N
distance: 4369.91360520342

Start point: 4.9217° E, 52.3602° N; end point: 4.9412° E, 52.3302° N
distance: 4840.5173340848

Results for the North-Holland area in WGS84 (longitude, latitude):

Start point: 4.88620° E, 52.36770° N; end point: 4.85940° E, 52.35760° N
distance: 2601.44336740279

Start point: 4.86200° E, 52.32820° N; end point: 4.88730° E, 52.32570° N
distance: 2080.60715699327

Start point: 5.09060° E, 52.76522° N; end point: 4.88557° E, 52.37674° N
distance: 52206.8327247351

Start point: 5.09060° E, 52.76522° N; end point: 5.10949° E, 52.76080° N
distance: 1733.93155080236

Appendix XVI Tables of measurements

Medemblik

type	import	import	network	network	empty	empty	count	count	gml	gml	closeto	closeto	dijkstra	dijkstra
	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis
i	s	s	s	s	s	s	s	s	s	s	s	s	s	s
1	12	5	3	1,2	0,02	0,03	0,6	0,4	3,9	0,55	0,12	0,44	0,12	0,7
2	8	4	5,5	0,9	0,03	0,02	1,1	0,38	3,5	0,56	0,1	0,31	0,28	0,8
3	10	4	4	0,9	0,03	0,03	1,21	0,41	3,6	0,58	0,15	0,38	0,12	0,7
4	9	4	3	1	0,04	0,02	0,75	0,42	3,8	0,57	0,09	0,3	0,13	0,74
5	10	3	4	1,1	0,02	0,03	1,3	0,38	3,5	0,56	0,15	0,5	0,11	0,8
6	10	4	5	0,8	0,02	0,08	0,77	0,39	3,9	0,55	0,1	0,6	0,3	0,68
7	8	5	3	1,2	0,02	0,04	1,6	0,4	4,1	0,54	0,14	0,48	0,077	0,6
8	9	3	4,5	0,9	0,03	0,03	0,75	0,4	3,9	0,59	0,1	0,46	0,14	0,88
9	10	4	3,5	1	0,02	0,06	0,76	0,42	4,2	0,57	0,13	0,32	0,12	0,82
10	12	4	3	1,1	0,02	0,02	0,8	0,37	3,8	0,55	0,11	0,3	0,12	0,73
mean	9,75	4	3,75	1,013	0,024	0,033	0,930	0,398	3,813	0,561	0,119	0,399	0,143	0,746

Amsterdam

type	import	import	network	network	empty	empty	count	count	gml	gml	closeto	closeto	dijkstra	dijkstra
	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis
i	s	s	s	s	s	s	s	s	s	s	s	s	s	s
1	550	49	277	429	0,02	0,02	12	0,95	111,8	11,2	0,5	0,5	1,2	3,3
2	560	46	322	404	0,01	0,03	17	0,98	121,5	9,9	0,56	0,48	1,3	3,6
3	663	55	502	399	0,03	0,04	14	0,99	108,4	10,9	0,49	0,48	0,9	3,5
4	630	51	401	531	0,03	0,01	16	1,07	111,9	10,5	0,52	0,5	1,3	3,5
5	606	42	294	429	0,02	0,02	14	0,96	110,1	11	0,54	0,58	1,2	4,1
6	583	41	305	443	0,01	0,02	16	1,02	109	10,8	0,5	0,57	1,5	3,5
7	592	52	338	450	0,02	0,04	14	0,98	108,7	11,1	0,53	0,48	1,8	3,8
8	643	46	462	412	0,03	0,03	15	1,01	112,8	10,2	0,55	0,5	1,2	3,6
9	620	43	375	408	0,02	0,02	15	1,06	113,1	10,8	0,56	0,55	1,3	3,3
10	570	54	312	463	0,02	0,02	14	1,08	111,2	10,3	0,5	0,48	1,2	3,5
mean	601	48	351	430	0,021	0,025	14,750	1,009	111,075	10,700	0,525	0,508	1,275	3,538

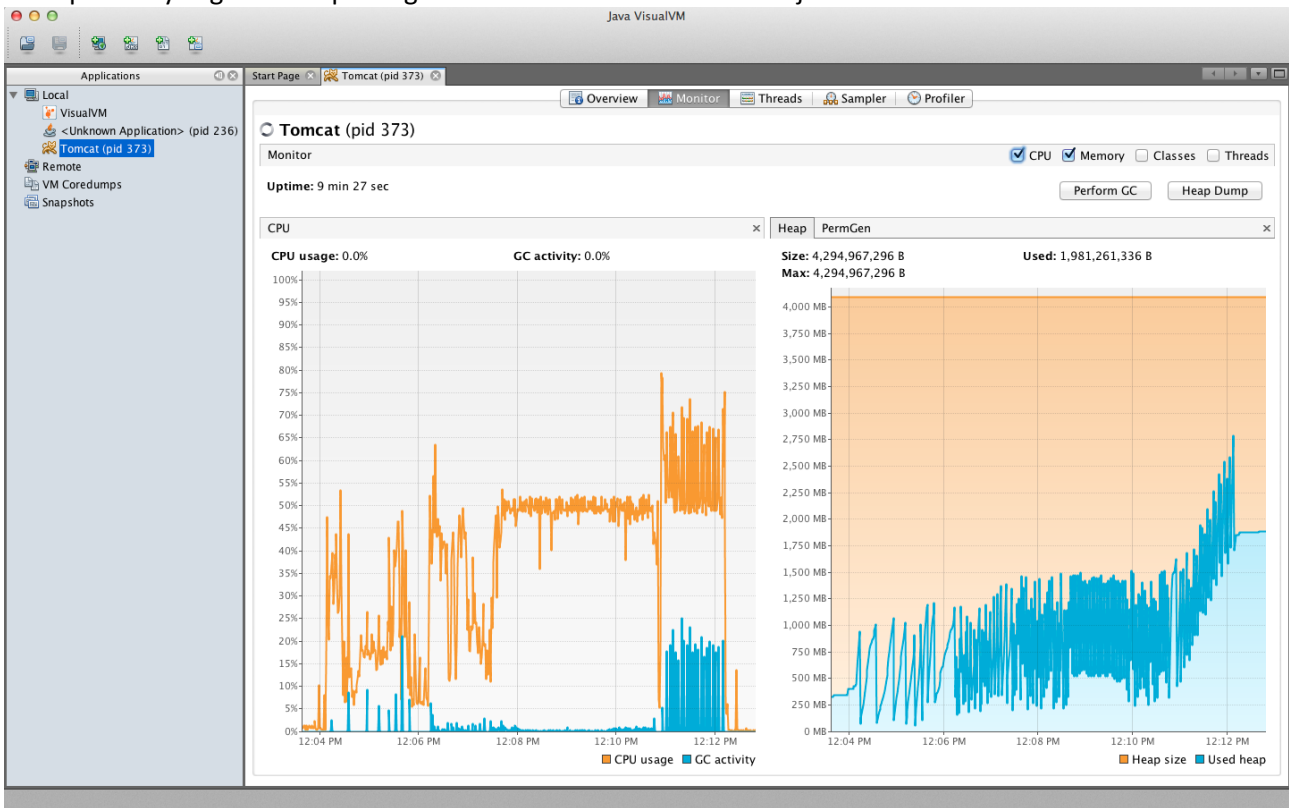
North-Holland

type	import	import	network	network	empty	empty	count	count	gml	gml	closeto	closeto	dijkstra	dijkstra
	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis	neo4j	postgis
i	s	s	s	s	s	s	s	s	s	s	s	s	s	s
1	13955	627	8366	26862	0,03	0,03	58	2,9	706	42,6	1,8	0,9	5,9	29,8
2	12633	474	8184	27964	0,01	0,04	52	2,4	698	40,3	2,3	0,9	4,8	29,4
3	15475	714	8094	27418	0,02	0,03	54	2,5	701	41,8	2	1	6	29,6
4	14032	631	8279	27771	0,02	0,02	55	2,6	712	42,2	1,9	0,8	4,7	30,1
5	13411	719	8461	27062	0,03	0,03	56	2,8	689	40,5	2,1	0,9	4,5	29,3
6	13076	712	8231	27015	0,01	0,03	57	2,9	703	40,8	2,2	1	5,9	29,6
7	14327	632	8286	26868	0,02	0,03	58	2,5	706	42	1,8	1	4,9	29,1
8	13291	637	8458	26988	0,01	0,03	59	2,8	702	41,7	2,2	0,8	6,1	29,3
9	15249	692	8374	27231	0,02	0,02	60	3	673	45,7	1,9	0,9	4,7	28,9
10	13583	660	8098	27170	0,03	0,02	61	2,5	710	44,1	2	1	4,5	29,7
mean	13866	663	8285	27190	0,020	0,028	57,125	2,688	701,875	41,963	2,013	0,925	5,175	29,475

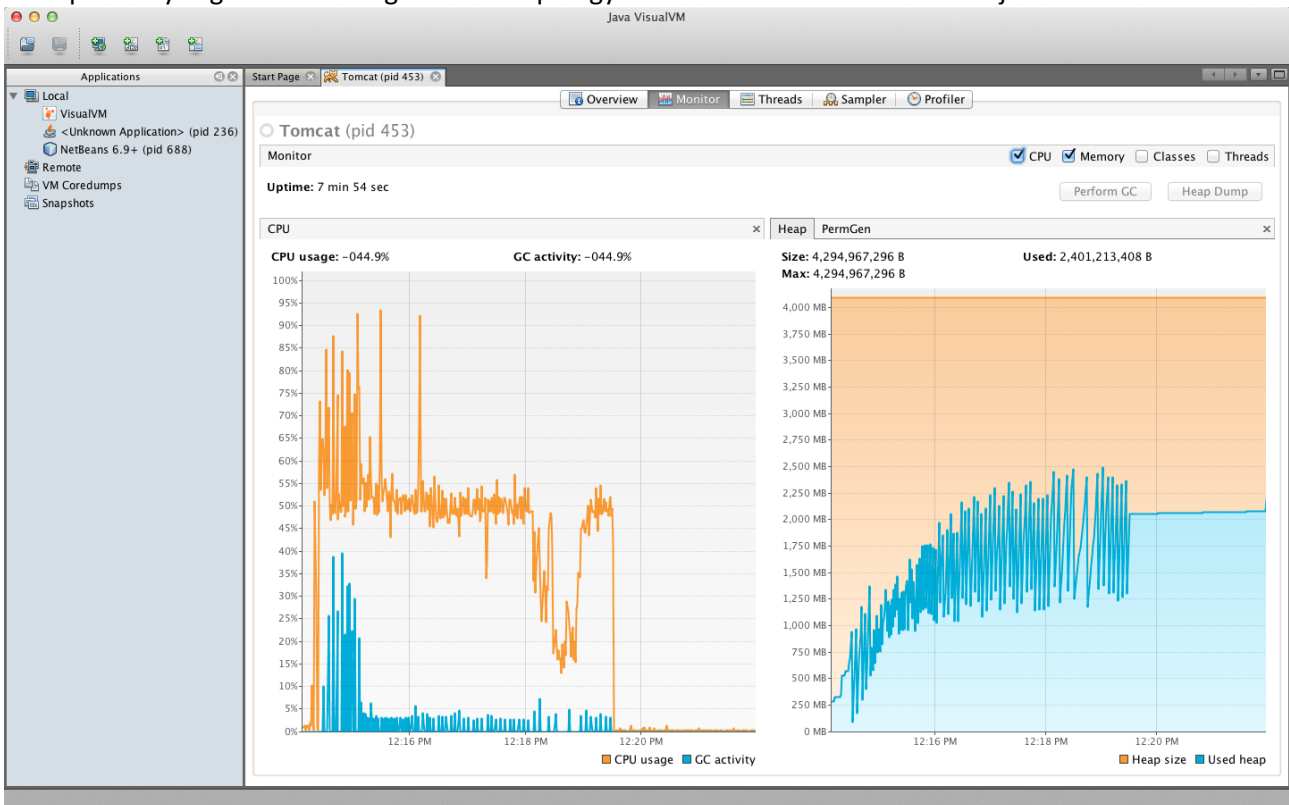
Appendix XVII Memory measurements Neo4j

Visualvm graph

Example analyzing while importing Amsterdam OSM file into Neo4j



Example analyzing while creating network topology for Amsterdam dataset in Neo4j



Results in a table for all area's

Measured correlation between primitives and memory

Name	OSM-file	Area	Primitives	Nodes	Relationships	Properties	Heap space	Stack space
Entity	(MB)	(km2)	(count)	(count)	(count)	(count)	(GB)	(GB)
Medemblik	4	6,3	198403	50069	80660	67674	0,5	0,05
Amsterdam	106	100,0	5261177	1328724	2111097	1821356	2	0,25
North-Holland	1229	4800,0	61050667	15469086	24771181	20810400	6	1
Netherlands	10691	81000,0	553886890	129848780	199395099	224643011	16	3

Regression primitives and RAM (approximated)

primitives	osm-file	log10	regression1	regression2	ram	ram	regression	sqrt
(count)	(MB)	(primitives)	$2^{(\log_{10}-1)}$	$\sqrt{\text{reg1}}$	(GB)	(GB)		(regression)
10	0,0002	1	1	1	0,0005	0,0005	—	
100	0,002	2	2	4	0,002	0,002	4	2
1000	0,02	3	4	16	0,008	0,008	16	4
10000	0,2	4	8	64	0,032	0,032	64	8
100000	2	5	16	256	0,128	0,128	256	16
1000000	20	6	32	1024	0,512	0,512	1024	32
10000000	200	7	64	4096	2,048	2,048	4096	64
100000000	2000	8	128	16384	8,192	8,192	16384	128
1000000000	20000	9	256	65536	32,768	32,768	65536	256
10000000000	200000	10	512	262144	131,072	131,072	262144	512