MAPPING GEOGRAPHIC CONCEPTUAL SCHEMAS TO NOSQL GRAPH DATABASES

DANILO BOECHAT SEUFITELLI

MAPEAMENTO DE ESQUEMAS CONCEITUAIS GEOGRÁFICOS PARA BANCOS DE DADOS NOSQL EM GRAFO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Mirella Moura Moro
Coorientador: Clodoveu A. Davis Jr.

Belo Horizonte
Agosto de 2016

DANILO BOECHAT SEUFITELLI

MAPPING GEOGRAPHIC CONCEPTUAL SCHEMAS TO NOSQL GRAPH DATABASES

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Mirella Moura Moro
Co-Advisor: Clodoveu A. Davis Jr.

Belo Horizonte
August 2016

© 2016, Danilo Boechat Seufitelli.
Todos os direitos reservados.

Boechat Seufitelli, Danilo

S496m Mapping Geographic Conceptual Schemas to NoSQL Graph Databases / Danilo Boechat Seufitelli. — Belo Horizonte, 2016
xxiv, 56 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientadora: Mirella Moura Moro
Coorientador: Clodoveu A. Davis Jr.

1. Computação - Teses. 2. Modelagem Conceitual. 3. Bancos de dados geográficos. 4. NoSQL. 5. Bancos de dados em grafos. I. Orientadora. II. Coorientador. III. Título.

CDU 519.6*72(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Mapping geographic conceptual schemas to nosql graph databases

DANILO BOECHAT SEUFITELLI

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
PROFA. MIRELLA MOURA MORO - Orientadora
Departamento de Ciência da Computação - UFMG

[illegible]
PROF. CLODOVEU AUGUSTO DAVIS JÚNIOR - Coorientador
Departamento de Ciência da Computação - UFMG

[illegible]
DRA. KARLA ALBUQUERQUE DE VASCONCELOS BORGES
Prodabel

Renata Galanti
PROFA. RENATA DE MATOS GALANTE
Instituto de Informática - UFRGS

Belo Horizonte, 07 de dezembro de 2016.

“I dedicate to all who had a moment of weakness. It will not hurt forever, so do not let it affect what is best in you.”

Acknowledgments

First and foremost, I'd like to thank God for all the opportunities that I have had in my life. I also want to thank my advisors, Mirella and Clodoveu, for their patience and support. This work would not have been possible without their assistance.

I also want to thank my parents José Antonio and Cristiane and brothers Bruno and Claudia for the care, support, patience and understanding that I need to fly around the world to achieve my goals.

I need to thank Elaine Muniz, for her kindness, support, patience and for always believing in me. She is my daily inspiration.

My most sincere thanks to the colleagues and friends who helped so much during my studies: Alberto Ueda, Guilherme Vezula, Ivan Nunes, Janaína Henriques, Jhielson Montino, Michel Melo, Michele Brandão, Natália Gonçalves, Paulo Nonaka, Pedro Onofre, Ramon Pereira and Thiago Rodrigues.

Last but not least, I want to thank all my old friends, whose friendship has grown since I was a little kid: Aquila Ditzz, Carolina Silveira, Cynthia Medeiros, Erika Vieira, Felipe Almeida, Guilherme Borges, Rodrigo Torquato, Tatiane Martins, Thawler Andrade, Valéria Alves and Victor Silveira. If I have forgotten your name, I am so sorry. There are many important people in my life, they are too many to remember.

This work is supported by a CAPES scholarship. This financial support is gratefully acknowledged.

"Tell me and I forget, teach me and I may remember, involve me and I learn."
(Benjamin Franklin)

Resumo

A modelagem conceitual geográfica, assim como a modelagem conceitual tradicional, é uma atividade básica em projetos de aplicações geográficas. Os modelos conceituais geográficos fornecem primitivas para representar a geometria e a topologia dos dados geográficos, que são geralmente armazenados em documentos GML (*Geography Markup Language*) ou em bases de dados espaciais. O projeto conceitual deve ser independente dos mecanismos subjacentes a serem usados para a implementação do banco de dados. Um bom esquema conceitual pode ser mapeado para vários esquemas físicos, de modo que o aplicativo pode se beneficiar das melhores características do sistema de gerenciamento de banco de dados utilizado em sua implementação. Permite também a reutilização total ou parcial do esquema, uma vez que a parte modelada do mundo real pode ser percebida de forma semelhante por diferentes aplicações. No entanto, modelos conceituais que consideram dados geográficos e aplicações, tais como a Técnica de Modelo de Objeto para Aplicações Geográficas (OMT-G), incluem uma maior variedade de primitivas de representação para classes e relacionamentos. Ao mesmo tempo, vários sistemas de gerenciamento de banco de dados alternativos, que implementam novos paradigmas de representação e estruturas de dados, surgiram recentemente com o nome genérico de NoSQL (ou não apenas SQL). Esta dissertação define procedimentos e algoritmos para implementar o mapeamento entre esquemas OMT-G e esquemas lógicos e físicos baseados em grafos NoSQL. Um exemplo abrangente do processo proposto é apresentado. Conclusões indicam que mapear esquemas conceituais geográficos para esquemas híbridos (relacionais e grafos) físicos pode ser desejável no futuro, a fim de combinar as melhores características de cada alternativa de implementação.

Palavras-chave: Modelagem Conceitual, Bancos de Dados Geográficos, NoSQL, Bancos de Dados em Grafos.

Abstract

Geographic conceptual modeling, as traditional conceptual modeling, is a basic activity in the design of geographic applications. Geographic conceptual models provide primitives to represent the geometry and the topology of geographic data, which are generally stored in Geography Markup Language (GML) documents or in spatial databases. Database design at the conceptual level must be independent from the underlying mechanisms that implement the database. A good conceptual schema can be mapped to various physical schemas, so that the application can benefit from the best characteristics of the database management system used in its implementation. It also allows total or partial schema reuse, since the modeled part of the real world can be perceived in a similar manner by different applications. However, conceptual models that consider geographic data and applications, such as the Object Modeling Technique for Geographic Applications (OMT-G), include a wider variety of representation primitives for classes and relationships. At the same time, a number of alternative database management systems, which implement novel representation paradigms and data structures, have recently arisen with the generic name of NoSQL (or not only SQL). This dissertation defines rules and algorithms to implement the automatic mapping between OMT-G schemas and NoSQL graph-based logical and physical schemas. A comprehensive example of the proposed process is presented. Conclusions indicate that mapping geographic conceptual schemas to hybrid (relational and graph) physical schemas may be desirable in the future, in order to combine the best characteristics from each implementation alternative.

Keywords: Conceptual Modeling, Geographic Databases, NoSQL, Graph Databases.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Context

Many spatial data modeling methods extend the entity-relationship model for describing geographic data. While the result of such data modeling processes is usually associated with relational database management systems (DBMS), other non-relational spatial solutions are used by very large Geographic Information Systems (GIS). With the NoSQL (Not Only SQL) arrays taking over Big Data, data models that easily fit relational DBMS are being translated and adapted into other systems. However, this may not prove optimal when handling large amounts of data, as it is not so easy to choose which mapping strategy (or even more than one) to maintain, either on relational or non-relational DBMS. Therefore, modeling strategies based on the systems requirements and features are necessary.

Modeling spatial classes must consider that the association of primary and foreign keys applied to numbers and strings provide an easy way to join tables, but may not be possible with geospatial attributes. Spatial representations can be seen as a multidimensional attribute that by its nature is not prone to simple joining, as the interaction between two spatial objects may depend on geometric shapes. While a conventional join is determined by a binary predicate, the outcome of spatial relationships can be complex and must be calculated dynamically.

Considering that spatial relationships cannot be represented as foreign key associations, Borges et al. demonstrated how spatial data characteristics make modeling more complex and proposed the Object Modeling Technique for Geographic Applications (OMT-G) [5]. OMT-G uses specific UML (Unified Modeling Language) primitives in class diagrams and introduces geographic features to increase the semantic representativeness of the model. Thus, OMT-G provides primitives to model topology and geometry as part of geographic data, and defines spatial classes and relationships [5]. OMT-G is currently used by various governmental, industrial and academical organizations such as Prodabel[1], INPE[2], UFMG[3] and UFV[4] [22].

When mapping spatial objects into a DBMS, the relational systems usually end up with normalized tables (with some exceptions focusing either on performance or data redundancy) containing spatial columns. The same does not apply to NoSQL DBMSs, in which there might be more than one way to encode a spatial class. Geographic conceptual modeling is independent from physical schemas, but it should be possible to map a given schema to various types of DBMS, provided there is support for spatial data types and relationships.

1.2 Motivation

According to Newman [34], complex networks are data networks where the relationships between the elements is as or more important than the elements themselves. To represent them through graphs, the relationships are described by edges and elements by vertices. Using traditional database models to store and manage complex networks can generate bottlenecks in the data handling, due to the large amount of existing data. In addition, traditional models are not able to explore the fundamentals of graphs with respect to their relationships, neighborhoods and standards [2]. Therefore, a database model based on graphs that is able to meet the peculiarities of complex networks is more appropriate. According to Angles and Gutierrez [2], there are several advantages to using the graph model, including (i) more natural data modeling for complex networks; (ii) easier queries over the graph structure, for instance, to return adjacent vertices; (iii) a higher level of abstraction, by working directly with graphs and related operations, facilitating data manipulation by the developer; and (iv) more efficient graph algorithms in graph database implementations.

Indeed, the use of NoSQL graph databases as a DBMS solution to store data in applications in increasing, mostly because they make it easier to handle large volumes of data in an unstructured way [9]. However, integrating data between applications requires these applications to manage the same dataset. Thereby, defining rules to create logical graph schemas is an important step in the design of applications in order to promote systems interoperability. A positive point about the logical graph model is

[1]PRODABEL: https://prodabel.pbh.gov.br/
[2]INPE: http://www.inpe.br
[3]UFMG: http://www.ufmg.br
[4]UFV: http://www.ufv.br

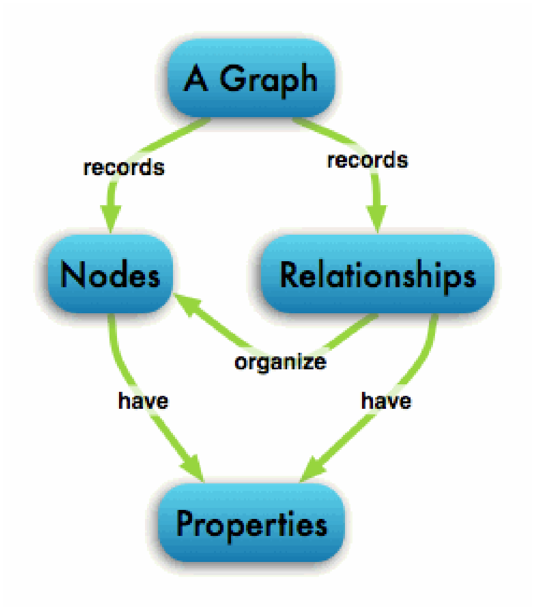


Figure 1.1: The building blocks of the Property Graph[5]

.

the independence of a DBMS's physical implementation. Logical model depends only on the DBMS type, as graph, relational, document- and column-oriented.

For example, a property graph model is a generic way to encode conceptual schemas, as in Figure 1.1. A property graph is similar to an object schema or an entity-relationship diagram. Property graphs contain connected entities (nodes) that can hold any number of attributes (key-value-pairs). Nodes can be tagged with labels representing their different roles in a domain. Labels contextualize node and relationship properties and serve to attach metadata, such as indexing or constraint information, to certain nodes. Relationships provide directed, named and semantically relevant connections between two node-entities. A relationship always has a direction, a type, a start node and an end node. As nodes, relationships can have any properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals or strengths.

Current studies [1, 4, 8, 9] do not address the mapping process from a geographic conceptual modeling to a NoSQL approach, as the graph model. Geographic data can be modeled as a graph, especially when the data represent networks, such as electrical systems, roads or rivers. Recent studies by Santos et al. [38] showed that graph databases, when used to store and query a large amount of geographic data, are able to outperform relational databases in specific query types (as urban routing and position tracing).

[5]Available at `https://neo4j.com/developer/graph-database/`. Accessed June 27, 2016.

1.3 Objective and Contributions

The objective of this dissertation is to define and implement a mapping from a geographic conceptual schema to a graph database, and is divided in two parts. First, we propose mapping procedures and algorithms to transform OMT-G schemas into logical graph schemas. Then, we develop a diagrammatic way to describe such logical schemas. We also highlight the limitations in graph modeling the logical schemas as well as the implementation requirements to handle integrity constraints. Second, we define how to map such logical graph schemas into physical graph schemas, using Neo4J DBMS with spatial extensions in order to validate the proposed logical schemas mapping process. We show that implementing integrity constraints on Neo4j is a challenge because Neo4j does not provide triggers (as in RDMBSs) to check consistency of data. Finally, we finish this work arguing that graph databases can solve parts of the schema (as network relationships) in a hybrid approach that considers the best of relational and graph models.

1.4 Organization

Chapter 2 describes related work and background concepts about spatial features in databases. We also cover spatial integrity constraints and NoSQL databases.

Chapter 3 provides an overview the OMT-G model primitives for conceptual modeling of geographic data. We explain how the modeling technique works, including diagrams to represent spatial objects and the integrity constraints adopted in the modeling, based on the original definitions from [5, 6].

Chapter 4 introduces our mapping methodology by explaining each processing stage and defining rules to convert a conceptual OMT-G schema to a logical graph schema. An algorithm is presented to guide the mapping process sequence.

Chapter 5 explains the steps for mapping the logical graph schema to a physical graph schema. We use Neo4j[6] in this step because it natively implements graph storage and a spatial plugin, Neo4j-Spatial[7].

Finally, Chapter 6 presents conclusions and future work.

[6]Neo4j: `http://www.neo4j.org/`
[7]Neo4j-Spatial: `http://www.neo4j.org/develop/spatial`

Chapter 2

Related Work

This chapter reviews relevant concepts and studies in topics related to this work. The chapter is organized as follows: Section 2.1.1 presents a summary about spatial data, Section 2.1.2 discusses the importance of spatial data modeling, Section 2.1.3 provides a basic view on how spatial database management systems (DBMSs) work, including spatial data management. Section 2.1.4 highlights the main strategies to ensure data integrity, while Section 2.1.5 refers to spatial data integrity constraints. Section 2.1.6 revises the main concepts of NoSQL databases. Section 2.1.7 presents the Neo4j DBMS. Section 2.2 discusses the related work and, finally, Section 2.3 presents our final considerations.

2.1 Concepts and Current Technology

2.1.1 Spatial Data

Spatial data are any type of data that describe phenomena that are associated with any spatial dimension [13, 26]. Nowadays, there is a large number of capable devices to register geographic information (as smartphones, tablets, gps and others), as well as an increasing number of applications that use such kind of data (as Facebook[1], Instagram[2], Waze[3] and others).

Geographic data have *geometry* and *topology*. Geometry is for property metrics wherein relationships are defined starting from primitive geometric features as points, lines and polygons that depict geometric entities. There are two geometric data types:

[1]Facebook: `http://www.fb.com`
[2]Instagram: `http://www.instagram.com`
[3]Waze: `http://www.waze.com`

Vector data and Raster data. Vector data include points, lines, and polygons, all of which are representations of the space occupied by real-world entities. Raster data are characterized as an array of points, where each point represents the value of an attribute for a real-world area [28, 41].

Topological properties (non-metrics) are based in relative positions of objects in the space, as connectivity, guidance (from, to), adjacency and contention. Some geographic entities have topological properties that are unaltered by elastic deformations. Examples include the link of a region and the connectivity between road intersections and road segments. Primitives are required for representing networks, graphs, and partitions as high-level entities [19]. Partitions relate to networks which associate regions with other regions by relationships, such as next or adjacent. It is natural to use a direct construction for networks and dividers in modeling, for example. Some spatial concepts can be measured both in geometric and topological domain[28].

2.1.2 Spatial Data Modeling

The spatial data model is the set of concepts that describe the structure and the operation of a geographic database. Each component can be visualized in different levels of complexity and details, i.e. according to the need of comprehension and representation from different entities of information system interest and their interactions. There are two main conceptual models to user for geographic data: GeoFrame and OMT-G.

GeoFrame, proposed by Jugurta and Iochpe [30], provides a diagram of basic classes, using the graphical UML notation. The GeoFrame has two base classes for any geographic applications, the classes *THEME* and *GEOGRAPHICREGION*. All geographic applications have as main objective the management and manipulation of a set of data for a given region of interest, creating a geographic database. For example, in a GIS application, the urban area can be specified as the geographic region of interest (*GEOGRAPHICREGION*). For such a geographic regions, the following topics (*THEME*) could be defined: limits of the urban area, road network, neighborhoods, buildings (e.g. schools, hospitals), public transport, and areas of garbage collection.

OMT-G, proposed by Borges et al. [5], is an object-oriented data model to geographic applications that provide primitives to model geometry and topology of spatial data. It also supports different topological structures, multiple views of objects and the spatial relationships. We chose OMT-G as conceptual modeling technique because it was developed by our research team and has the OMT-G Designer, an online diagramming application to design geographic database systems and applications based on OMT-G data model [31]. Therefore, OMT-G is further detailed in Section 3.

2.1.3 Spatial Database Management Systems

A Spatial Database Management System offers data and spatial queries by using spatial indexes and efficient algorithms to manipulate geographic information [19]. A spatial database supports both conventional and spatial data. Conventional data describe the features, and spatial data describe the geographic location and the geometric shape of spatial objects. Spatial objects are modeled to depict real-world elements by simple geometric forms such as points, lines and polygons. The main database spatial extensions are Oracle Spatial, PostGIS, MySQL Spatial Extension and IBM DB2 Spatial Extender.

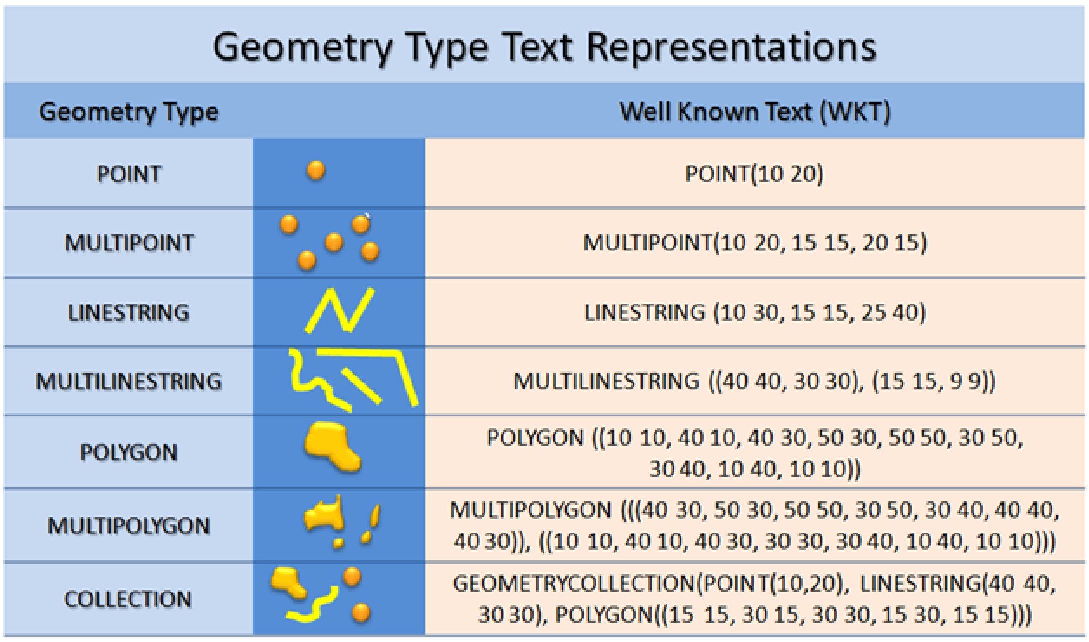
All data stored in a DBMS is ultimately in binary form. Storing geometry into a DBMS requires to define the internal format. There are currently two approaches[4].

One approach is to utilize the native way to organize, represent and store geometries. The data are built into DBMS according to their shape, such as SDO_GEOMETRY in Oracle Spatial or ST_GEOMETRY in IBM DB2 Spatial Extender. The data are still binary data, but they are organized in accordance with the expected format for geometric data by the DBMS. This data type (built into the DBMS) is called as *native geometry type.* Native Spatial DBMS also supports that data type with additional infrastructure, such as the automatic creation of spatial indexes or the provision of DBMS server commands that understand that data type.

The other approach is to utilize a generic data type, which is used at DBMSs that do not specify a pre-defined way to organize binary data to represent geometries. Almost all modern DBMS packages provide a generic binary data type to store unstructured binary data by the DBMS. This data type (not built into the DBMS) is called as *non-native geometry type.*

Although using non-native geometry types allows storing geometries within general-purpose, DBMS products without requiring a special "spatial" form of a DBMS, require a GIS application that supports the geometry formats to be used. Thereby, the OpenGIS Consortium (OGC) had defined two standard ways to express, transfer and store the same spatial-objects information at spatial databases: the Well-Known Text (WKT) form and the Well-Known Binary (WKB) form. Both WKT and WKB include information about the type of the object and the coordinates that shape the object [36]. For example, Figure 2.1 illustrates the WKT format.

[4]Extracted from: `http://www.georeference.org/doc/spatial_dbms.htm`. July, 2016



Geometry Type Text Representations

Geometry Type		Well Known Text (WKT)
POINT		POINT(10 20)
MULTIPOINT		MULTIPOINT(10 20, 15 15, 20 15)
LINESTRING		LINESTRING (10 30, 15 15, 25 40)
MULTILINESTRING		MULTILINESTRING ((40 40, 30 30), (15 15, 9 9))
POLYGON		POLYGON ((10 10, 40 10, 40 30, 50 30, 50 50, 30 50, 30 40, 10 40, 10 10))
MULTIPOLYGON		MULTIPOLYGON (((40 30, 50 30, 50 50, 30 50, 30 40, 40 40, 40 30)), ((10 10, 40 10, 40 30, 30 30, 30 40, 10 40, 10 10)))
COLLECTION		GEOMETRYCOLLECTION(POINT(10,20), LINESTRING(40 40, 30 30), POLYGON((15 15, 30 15, 30 30, 15 30, 15 15)))

Figure 2.1: WKT representations[5]

2.1.4 Data Integrity

Applying data integrity guarantees the quality in terms of consistency and accuracy of data. Therefore, integrity restrictions ensure these data properties in a relational database (i.e., ensure that data assertively represent the modeled reality). For example, if an employee is inserted with the employee ID value of 123, the database should not allow another employee to be inserted/updated with the same ID value [3, 18].

There are four categories of data integrity. *Entity integrity* specifies a table line as an exclusive entity from determined table; i.e., each line is unique within a table. It is defined through constraints like UNIQUE and PRIMARY KEY. *Domain integrity* ensures all data in a column have a pre-defined set of valid values. Examples of such constraint include: choosing the correct data type, the length for a column, and the set of possible values for an attribute. *Referential integrity* preserves the defined relationships between tables when rows are inserted or deleted. It is usually defined through FOREIGN KEY and CHECK constraints. *User integrity* allows the user to define business rules that do not fit into other categories of integrity.

[5]Available on: `https://goo.gl/bCkIUm`. Accessed May 24, 2016.

2.1.5 Spatial Integrity

Besides the regular data integrity constraints (explained in the previous section), spatial databases must also be compliant with constraints specific for the geo data. There are three may types, as follows [7].

Topological Integrity Constraints. Topology is the study of geometrical properties and spatial relations [16]. Then, this kind of constraint ensures the geographic boundaries of the objects. Modeling city neighborhoods is an example of this constraint: one neighborhood must be contained within the city limits, and there must not be any spot in the municipal territory that does not belong to some neighborhood. Logically, a neighborhood cannot belong to different cities.

Semantic Integrity Constraints. These constraints concern the meaning of geographic features. Specifically, they verify if a database state is valid due to the properties of the objects stored. An example is: a building cannot be intercepted by a street segment.

User Defined Integrity Constraints. User defined integrity constraints allow database consistency to be maintained as defined by the equivalent of "business rule" in non-spatial DBMS. This type of constraint acts, for instance, on the location of a gas station, which must lie farther than 200 meters from any existing school. The municipal permitting process must consider this limitation in its analysis.

2.1.6 Non-Relational Databases (NoSQL)

Nowadays, the term *Big Data* is a trending topic on academy and industry, with Characteristics usually shorten as "4 Vs": *Volume, Velocity, Variety and Veracity.* **Volume** describes the huge amount of storage data, **velocity** implies that the data is highly updatable, **variety** denotes that data may be stored in various formats (such as structured and semi-structured), and **veracity** portrays the uncertainty of the data due to inconsistent and incomplete data [25].

Non-relational databases are designed to scalable horizontal growth in order to support many reading and writing operations per second. All of those features contrast to traditional DBMSs that do not scale well when distributed across multiple servers. According to Cattell [10], the main features of these systems are: (i) to horizontally scale by adding new servers; (ii) to replicate and distribute across multiple servers; (iii) to provide a simple interface or access protocol (different from complex SQL languages); (iv) to be parallelizable and have a competition system weaker than transactions in relational DBMSs (with the option of reading, written and shared locks); (v) to have an efficient distribution of index and memory usage, and (vi) to have the ability to change records attributes dynamically.

There are four common architectures to handle NoSQL data: key-value, column-oriented, document-oriented and graph-oriented [27, 37].

Key-value. The key-value storage architecture is the simplest NoSQL databases model: a value assigned to a key. This method persists the data in a non-structured way (*schema-less*). Thus, data can be stored in a programming language data type or an object [37, 39]. The most popular databases that supports this storage model are the Riak[6], Redis[7], Memcached DB[8], HamsterDB[9], Amazon DynamoDB[10] and the Voldemort Project[11].

Columnar. A column-oriented NoSQL database organizes the data in a structured way and stores data from the same column continuously on the disk (as opposed to relational databases, where the rows are stored contiguously). Such systems are designed to comply with three issues: large number of columns, nature scarce data and frequent schema changes. Changing the storage project can result in better performance in some operations as aggregations, the ad-hoc supports, and dynamic query. Most columnar databases are compatible with MapReduce, which speeds up the processing of a huge amount of data, so as to distribute the problem in many systems [24, 27, 39]. The most popular open source column-oriented databases are: MonetDb[12], Hypertable[13], HBase[14] and Cassandra[15].

Document. The document term of a document-oriented database refers to the set of key-value pairs, usually in JSON (instead of documents and tables from traditional DBMSs). These documents are self-explained and have hierarchical tree structure, which can contain maps, collections and scalar values. The document-oriented database considers the document as a whole, instead of several key-value pairs. It allows documents with different structures to be grouped in the same set. The document-oriented databases support document indexes, including not only primary identifiers but document properties [21]. Typical examples of document-oriented databases include: Mon-

[6]Riak: http://basho.com/riak/
[7]Redis: http://redis.io/
[8]Memcached DB: http://memcached.org/
[9]HamsterDB: http://hamsterdb.com/
[10]Amazon DynamoDB: http://aws.amazon.com/pt/dynamodb/
[11]Voldemort Project: http://www.project-voldemort.com/voldemort/
[12]MonetDB: https://www.monetdb.org/Home
[13]Hypertable: http://hypertable.com/home/
[14]HBase: http://hbase.apache.org/
[15]Cassandra: http://cassandra.apache.org/

goDB[16], CouchDB[17] and Terrastore[18].

Graph. The graph-oriented database is another schema-less category that stores non-relational data. It consists in a set of nodes and edges: each node represents an entity (as person or company), and each edge represents a link or relations between two nodes. In a graph database, each node is defined by an unique identifier, a set of sink and/or input edges and a set of key-value pairs. Each edge is defined by a unique identifier, a source node and/or target node, and a property set. Graph databases apply graph theory to store information about the relationship between nodes. The friend relations between people in social networks is the most common example. The relationship between items and attributes in recommendation engines is another. Relational databases are not able to store data relationships between people, and the query types can be complex, slow and unpredictable [20]. The main graph databases are: Neo4j[19], HypergraphDB[20] and AllegroGraph[21].

Among the different NoSQL architectures, we chose the graph approach because geographical data can easily be seen as a graph, mainly to represent geographic networks (as road, hydrographic, electric and others). Also, graph promotes better performance in network queries as shown by Santos et al. [38]. We chose the Neo4j as physical database implementation because it has a native graph storage and a spatial plugin. Next, we briefly present Neo4j main features.

2.1.7 Neo4J

Neo4J is an open source NoSQL graph database management system developed by Neo Technology, Inc. The Neo4j is characterized by its developers as an ACID-compliant transactional database with native graph storage and processing [33]. Neo4j is the most popular graph database according to db-engines.com[22], in July 2016.

Neo4j offers different feature sets for editions of the graph database management system: the Neo4j Community Edition is the basic, fully functional, high-performance graph database, licensed under the free GNU General Public License (GPL) v3; and the Neo4j Enterprise Edition, as a more complex edition, dual licensed under Neo4j commercial license as well as under the free Affero General Public License (AGPL)

[16]MongoDB https://www.mongodb.org/
[17]CouchDB: http://couchdb.apache.org/
[18]Terrastore: https://code.google.com/p/terrastore/
[19]Neo4j: http://neo4j.com/
[20]HypergraphDB: http://www.hypergraphdb.org/index
[21]AllegroGraph: http://allegrograph.com/
[22]DB-Engines Ranking: http://db-engines.com/en/ranking

Table 2.1: Comparison Table of Neo4j Editions

Edition	**Enterprise**	**Community**
Property Graph Model	x	x
Native Graph Processing & Storage	x	x
ACID	x	x
Cypher - Graph Query Language	x	x
Language Drivers most popular languages	x	x
REST API	x	x
High-Performance Native API	x	x
HTTPS (via Plug-in)	x	x
Performance & Scalability Features		
Enterprise Lock Manager	x	-
Cache Sharding	x	-
Clustered Replication	x	-
Cypher Query Tracing	x	-
Property Existence Constraints	x	-
Hot Backups	x	-
Advanced Monitoring	x	-

v3 [33]. Table 2.1 compares the Neo4j Editions. The current Neo4j version is 3.0.1 released in May 2016.

Neo4j is implemented in Java and accessible from software written in other languages using the Cypher Query Language (CQL) through Java API or RESTful HTTP API. Cypher is Neo4j's open graph query language. Cypher is a declarative, pattern-matching query language that makes graph database management systems understandable and workable for any database user using an ascii-art syntax [33].

In Neo4j, everything is stored as an edge, a node or an attribute in order to create the graph. Each node and edge can have any number of attributes (key-value pairs). The nodes are related by the edges that intertwine creating paths in an organized manner with explicit relations. Nodes can be labeled, as the table names in relational databases. Labels can be used to narrow searches. Every edge must have a relationship type, and there are no restriction about the number of edges between two nodes. The pattern representation is inspired by traditional graph representation of circles and arrows. Vertex patterns are represented in parenthesis; and edge patterns in brackets between hyphens, one of which with a right angle bracket to indicate the edge direction. For example, the expression *(a)-[r:RELATED]->(b)* is interpreted as two vertex patterns a and b and one edge pattern r, type RELATED, that starts on vertex a and ends in vertex b [33].

2.2 Recent Work

Bugiotti et al. [8] proposed a database design methodology for NoSQL systems, based on NOAM (NoSQL Abstract Model), an innovative abstract data model to NoSQL databases that exploits the similarity of various NoSQL systems and specifies an independent representation of the application data system. The methodology aims to project a good representation of application data in a NoSQL database, to support scalability, performance and consistency of the new generation of web applications. The experiments showed that the NoSQL database design should be done carefully, because it affects the performance and consistency of data access operations and its methodology provides an effective tool for choosing between different alternatives.

Due to spatial data features, as explained in Section 2.1.1, along with the flexibility of non-relational databases, there is no geographic data modeling technique that fits all applications as a unique and ideal model because there are various implementations of NoSQL technology. NoSQL solutions can provide the necessary efficiency for applications using geographic data. Amirian et al. [1] provided a survey of the main characteristics of the huge volume of geospatial data and its possible solutions for management and treatment. They point an overview of the main types of NoSQL solutions, their advantages and disadvantages, and the challenges in managing these large volumes of data in the development of a geospatial data server using standard web geospatial services with a NoSQL XML database as a backend.

A similar context is clinical data, which are usually organized in a hierarchical form and stored as free text and number. For those, Lee et al. [29] analyzed three database models (NoSQL, XML and native XML) regarding three features: query performance, scalability and extensibility. Results showed that a NoSQL database is the best choice as to query speed, while XML is advantageous in terms of scalability, flexibility and extensibility, which are essential for dealing with the characteristics of clinical data (dynamic, sporadic and heterogeneous in essence).

Yet a different perspective is to separate the data using two different models for persistence. For example, to help decide which portion of the data of a company is persisted as XML and part as relational data, Moro et al. [32] described the ReXSA, a tool that tackles the challenge of designing hybrid database schemes. ReXSA evaluates and recommends a database schema that combines relational and XML models from a note of the data information model of a company. It has the advantage of considering qualitative properties of the information model as reuse, evolution and performance profiles to decide how to persist the data.

2.3 Considerations

Spatial data management is not easy mostly because the spatial data particularities and the many tools to treat these data. Although there is a tendency to increase the sharing of geospatial data, mainly with the help of systems that work over the Internet, little has been done to facilitate the reuse of modeling geographic database solutions in this new database paradigm, NoSQL. As explained in Section 2.1.6, NoSQL can provide an efficient way to managing these spatial data as a great solution to the increasing demand for faster geographic applications.

However, modeling non-relational database, whether for geographic data or not, is a non-trivial activity, because each application requires a modeling that fits its needs. Nevertheless, a good modeling promotes characteristics such as scalability, flexibility and extensibility and also provides schema reuse, or much of it.

There are different proposals for modeling non-conventional data, especially in XML as a NoSQL DBMS. Therefore, this dissertation advances the state of the art of NoSQL modeling by providing the primitives to map a OMT-G conceptual schema to a logical graph model schema.

Chapter 3

OMT-G

This chapter briefly overviews the OMT-G model primitives for conceptual modeling of geographic data. We explain how the modeling technique works, including the diagrams to represent spatial object classes and their relationships, and the specification of spatial integrity constraints from the conceptual schema.

3.1 Overview

OMT-G, proposed by Borges et al. [5], is an object-oriented data model for geographic applications that provides primitives to model the geometry and topology of spatial data, with support to topological structures, multiple views of objects and spatial relationships. It is an extension of UML, and therefore all conventional database components can be specified as well. OMT-G also includes tools to specify representation transformation processes, to support multiple representations and the specification of various presentation alternatives for each spatial representation.

According to Borges et al. [5], OMT-G is based on three main concepts: classes, relationships and spatial integrity constraints. Classes and relationships define the basic primitives that are used to create static schemas of applications. Spatial integrity constraints guarantee the necessary conditions to preserve database consistency.

OMT-G proposes the use of three different diagrams in the modeling of a geographic application:

- **Class Diagram:** all classes are specified along with their representations and relationships. From this diagram, it is possible to derive a set of spatial integrity constraints that must be observed in the implementation.

- **Transformation Diagram:** transformations between primary and secondary representations of real world objects are specified, allowing the identification of methods required for each transformation.

- **Presentation Diagram:** used to specify visualization options for each class, including screen display views and map printouts.

Next, a brief description of class diagram primitives (classes and relationships) is given, along with spatial integrity constraints. Transformation and presentation diagrams will not be discussed, since they are not used in the mapping to NoSQL schemas discussed in this dissertation. More details on OMT-G can be found in [5, 6].

3.2 Class Diagram

3.2.1 Class Structure

The OMT-G model uses UML class diagram primitives and introduces geographic features to increase its capacity of represent the semantics of spatial data. OMT-G includes primitives to model the geometry and topology of geographic data. Thus, it supports structures such as spatial aggregations, networks and topological relationships.

Classes and relationships are the basic primitives to create OMT-G schemas. Two types of classes are proposed by the OMT-G model: georeferenced and conventional, as shown in Figure 3.1. Conventional classes do not include geographic features and behave as UML classes. Spatial classes include a geographic representation, which can be individualized, associated to real world elements (*geo-objects*) or continuously distributed in space (*geo-fields*).

A geo-object with geometry class describes objects whose spatial representation can be abstracted as simple geometric shapes, such as points, lines, and polygons, as shown in Figure 3.2. Examples include, respectively, bus stop, curb line, and municipal limits. A geo-object with geometry and topology represents objects that have both a geometric representation and a role in a network, i.e., topological connectivity properties, such as network nodes and arcs. Such representations are specifically suited to the representation of spatial network structures, such as electrical distribution systems, hydrographic networks, or road networks. A geo-field represents variables that continuously cover the space of interest, such as soil type, temperature and relief, and can usually be seen as a surface. Geo-fields can be represented as isolines, tessellations,

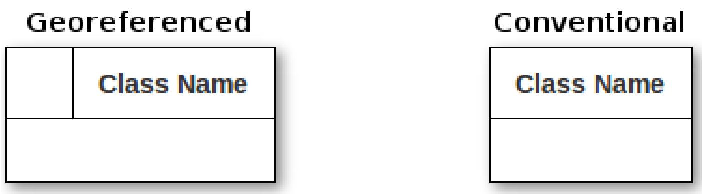


Figure 3.1: Georeferenced and Conventional classes.

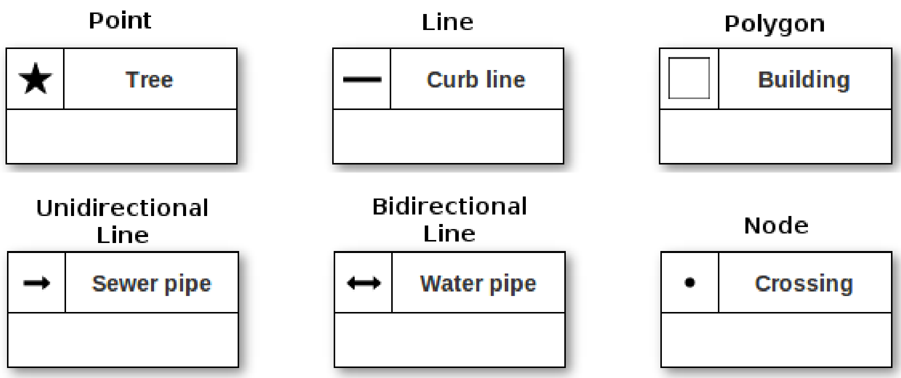


Figure 3.2: Geo-object classes.

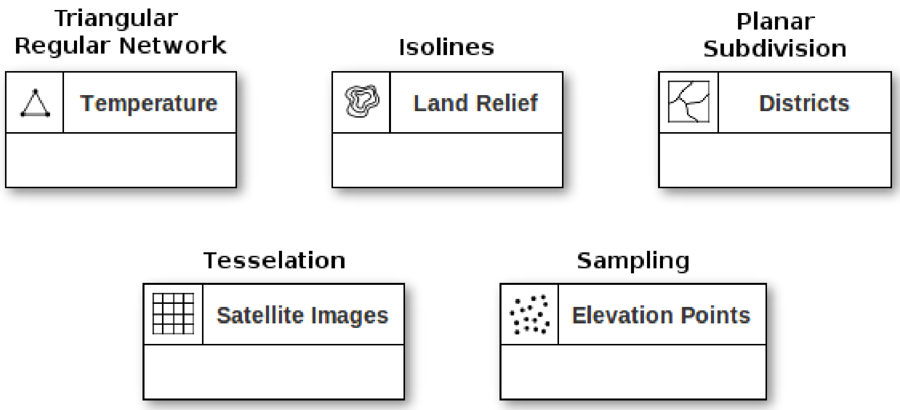
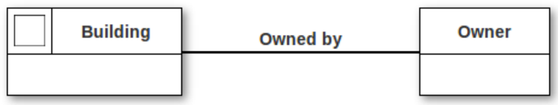


Figure 3.3: Geo-field classes.

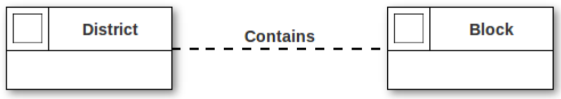
planar subdivisions, triangulated irregular networks (TIN) or sets of samples, as shown in Figure 3.3.

3.2.2 Relationships

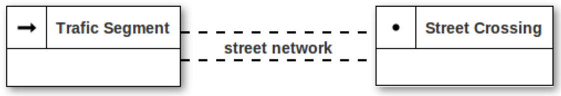
Relationships can also be simple associations, as in UML relationships, or spatial. Spatial relationships include topological relationships (such as contains, within, overlaps, and others), network relationships (arc-node networks) and spatial aggregations (whole-part aggregations, e.g. country-state). Figure 3.4 illustrates these relationships.



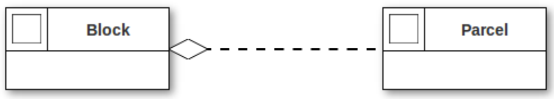
(a) Simple association



(b) Spatial relationship



(c) Arc-node relationship



(d) Spatial aggregation

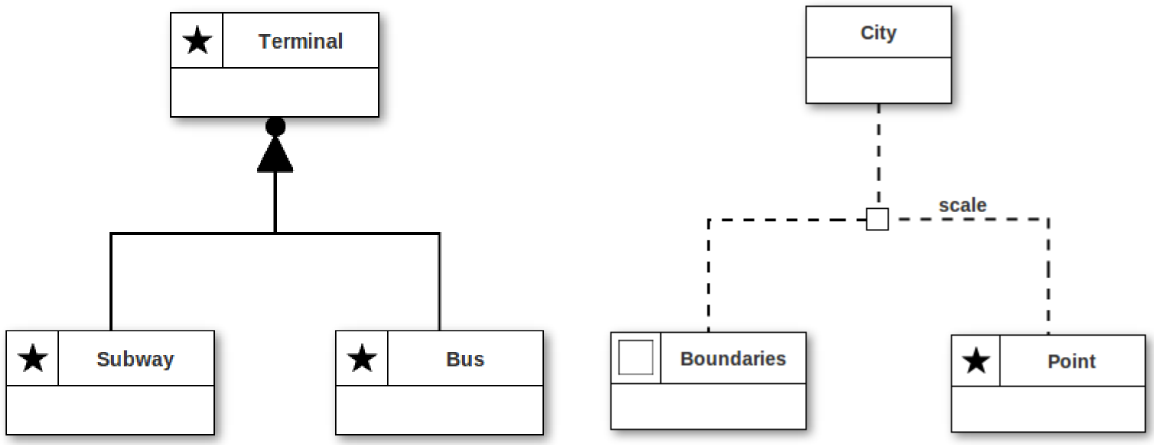
Figure 3.4: Relationships.

OMT-G also provides a primitive for generalization and specialization. Its behavior is similar to conventional object-oriented hierarchies, in which the geographic representation used for the superclass is inherited by all subclasses. The distinction between generalization and specialization depends on the nature of the classes being represented. Generalizations and specializations can be total or partial; in a total generalization, all instances must belong to at least one subclass. The can also be disjoint or overlapping; in a disjoint generalization, an instance can belong to at most one subclass.

Conceptual generalization is a primitive that allows for multiple geographic representations. In this kind of generalization, the superclass has no geographic representation, and each subclass can have a different representation. Attributes defined for the superclass are inherited by the subclasses. The geographic representation of the subclass can be defined according to its intended representation scale (or range of scales), or it can be taken as an alternative way to represent the same object in different contexts. Instances in subclasses of conceptual generalizations can be either disjoint or overlapping.

3.3 Spatial Integrity Constraints

The semantics of class and relationship primitives in OMT-G leads to the specification of several types of spatial integrity constraints. There are three kinds of such constraints: (1) geo-field constraints, (2) spatial relationship constraints, and (3) geo-object constraints. The individual constraints are formally defined next. More information about OMT-G spatial integrity constraints can be found in [5, 7, 14].



(a) Generalization (b) Conceptual generalization

Figure 3.5: Generalizations.

3.3.1 Geo-field spatial integrity constraints

Since geo-fields are expected to cover the entire space of interest, the first constraint (C1) is meant to ensure that full coverage. The other geo-field integrity constraints relate to the nature of each type of representation, ensuring that they are correct from the standpoint of the semantics of the intended representations.

C1: Planar Enforcement Rule. Let F be a geo-field and P be a point such that $P \subset F$. Then a value $V(P) = f(P, F)$, i.e., the value of F at P, can be univocally determined.

C2: Isoline. Let F be a geo-field and $\{v_o, v_1, ..., v_n\}$ be $n+1$ points in the plane. Let $a_0 = \overline{v_0v_1}, a_1 = \overline{v_1v_2}, ..., a_{n-1} = \overline{v_{n-1}v_n}$ be n segments, connecting the points. These segments form an isoline L, if, and only if, (i) the intersection of adjacent segments in L is only the extreme point shared by the segments (i.e., $a_i \cap a_{i+1} = v_{i+1}$), (ii) non-adjacent segments do not intercept (i.e., $a_i \cap a_j = \emptyset$, for all i, j such that $j \neq i+1$), and (iii) the value of F at every point P such that $P \in a_i, 0 \leq i \leq n-1$, is constant.

C3: Tesselation. Let F be a geo-field. Let $C = \{c_0, c_1, ..., c_n\}$ be a set of regularly-shaped cell covering F. C is a *tesselation* of F if and only if for any point $P \subset F$ there is exactly one corresponding cell $c_i \in C$ and, for each cell c_i the value of F is given.

C4: Planar Subdivision. Let F be a geo-field. Let $A = \{A_0, A_1, ..., A_n\}$ be a set of polygons such that $A_i \subset F$ for all i such that $0 \leq i \leq n-1$. A forms a *planar subdivision* representing F if and only if for any point $P \subset F$, there is exactly one corresponding polygon $A_i \in A$, for which a value of F is given (that is, the polygons are non-overlapping and cover F entirely).

C5: Triangular Irregular Network (TIN). Let F be a geo-field. Let $T =$

$\{T_0, T_1, ..., T_n\}$ be a set of triangles such that $T_i \subset F$ for all i such that $0 \leq i \leq n-1$. T forms a *triangular irregular network* representing F if, and only if, for any point $P \subset F$, there is exactly one corresponding triangle $T_i \in T$, and the value of F is known at all of vertices of T_i.

3.3.2 Spatial relationships integrity constraints

Primitives that specify spatial relationships between geo-object classes are expected to behave in a precise way. The following spatial integrity constraints determine the expected behavior for each spatial relationship primitive.

C6: Arc-Node Network. Let $G = N, A$ be a network structure, composed of a set of nodes $N = \{n_0, n_1, ..., n_p\}$ and a set of arcs $A = \{a_0, a_1, ..., a_q\}$. There is a relationship between members of A and N according to the following constraints: (i) for every node $n_i \in N$ there must be at least one arc $a_k \in A$; (ii) for every arc $a_k \in A$ there must be exactly two nodes $n_i, n_j \in N$.

C7: Arc-arc Let $G = \{A\}$ be a network structure, composed of a set of arcs $A = \{a_0, a_1, ..., a_q\}$. Then the following constraint applies if: (i) every arc $a_k \in A$ must be related to at least one other arc $a_i \in A$, where $k \neq i$.

C8: Spatial Aggregation. Let $P = \{p_0, p_1, ..., p_n\}$ be a set of geo-objects. Then P forms another object W by spatial aggregation if, and only if, (i) $p_i \cap W = p_i$ for all i such that $0 \leq i \leq n$, and (ii) $(W \cap \bigcup_{i=0}^{n} p_i) = W$, and (iii) $((p_i$ *touches* $p_j) \vee (p_i$ *disjoint* $p_j)) = TRUE$ for all i, j such that $i \neq j$.

Besides constraints C6, C7 and C8, OMT-G defines integrity constraints based on topological relationships, which appear in class diagrams as particular spatial relationships. While the modeler can define any kind of spatial relationship required by the application, most situations can be specified using Egenhofer's [16] 9-intersection matrix [12, 16], also adopted by OGS standards. These constraints are presented and discussed separately, in Section 3.3.4.

3.3.3 Geo-object spatial integrity constraints

The spatial representation of geo-objects as lines and polygons also implies an expected geometric configuration. A major concern is on the existence of self-intersections along polygonal lines or on the boundaries of polygons. Self-intersections cause problems for geometric algorithms, and are usually banned from the representation of geo-objects.

C9: Line. Let $v_o, v_1, ..., v_n$ be $n+1$ points in the plane. Let $a_0 = \overline{v_0 v_1}$, $a_1 = \overline{v_1 v_2}$, ..., $a_{n-1} = \overline{v_{n-1} v_n}$ be n segments, connecting the points. These segments form a *polygonal* *line* L if, and only if, (i) the intersection of adjacent segments in L is only the extreme point shared by the segments (i.e., $a_i \cap a_{i+1} = v_{i+1}$), (ii) non-adjacent segments do not intercept (that is, $a_i \cap a_j = \oslash$ for all i, j such that $j \neq i + 1$), and (iii) $v_0 \neq v_{n-1}$, that is, the polygonal line is not closed.

C10: Simple Polygon. Let $v_o, v_1, ..., v_n$ be $n+1$ points in the plane, with $n > 3$. Let $s_0 = \overline{v_0 v_1}$, $s_1 = \overline{v_1 v_2}$, ..., $s_{n-2} = \overline{v_{n-2} v_{n-1}}$ be a sequence of *n - 1* segments, connecting the points. These segments form a *simple polygon* P if, and only if, (i) the intersection of adjacent segments in P is only the extreme point shared by the segments (i.e., $s_i \cap s_{i+1} = v_{i+1}$), (ii) non-adjacent segments do not intercept (i.e., $s_i \cap s_j = \oslash$ for all i, j such that $j \neq i + 1$), and (iii) $v_0 = v_{n-1}$, that is, the polygon is closed.

C11: Polygonal Region. Let $R = \{P_0, P_1, ..., P_{n-1}\}$ be a set formed by n simple polygons in the plane, with $n > 1$. Considering P_0 to be a basic polygon, R forms a *polygonal region* if, and only if, (i) $P_i \cap P_j = /$, for all $i \neq j$, (ii) polygon P_0 has its vertices coded in a counterclockwise fashion, (iii) P_i disjoint P_j for all $P_i \neq P_0$ in which the vertices are coded counterclockwisely, and (iv) P_0 contains P_i for all $P_i \neq P_0$ in which the vertices are coded clockwisely.

3.3.4 Topological integrity constraints

The identification of topological relationships in conceptual modeling is usually done with natural language expressions, such as "contains", or "crosses". Naturally, the names that would be used by modelers to specify topological relationships in conceptual modeling can vary widely, and their interpretation by other humans can also vary. Egenhofer et al. [15, 16, 17] argues that human languages are more imprecise as to verbal definitions of spatial relationships, and therefore misinterpretation can easily occur. To that effect, they proposed a simpler model, considering the intersections between two polygons as to their interiors and boundaries, configuring a *4-intersection matrix*. Egenhofer and Franzosa [16] showed that there are only 8 viable configurations, and a name has been assigned to each of them, in an attempt for standardization. The eight relationships are called *disjoint, meet, equal, inside, contains, coveredBy, covers* and *overlap*.

However, the extension of the 4-intersection matrix to other types of geometries proved to be difficult and error-prone. Clementini et al. [11, 12, 16] proposed an extended model that has been used as the standard procedure to topologically compare two geospatial objects. It translates the relationship between two geometries into a set of outcomes based on a decision tree. An example of the result matrix, called the Dimensionally Extended nine-Intersection Model (DE-9IM) can be seen in Figure 3.6.

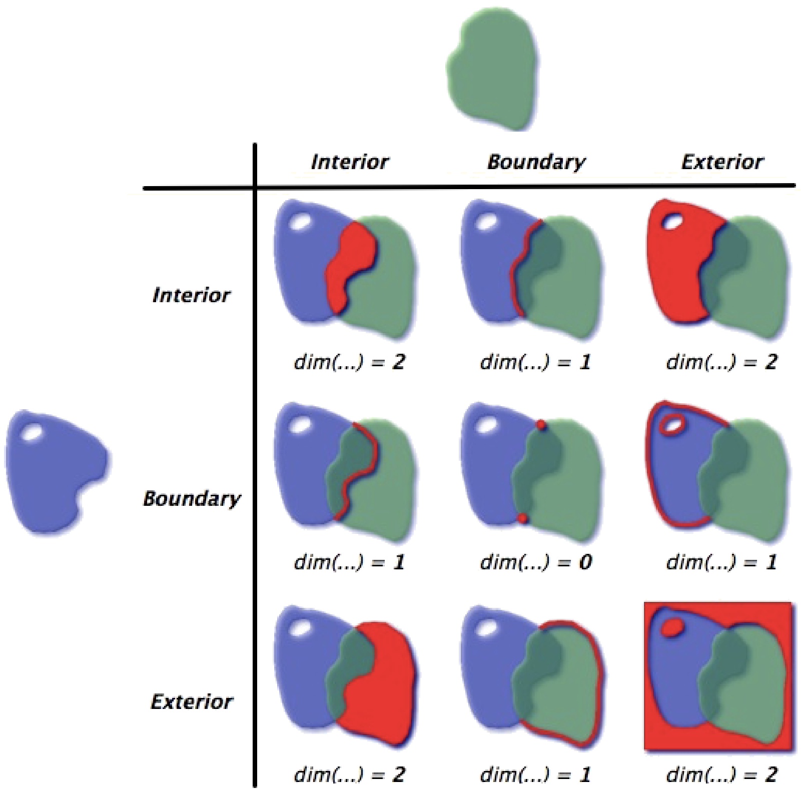


Figure 3.6: DE-9IM over spatial object interactions[1].

The intersection of the interiors is a two-dimensional area, so that matrix cell's value equals 2, indicating that the object resulting from that operation is a two-dimensional object, i.e., a polygon. When intersections are over single lines, that matrix cell's value equals 1, indicating that a one dimensional object is the result. When the polygons touch over single points, that portion of the matrix equals 0, which indicates the 0-dimensional point objects as results. When there is no intersection between components, the respective matrix value is set to a Boolean false. Likewise, when any kind of intersection is sufficient to configure a relationship, a Boolean True is used.

The DE-9IM model has been adopted by the OGC and implemented in OGC-compliant spatial database management systems, such as Oracle Spatial and PostGIS.

Spatial integrity constraints must be mapped to the physical design phase and physically implemented in geographic database management systems. For instance, a CHECK clause can use spatial functions to ensure the geometric consistency of objects represented by lines or polygons. However, ensuring the consistency of aggregations or arc-node relationships is more complicated, usually requiring the development of

[1]OpenGeo Suite: `http://suite.opengeo.org/`

triggers. Some applications for which there are performance limitations may choose not to implement the spatial integrity constraints directly in the database, an prefer creating procedures to check for inconsistencies from time to time.

Therefore, OMT-G class diagrams and the other components of the conceptual schema should contain all necessary information to ultimately be used to generate the spatial database structures, constraints and rules that must ensure the database's integrity. Borges et al. [6] present an algorithm, inspired by Elmasri and Navathe [18], that allows for the mapping between an OMT-G class diagram and an object-relational schema, which includes basic geometric representations as part of relations, along with conventional attributes. A list of conventional and spatial integrity constraints is also obtained. From the object-relational schema, a physical schema for spatially extended relational databases is easily derived, but spatial integrity constraints must be implemented using triggers, checks and assertions. Hora et al. [22] implemented an OMT-G mapping to generate Oracle physical schemas and XML schemas, including basic triggers for the first case. Lizardo et al. [31] show the creation of an online interactive modeling tool for OMT-G that includes Hora et al.'s mapping, and adds an alternative mapping algorithm to PostGIS, including spatial integrity constraints. Hora et al. [23] propose a methodology and an algorithm to map arcs and nodes, organized in a network using spatial relationships, from a OMT-G schema to a GML document. Seufitelli et al. [40] identify the challenges in mapping OMT-G primitives for NoSQL paradigms in order to integrate relational and non-relational databases, creating a hybrid approach.

In the next chapter, we present an algorithm created to map OMT-G schemas to spatially extended NoSQL database managers that implement the graph database model. As shown in Chapter 2, previous works propose modeling methodologies directly created for NoSQL, taking int consideration both the characteristics of the modeled data classes and the NoSQL manager's features and limitations. To the best of our knowledge, this is the first mapping algorithm for the mapping of spatial databases to NoSQL platforms.

Chapter 4

Mapping OMT-G to Graph Logical Schemas

This chapter introduces the procedures for mapping OMT-G to graph primitives. Such primitives include spatial representations, conventional and spatial relationships, generalizations, attributes and constraints. Section 4.1 describes the mapping steps, and Section 4.2 details an example of transformation process from a conceptual schema OMT-G to a graph logical schema. Lastly, Section 4.3 highlights some main aspects and challenges of mapping.

4.1 Mapping Steps

Mapping an OMT-G schema to a graph logical schema requires dealing with graph properties that do not exist on the traditional mapping to relational schemas. Graphs are defined by nodes connected through arcs, rather than by classes and relationships. Hence, mapping to graphs has limitations and may require additional constraints to adequately encode the semantics from the conceptual schema.

Next, we introduce the mapping steps summarized by Algorithm 1. Each step is described formally along with a discussion on limitations and implementation requirements. The formal procedures are based on the following definitions.

OMT-G Schema. Let $O = \{C, R\}$ be an OMT-G schema with $C = \{c_1, c_2, ..., c_n\}$ as a set of spatial/conventional classes and $R = \{r_1, r_2, ..., r_n\}$ as the set of their spatial/conventional relationships.

Graph Schema. Let $G = (V, E)$ be a geographic schema represented by a labeled graph where $V = \{v_1, v_2, ..., v_n\}$ is a set of nodes for spatial and non-spatial objects, and $E = \{e_1, e_2, ..., e_n\}$ is the set of their spatial or conventional relationships (edges).

Algorithm 1 General Mapping Algorithm

```
Input: OMT-G Conceptual Schema $O$
Output: Graph Logical Schema $G$
 1: Create an Empty Graph
 2: for each class $C$ in $O$ do
 3:     Create a correspondent vertex $v$
 4:     for each class attribute do
 5:         Create a correspondent attribute in $v$
 6:     end for
 7: end for
 8: for each relationship $r \in R$ between classes $c_1$ and $c_2$ in $O$ do
 9:     Create an edge $e$ between correspondent vertices $v_1$ and $v_2$
10: end for
11: for each generalization relationship $R$ in $O$ do
12:     Create a labeled edge $e$ between correspondent vertices to represent it
13: end for
```

Table 4.1: Mapping Geometry Types

OMT-G Representation		OpenGIS Representation (Simple Features Specification)
Geo-Object	Point	Point
Geo-Object	Line	LineString
Geo-Object	Polygon	Polygon
Geo-Object	Network Node	Point
Geo-Object	Unidirectional Arc	LineString
Geo-Object	Bidirectional Arc	LineString
Geo-Field	Sampling	Point
Geo-Field	Isolines	LineString and/or Polygon
Geo-Field	Planar Subdivision	Polygon
Geo-Field	Triangulation	Point (nodes) e Polygon (triangles)
Geo-Field	Tessellation	GeoRaster, long binary field

4.1.1 Step 1: Classes

Starting from any class in an OMT-G schema, map the class to a graph vertex. The geographic representation type from the OMT-G class is represented by an attribute called *geom*. The *geom* attribute is defined according to Table 4.1. Conventional OMT-G classes do not include the *geom* attribute. Figure 4.1 shows the mapping from a planar subdivision class to a vertex with a *geom* attribute of the type Polygon. The formal procedure for the class mapping is as follows.

Procedure 1. Conventional and Spatial Classes: For each class $c_i \in O$, create a vertex $v_i \in V$ (except for uni or bidirectional arc classes and Generalization/Special-

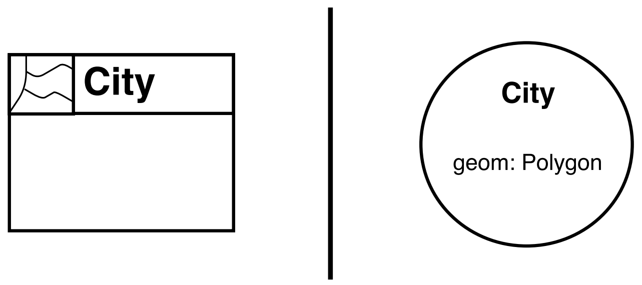


Figure 4.1: Mapping geographic classes to vertices.

ization classes). If c_i is a spatial class, create an attribute in v_i called *geom* with the geometry according to the class type, indicated by OGC representation. For Point, Node or Sampling classes, create a Point type attribute. For Line, Unidirectional Arc or Bidirectional Arc classes, create a LineString type attribute. For Polygon or Planar Subdivision classes, create a Polygon type attribute. For Isoline classes create a LineString and/or Polygon attribute. For Triangulation classes, create a Point type attribute. Triangulation vertices will contain a self-edge to interconnect the triangular irregular network. To the Tessellation classes, create a GeoRaster attribute (representing the grid cells). For conventional classes, the *geom* attribute does not exist (as it is intrinsic to geographical objects).

Limitations. This step does not consider the particular cases of Unidirectional and Bidirectional Arc classes (as they will not have a correspondent node in G). Such cases are treated by step 3. This step also does not consider Generalization/Specialization classes because they are treated in Step 4. The logical schema does not show the planar subdivision constraint, i.e., it contains no elements with which to verify whether polygons are always neighboring or distinct. Nor does the logical schema show constraints in whole-part aggregations. For instance, consider a class *City* of Planar Subdivision type and a class *District* of polygon type. Both Polygon and Planar Subdivision types are transformed to nodes with a geographic attribute of polygon type. Thereby, verifying that City is geometrically composed by the union of districts is not directly achievable by simply traversing the graph.

Implementation Requirements. Preserving the schema semantics is required to implement the corresponding integrity constraints for each geographic class layout (topological constraints). Such constraints are explained in Chapter 3.

4.1.2 Step 2: Class attributes

In this step, all class attributes and their constraints are mapped to corresponding attributes of vertices or edges. Key attributes are underlined. Attribute constraints are mapped into a JSON format (key-value pairs). The formal procedures for the class attributes mapping are as follows.

Procedure 2. Attributes: For each simple or multivalued attribute a_i of each class $c_i \in O$, create an element p_i in $v_i \in V$ that represents the c_i class according to its type.

Procedure 3. Attribute Constraints: For each attribute a_i of each class $c_i \in O$ that is a primary key, add the key attribute in v_i (underlined). In case of a foreign key type, add a keyref attribute. If a_i is multivalued, append minimum and maximum cardinality constraints, as *minOccurs* and *maxOccurs* in v_i, for example *phone: {minOccurs: 1, maxOccurs: 3}*. If there is a constraint on string length, e.g. String[15], add in e_i the attribute *maxLength* with the max string size allowed, for example *cpf: {maxLeng:11}*. If a_i has domain constraints, e.g. sex={M, F}, add in v_i the attribute *enumeration* with the values, for example *sex: {enumeration: M, F}*.

Limitations. Foreign keys are not considered in graphs, because relationships are treated directly through edges connecting vertices. NoSQL databases use key-value pairs to store their attributes. Thereby, there is a limitation on attribute constraints.

Implementation Requirements. The usage of foreign keys requires implementing referential integrity constraints. Constraints of length, domain and type requires implementing such attribute constraints to guarantee them. In the case of spatial attributes, geographic functions must be used to verify the attribute correctness. The uniqueness of vertices and edges requires implementing primary key constraints.

4.1.3 Step 3: Binary relationships and aggregations

This step maps relationships between classes to edges in the graph. Such edges relate graph vertices obtained in Step 1. Figure 4.3 shows an example of mapping relationships. Unidirectional Arc and Bidirectional Arc classes are mapped to edges that relate to Node type vertices. In graph databases, edges may store attributes as well. Thereby, geographic relationships involving OMT-G arcs and nodes are implemented using an attribute *geom* in the edge, containing the arc's geometry. Other types of edges, such as the ones used to materialize other relationships, do not have a *geom* attribute. In graphs, relationship name attributes directly indicate the nature of the relationship, because each edge between two nodes can assume only one role. For instance, Belo

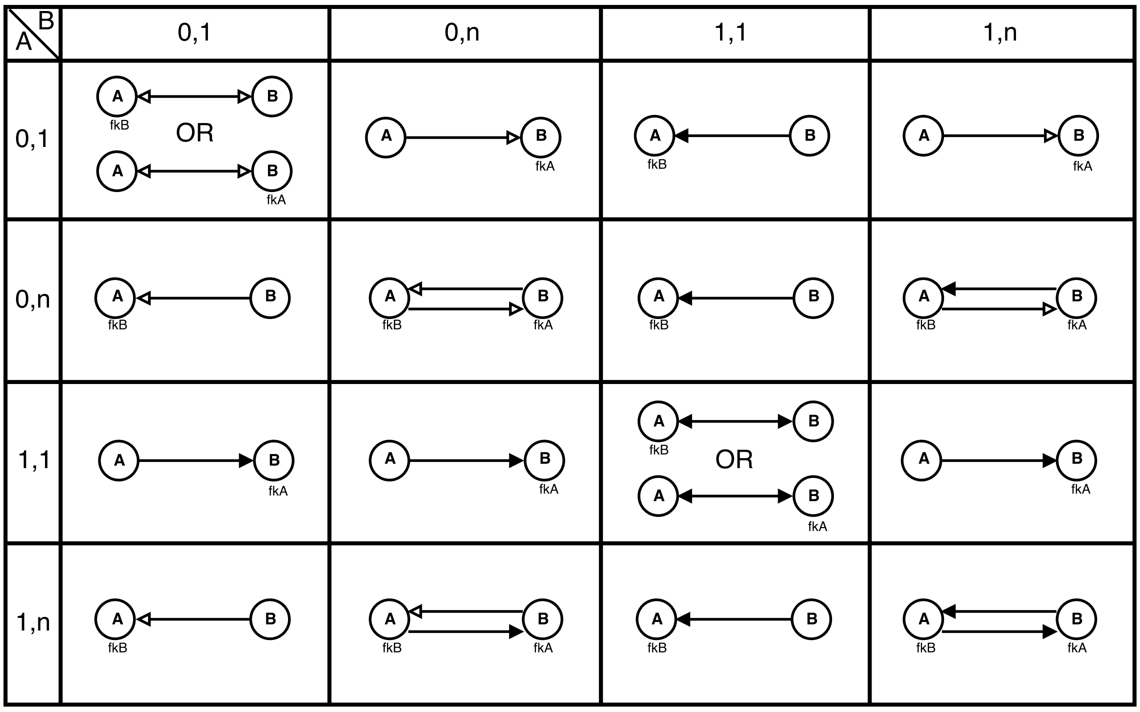


Figure 4.2: Mapping of conventional, topological and aggregation relationships

Horizonte (City instance) contains a Federal University (Building instance). The *relName* attribute of the edge that connects these two instances contains the name of the actual relationship, in this case“contains”.

The attributes *occurA* and *occurB* are created to guarantee the desired cardinality. These attributes keep occurrences of nodes A and B connected by an edge. Figure 4.2 shows a graphical notation to represent the cardinality of relationships in graphs. The arrow direction indicates the foreign key location. For example, a relationship one-to-many (1:N) between two classes A and B has a foreign key of A in B. Double arrow indicates a relationship many-to-many (N:N), and a single arrow implies a relationship at most 1 (1:N). Then, single bidirectional arrow indicates a one-to-one relationship (1:1). Finally, a filled arrow means that the foreign key cannot be null (NOT NULL), and a blank arrow that the foreign key can be null. The formal procedures for the binary relationships and aggregations mapping are as follows.

Procedure 4. Conventional and Spatial Relationships: For each simple association, topological and aggregation relationships $r_i \in R$ between classes c_i, $c_j \in C$, create an edge e_i between vertices v_i, $v_j \in V$. Such vertices represent the corresponding classes in O. For cardinalities, create an attribute *occurA* in e_i for c_i and *ocurrB* in e_i for c_j. Create an attribute *relName* in e_i to name the relationship between classes as: *contains, is part of, belongs to* and others. Aggregations (i.e. “whole-part” aggre-

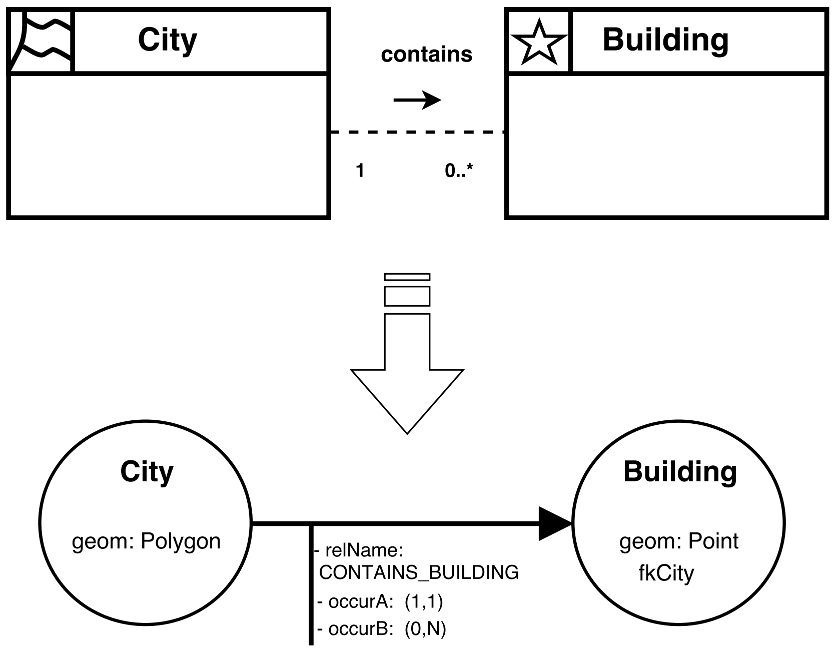


Figure 4.3: Mapping relationships to edges

gations) are handled as relationships with (1:1) cardinality in *whole-classes* and (1:N) cardinality in *part-classes.*

Procedure 5. Arc-node Relationships: For each unidirectional or bidirectional class $c_i \in C$, transform it into an edge e_i between vertices $v_i \in V$ that are vertices of Node type (point). Create a *geom* attribute with the class *geometry* in edge e_i. The flow direction of the relationship (uni or bidirectional) is denoted in the logical schema by arrows. The attribute *relName* describes the name of relationship (e.g. CONTAINS_STREET for a road network). Figure 4.4 illustrates this case. Figure 4.4 (a) shows an OMT-G schema fragment, Figure 4.4 (b) shows the proposed graph schema, and Figure 4.4 (c) shows a road network graph instance.

Procedure 6. Arc-arc Relationships: For each arc-arc relationship $r_i \in R$ of a class $c_i \in C$, where c_i is a Arc type class (bi or unidirectional), create a self-edge e_i in vertex $v_i \in V$ that represents c_i. The attribute *relName* describes the name of relationship. Create a *geom* attribute with the class *geometry* in edge e_i. The flow direction of the relationship (uni or bidirectional) is represented by arrows. Figure 4.4 presents examples of such procedure.

Limitations. Notice that arc-arc relationship is not possible to be converted into a graph, since there is no corresponding class in OMT-G that could be mapped as a graph node. Thus, the edges of an Arc type are not connected to any node.

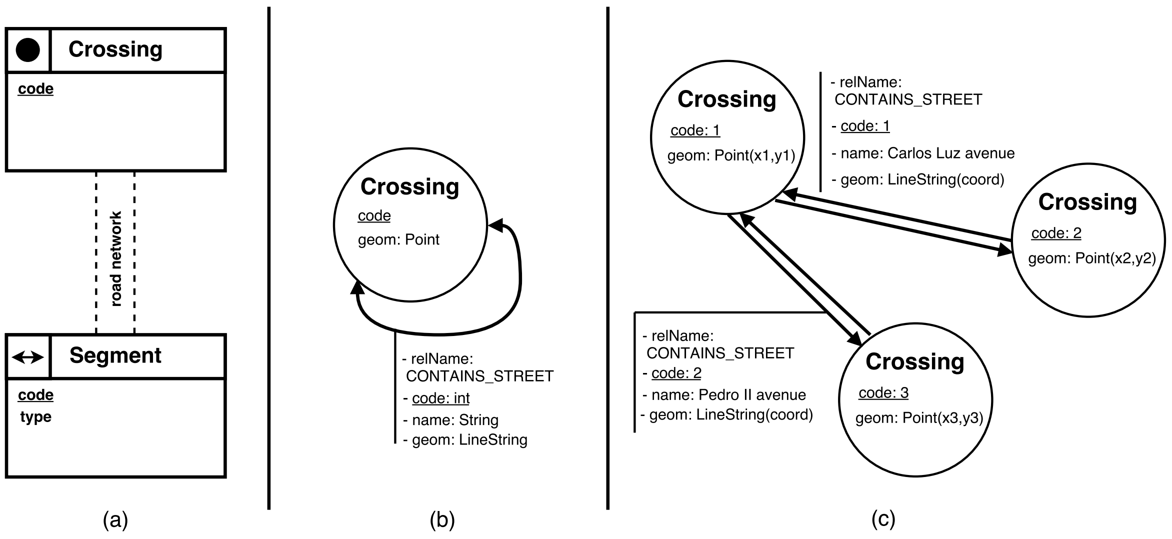


Figure 4.4: Mapping Arc-Node relationships

4.1.4 Step 4: Generalizations and specializations

This step maps all generalizations and conceptual generalizations from a conceptual schema to labeled edges. Conceptual generalization models different representations for the same geographical object in the real world. It can vary according to shape or scale. For example, a river can be perceived as the space between its margins or by the polygon occupied by water (varying in shape). Varying in scale, a school may be represented by an area (Polygon) on a larger scale and by a symbol (Point) on a smaller scale. According to total/partial and disjoint/overlapping properties of generalizations, the cardinality of the relationships between instances of the superclass and instances of the subclasses need to be constrained. The formal procedures for the generalizations and specializations mapping are as follows.

Procedure 7. Total/Disjoint Generalization: For each total/disjoint generalization $r_i \in R$, create one vertex $v_i \in V$ for each subclass in the schema (specialization classes) depicted by a double circle. To represent the superclass, these vertices must have two labels. The first label provides the superclass name, and the second one provides the subclass name. This second label is different in each vertex to represent its correspondent class in the OMT-G schema. This relationship is treated as a one-to-one (1:1) relationship. The *geom* attribute is created to explicit the vertex geometry. Subclass attributes are created in the correspondent vertices. Figure 4.5 shows this relationship.

Procedure 8. Total/Overlapping Generalization: For each total/overlapping generalization $r_i \in R$, create a vertex $v_i \in V$ for each subclass in the model (specializa-

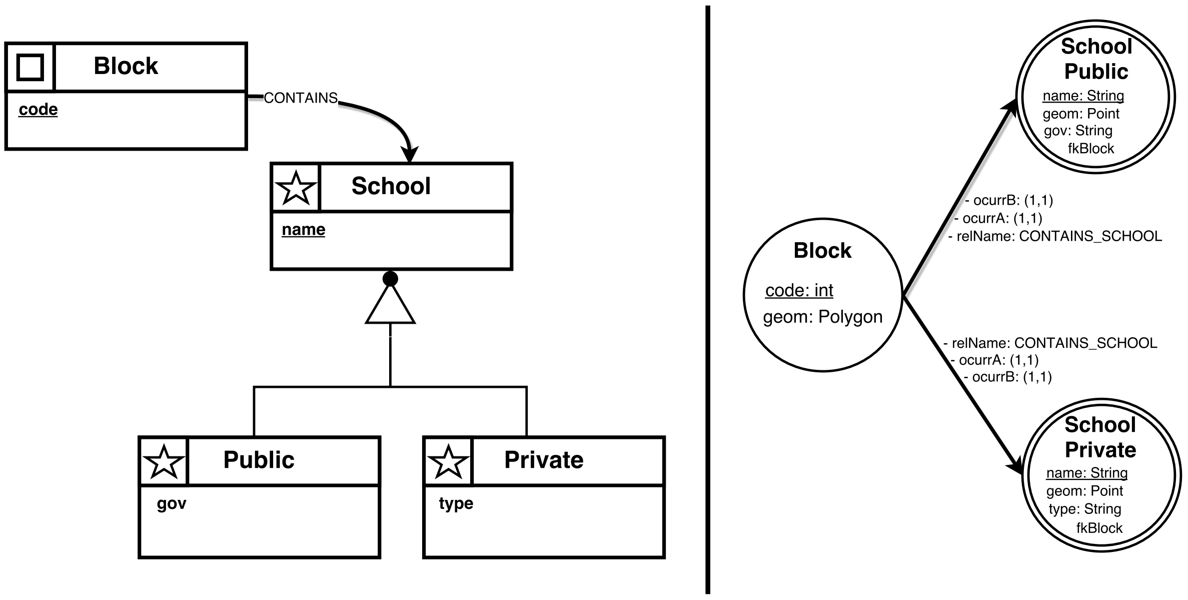


Figure 4.5: Mapping Total/Disjoint relationships

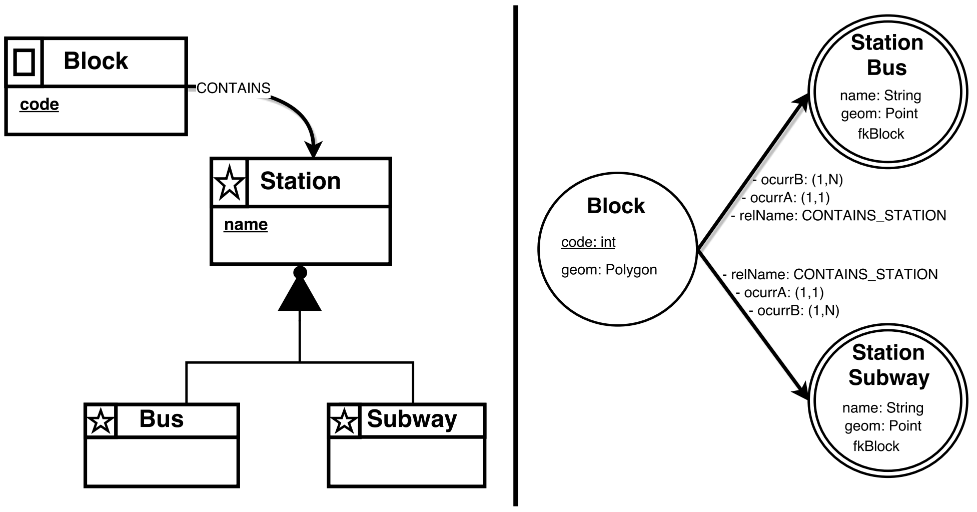


Figure 4.6: Mapping Total/Overlapping relationships

tion classes) depicted by a double circle. To represent the general class, these vertices must have two labels. The first label provides the general class name, and the second one provides the specialization class name. This second label is different in each vertex to represent its correspondent class of OMT-G schema. This relationship is treated as a one-to-many (1:N) relationship. The *geom* attribute is created to explicit the vertex geometry. Subclass attributes are created in the correspondent vertices. Figure 4.6 illustrates this relationship.

Procedure 9. Partial/Disjoint Generalization: For each partial/disjoint generalization $r_i \in R$, create a *super* type vertex $v_i \in V$ to represent the most generic class.

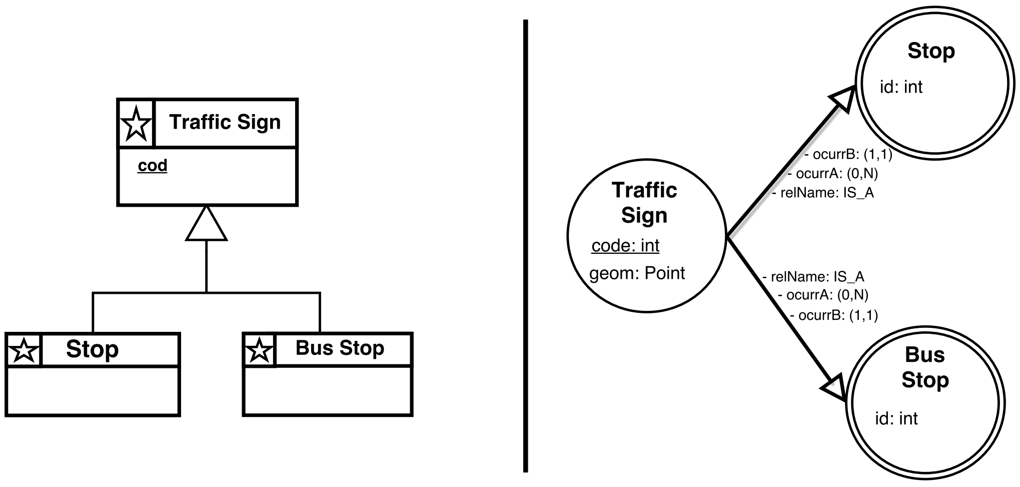


Figure 4.7: Mapping Partial/Disjoint relationships

For each subclass, create a *sub* type vertex $v_i \in V$ depicted by a double circle. This relationship is treated as a one-to-many (1:N) relationship. In the super vertex, create a *geom* attribute to specify its geometric type. The sub type vertices are no geometric. They are used to relate the geometry of the super vertex to the role specified by the sub vertex through a labeled relationship. Figure 4.7 illustrates this relationship.

Procedure 10. Partial/Overlapping Generalization: For each partial/overlapping generalization $r_i \in R$, create a *super* type vertex $v_i \in V$ to represent the most generic class. For each subclass, create a *sub* type vertex $v_i \in V$ depicted by a double circle. This relationship is treated as a zero-to-many (0:N) relationship. In the super vertex, create a *geom* attribute to specify its geometric type. The sub type vertices are not geometric. They are used to relate the geometry of the super vertex to the role specified by the sub vertex through a labeled relationship. In this case, the edges are depicted by a dotted arrow to indicate that can or cannot exist a "subvertex". Figure 4.8 shows this relationship.

Procedure 11. Overlapping Conceptual Generalization: For each overlapping conceptual generalization $r_i \in R$, create a *super* type vertex $v_i \in V$ to represent the most generic class. For each subclass, create a *subvertex* $v_i \in V$ depicted by a double circle. This relationship is treated as a zero-to-many (0:N) relationship. Here, the super vertex does not have a geometric attribute. The subtype vertices have a geometric attribute according to their geometry. Figure 4.9 illustrates this relationship.

Procedure 12. Disjoint Conceptual Generalization: For each disjoint conceptual generalization $r_i \in R$, create a *supervertex* $v_i \in V$ to represent the most generic class. For each subclass, create a *subvertex* $v_i \in V$ depicted by a double circle. This

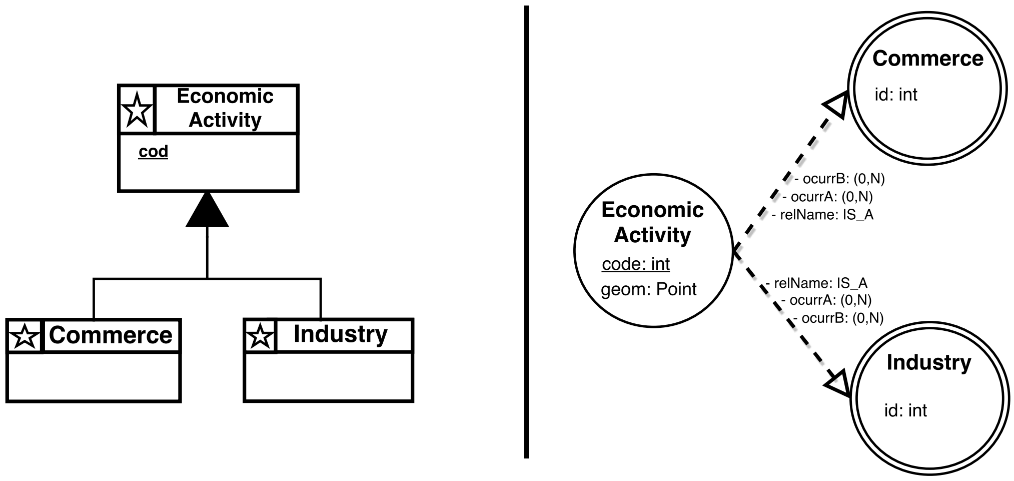


Figure 4.8: Mapping Partial/Overlaid relationships

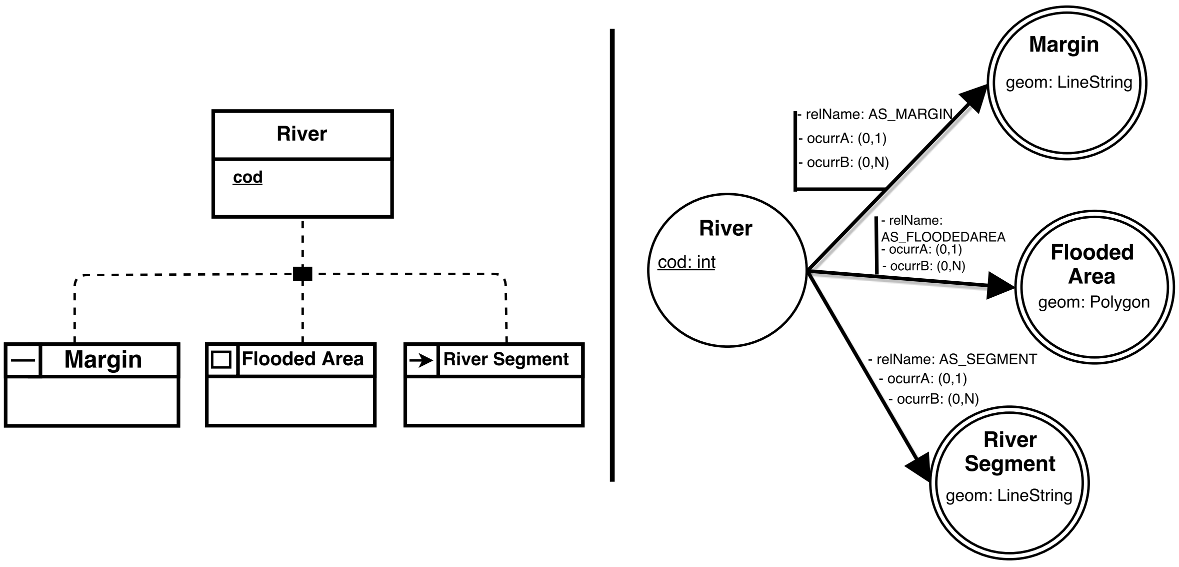


Figure 4.9: Mapping Overlapping Conceptual Generalization relationships

relationship is treated as zero-to-many (0:N) relationship. Again, the *supervertex* does not have a geometric attribute. The subvertices have a geometric attribute according to their geometry. Figure 4.10 shows this relationship.

Limitations. In graphs, the generalization/specialization relationships are characterized by the edges between vertices. Such edges are labeled to characterize the generalization/specialization. For instance, consider the following: a Block can contain a School, which can be Public or Private. In this case, there is a Total/Disjoint relationship between School and Public/Private classes, with School as a superclass and Public and Private as subclasses. Modeling it in a graph requires a labeled edge as “CONTAINS_SCHOOL” from a Block vertex instance to a School vertex instance,

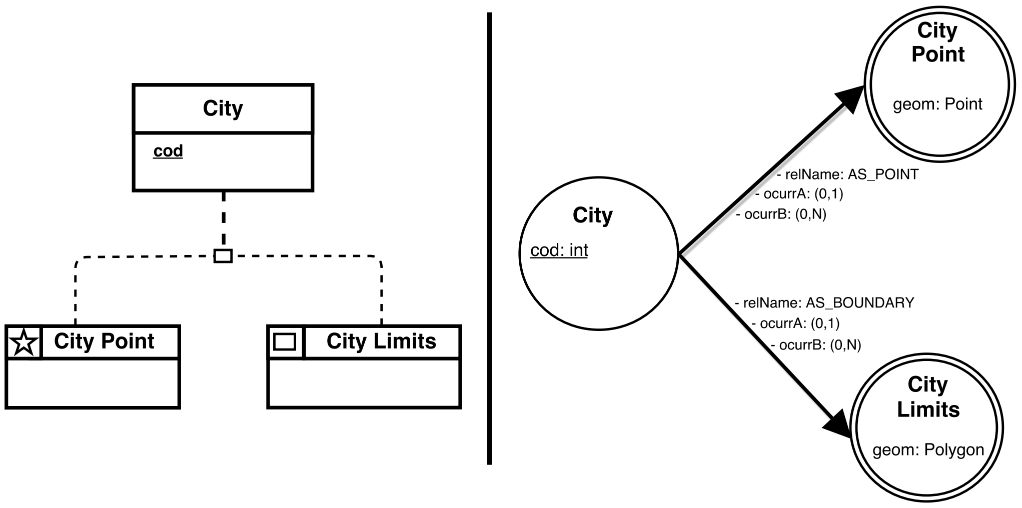


Figure 4.10: Mapping Disjoint Conceptual Generalization relationships

that is Private or Public, as in Figure 4.5. The labeled edge takes the role of the School class; thereby, the School class is suppressed in the graph schema.

Implementation Requirements. Ensuring the consistency of generalization relationships requires implementing integrity constraints. However, such constraints must consider the labeled edges assuming the role of sub or superclasses. Also, ensuring the number of edges starting from a node requires implementing integrity constraints. Such constraints are important when just one subnode type is allowed in the schema.

4.2 Example

Figure 4.11 presents an example of an OMT-G schema. It can be seen as a directed graph G, where the edge direction indicates the cardinality (i.e. (1:1)) and the nesting of the structure (i.e. (1:1) $\rightarrow$(1:N)). Algorithm 1 provides the sequence of steps to transform the conceptual geographic schema to a logical graph schema, as detailed in Section 4.1. The resulting graph schema is shown in Figure 4.12.

Each class of Figure 4.11 was mapped into a vertex of a graph, except the Building (generalization) and Segment (network bidirectional arc) classes, which are mapped in procedures 5 and 7. The resulting vertices are Region, District, Block, Building Public, Building Private, Address, Place and Crossing. These vertices are mapped according to procedure 1 of Step 1, which ensures that all conventional and spatial classes have their correspondent vertices in the resulting graph. The *geom* attribute results from mapping the geographic representation type indicated in OMT-G class primitives. For instance, the Region class is a Planar Subdivision according to OMT-G schema, and

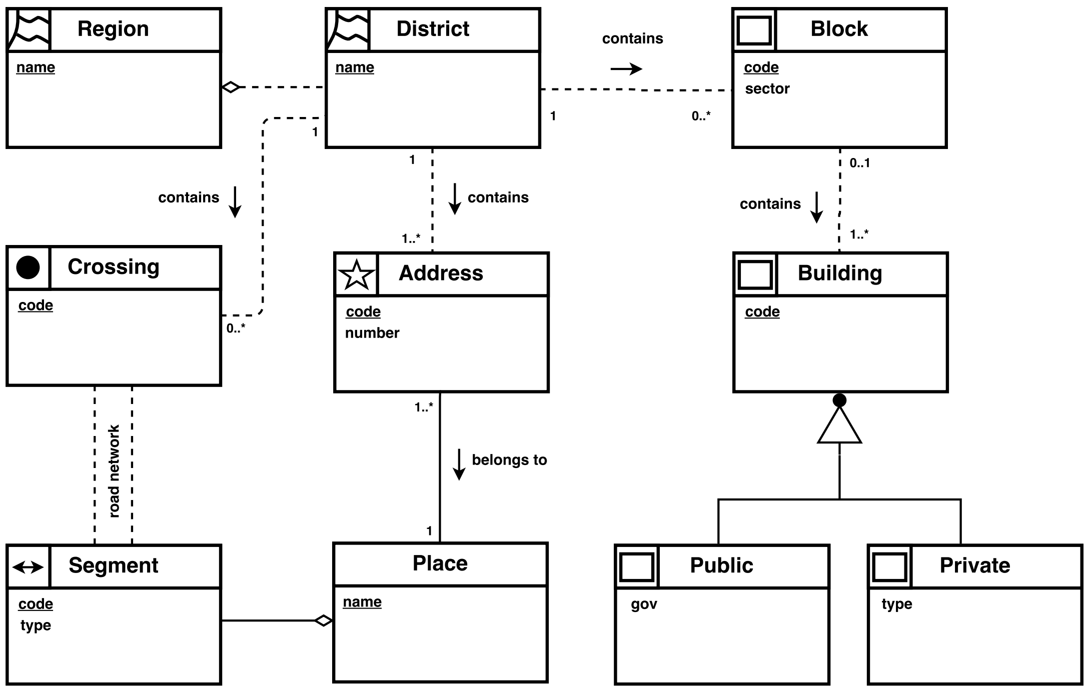


Figure 4.11: Sample OMT-G schema

its geometry type is a Polygon in the mapped vertex. Similarly, the Address class, represented by points in OMT-G, is mapped as a point to the *geom* attribute. Table 4.1 sums up this mapping step.

Attributes and their constraints are mapped to the vertex and edges according to procedures 2 and 3 of Step 2. For instance, the *number* attribute of the Address class is included in the Address vertex. A specific procedure evaluates when an attribute needs to be added into the schema to guarantee semantics. For instance, attributes *occurA: (1:1)* and *occurB: (0:N)* are created in the edge between Block vertex and Building Public and Building Private vertices. Such attributes describe the vertices' cardinalities (procedure 4 of Step 3). The attribute *relName: CONTAINS_BUILDING* describes the name of the relationship (procedure 4 of Step 3).

Relationships between vertices are mapped according to procedures 4, 5 and 6 of Step 3 and summed up in Figure 4.2. For instance, there is a unidirectional edge starting from District vertex to the Block vertex. Such edge is depicted by a filled arrow showing the cardinality of the relationship, that is 1:N. The arrow is filled because the foreign key cannot be null. In case of Building class there is a total/disjoint generalization, because an building can only be public or private. To map generalizations according to procedure 7 of Step 4, the superclass (Building) must be excluded. Such classes have a double label at the vertex corresponding to each subclass (Building, Public and Private).

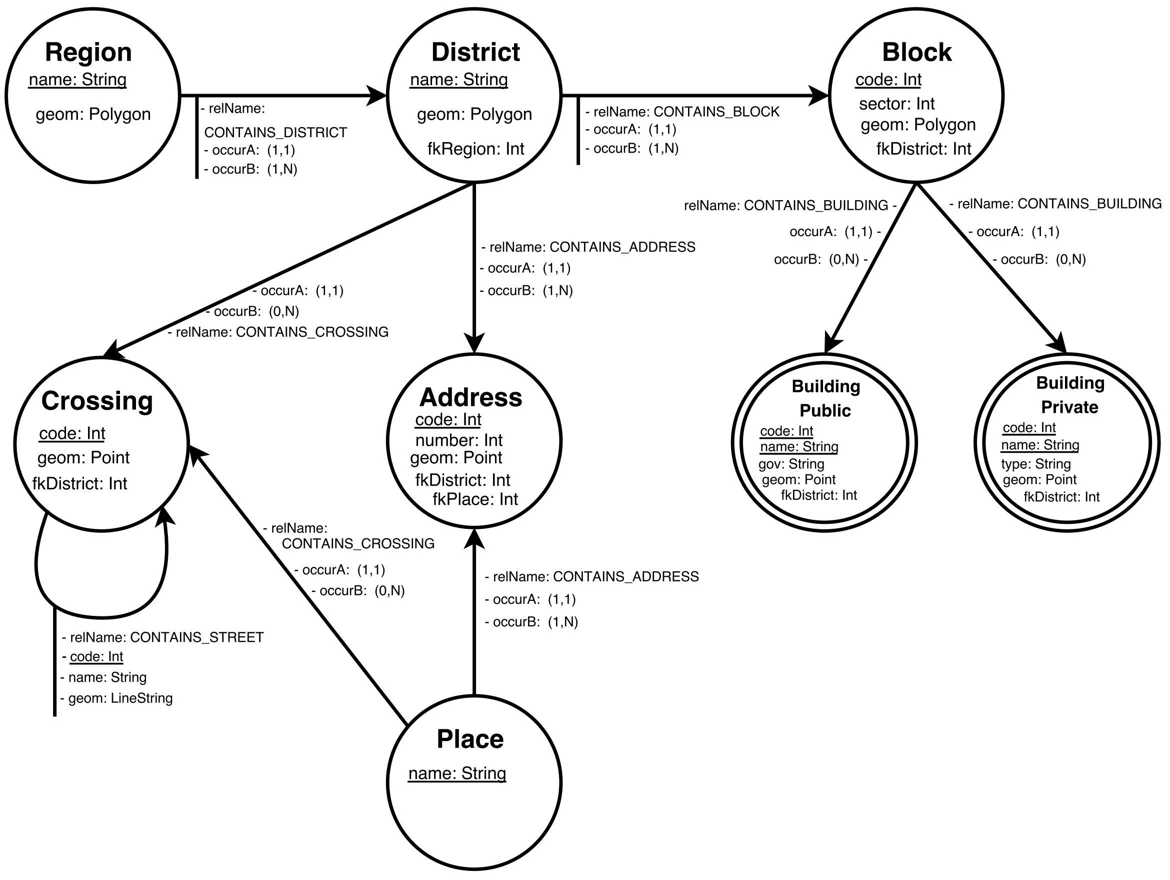


Figure 4.12: Logical Graph Schema

4.3 Considerations

In this chapter, we presented steps to perform the mapping from a geographic conceptual schema (OMT-G) to a logical graph schema. We introduced procedures and an algorithm to execute the mapping process as well as a diagrammatic way to draw a logical graph schema. All geographic primitives were considered from OMT-G (classes, relationships and attributes), including a new approach to map generalizations to graphs, which requires a direct representation through labeled edges and vertices in the graph schema.

Logical schemas are not built with the features of a specific DBMS in mind, but rather are required to generically move towards the subjacent data structures of a DBMS class, such as relational (tables), document-oriented (hierarchies), graph-oriented (graphs) and others. Some integrity constraints can be represented in the graph logical schemas and others cannot, as detailed in the steps of the mapping process. However, the implementation of these integrity constraints should be done in a specific DBMS, using its tools to enforce data consistency.

In the next chapter, we introduce the mapping process from a logical graph schema to a physical graph database using Neo4j. We explain how to create nodes and their relationships, as well as the integrity constraints that are supported by Neo4j.

Chapter 5

Physical Mapping Process

Mapping an OMT-G schema to a physical schema is similar to of conventional conceptual schemas when only non-geographic primitives are considered (e.g., Entity-Relationship). However, the semantics of spatial classes and their relationships exceed those of conventional representations, and must be preserved in the database implementation, especially spatial integrity constraints.

This chapter introduces the steps to mapping a logical graph schema to a physical graph schema, using Neo4j as a graph DBMS. As an overview, Figure 5.1 presents the whole transformation process: it starts with a conceptual geographic schema (in OMT-G) that goes through transformation rules and becomes a logical graph schema, which is mapped to the physical graph database. Next, Section 5.1 provides the steps to build the physical schema, Section 5.2 discusses the physical integrity constraints, Section 5.3 proposes a hybrid schema, highlighting its pros and cons, and Section 5.4 discusses some main aspects and challenges of such a mapping.

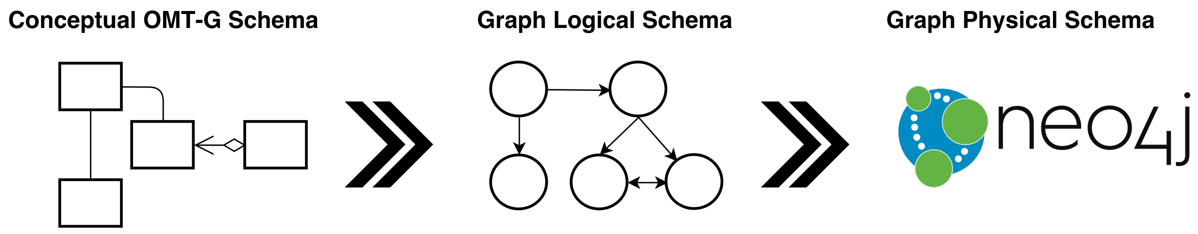


Figure 5.1: Overview of Mapping Process

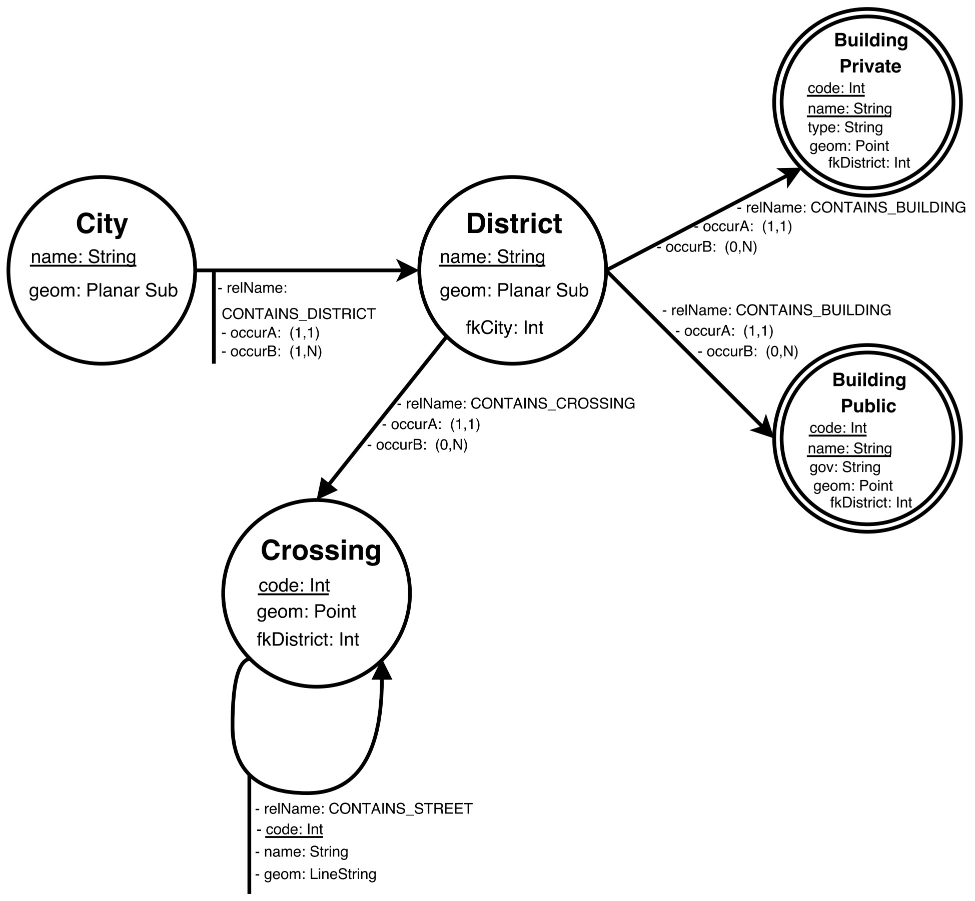


Figure 5.2: Sample Graph Schema

5.1 Mapping Steps

Mapping a geographical logical schema to a specific DBMS requires adopting the DBMSs strengths and working around its limitations. Each DBMS (relational or non-relational) has a number of peculiarities, and the limitations should be overcome according to the needs expressed in the conceptual design. Thereby, this section provides steps to map a previously modeled graph logical schema to the Neo4j DBMS. The mapping is described using the Cypher Query Language. Figure 5.2 shows the logical schema to be mapped, as a running example, which corresponds to the example discussed in Section 4.

Next, we present the mapping steps as summarized in Algorithm 2. In each step, we highlight the Neo4j limitations when they occur. The dataset in Table 5.1 must be considered as part of mapping process, as Neo4j does not have a previously materialized schema, as traditional relational databases do. Structuring and storing data into Neo4j requires inserting data in the moment of graph creation.

Algorithm 2 Physical Mapping Algorithm

```
Input:Graph Logical Schema
Output: Graph Physical Schema
1: Create a Spatial Label
2: Create Nodes and their constraints
3: Create Relationthips and their constraints
4: Associate the nodes id
5: Add nodes to layer
```

Table 5.1: Sample dataset (random data)

Id	Entity	Name	Of City/District	Coordinates
C1	City	Belo Horizonte	-	POLYGON((30 10, 40 40, 20 40, 10 20, 30 10))
D1	District	Ouro Preto	C1	POLYGON((15 20, 18 25, 15 25, 17 14, 15 40))
D2	District	Belvedere	C1	POLYGON((20 35, 25 38, 35 33, 30 30, 24 25, 20 35))
E1	Public Edification	MPMG	D2	POINT(28 35)
E2	Public Edification	PU_Belvedere	D2	POINT(26 30)
E3	Private Edification	FAMINAS	D2	POINT(23 33)
E4	Private Edification	Drugstore Araujo	D1	POINT(18 20)
E5	Public Edification	CEFET-MG	D1	POINT(20 16)
C1	Crossing	1	D1	POINT(40 50)
C2	Crossing	2	D2	POINT(45 35)
C3	Crossing	3	D2	POINT(25 20)
S1	Segment	X Street	C1, C2	LINESTRING(40 50, 42 38, 45 35)
S2	Segment	Y Street	C2	LINESTRING(25 20, 35 30, 45 35)

5.1.1 Step 1: Spatial Layer

The primary type that defines a collection of geometries is a Layer. A layer contains an index for querying. In Neo4j, the *EditableLayerImpl* is the default editable layer implementation and can handle any type of simple geometry, including Point, LineString and Polygon, as well as Multi-Point, Multi-LineString and Multi-Polygon, as standardized by the Open Geospatial Consortium (OGC). A layer is a generic implementation that ignores any topological constraints that must be observed on the data, because each geometry is stored separately in a single property of a node. The storage format is WKB, or Well Known Binary, which is an OGC standard binary format to geographic geometries that is used by most popular spatial databases.

Creating the spatial layer requires running a command (REST call) on the Neo4J web console as follows.

```
:POST <dir>/<action> {"LAYER": <layer>, "FORMAT": <format>, "NodePropertyName": <nodeProperty>}
```

where the tag <*dir*> is the graph storage directory, tag <*action*> is an action to be run, tag <*layer*> names the layer, tag <*format*> is the layer format that is going to be utilized, and tag <*nodeProperty*> specifies the node attribute that contains the geographic data. For instance, the following code creates an *EditableLayer* (tag *<action>*) called *geom* on a graph that stores geographic data with *WKT format* on attributes called *coordinates*.

```
:POST /db/data/ext/SpatialPlugin/graphdb/addEditableLayer {"layer" : "geom", "format" : "WKT", "nodePropertyName" : "coordinates"}
```

5.1.2 Step 2: Nodes

Next task is to insert nodes into the graph. In this case, each node in the logical schema becomes a node label into Neo4j's graph. Nodes in Neo4j have labels to group similar nodes. Such labels work as table identifiers in relational databases. It is possible to create nodes with multiple labels, for instance, to create node instances for generalization or specialization. Neo4j automatically inserts unique IDs in every node. Thus, there is no need to create identifiers manually. Node integrity constraints are described in more detail in Section 5.2.

As in the previous step, a REST call must be run to insert the node data as follows.

```
:POST <dir>/<action> { "STATEMENTS" : [ { "STATEMENT" : <statement>, "PARAMETERS" : <parameters>}]}
```

where the tag *<dir>* specifies the graph directory, tag *<action>* specifies an action to be executed, tag *<statement>* has the insertion Cypher statement and tag *<parameters>* contains the statement parameters. For instance, the following code creates a node to store Belo Horizonte data.

```
:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (bh:City {data}) return id(bh)", "parameters" : { "data" : { "name" : "Belo Horizonte", "coordinates" : "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"}}}]}
```

where the *<action>* is a committed transaction, the *<statement>* is the Cypher command to insert a node and the *<parameters>* are the data themselves. The commands to insert the other nodes are listed in Appendix A.

5.1.3 Step 3: Relationships

At this step, all relationships between the created nodes are established. In graphs, relationships (edges) between nodes are important to correct behavior, because graph databases (as Neo4J) mainly implement graph algorithms to traverse the graph to answer queries. Note that the integrity constraints from relationships are treated in

Section 5.2. Creating relationships between nodes previously created requires running a command as follows. Tags work as explained in Step 2.

```
:POST <dir>/<action> { "STATEMENTS" : [ { "STATEMENT" : "<statement>"}]}
```

For instance, the following code creates the relationship HAS_DISTRICT between node instances of City and District:

```
:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:City), (b:District), CREATE
(a)[:HAS_DISTRICT]>(b)"}]}
```

Similarly, the relationships between instances of District and Building must be created. However, they require retrieving the right *Building*s to associate to the right *District*, because each *Building* must be associated to a single district. Such constraint is guaranteed by the relationships (edges) between nodes. Creating the road network relationships between *Crossing* nodes (which are related by an edge labeled as “STREET”) requires retrieving the right crossing nodes to relate them with segment edges. Neo4J allows including geometries in edges to store spatial networks. To this example, there are two edges containing a *linestring* geometry to represent street segments. Lastly, the relationship “HAS_CROSSING” between *District* and *Crossing* nodes must be created. The commands to create all relationships of this example are in Appendix A.

5.1.4 Step 4: Associating the node ID

In order to enable spatial queries with Cypher, Neo4j requires that every geo-indexed node has a property called *id* with the value of the node identifier. To execute this operation, the command tags works as previously explained changing only the Cypher statement. Thus, the command to this step is as follows.

```
:POST /db/data/transaction/commit { "statements" : [ { "statement" : "match (n) set n.id=id(n)" } ]}
```

5.1.5 Step 5: Adding nodes to layer

Lastly, the nodes must be added to the spatial layer created in Step 1. Such step is required to enable the spatial queries. Then, the REST call is as follows.

```
:POST <dir>/<action> {"LAYER" : "<layer>", "NODE" : "<node>"}
```

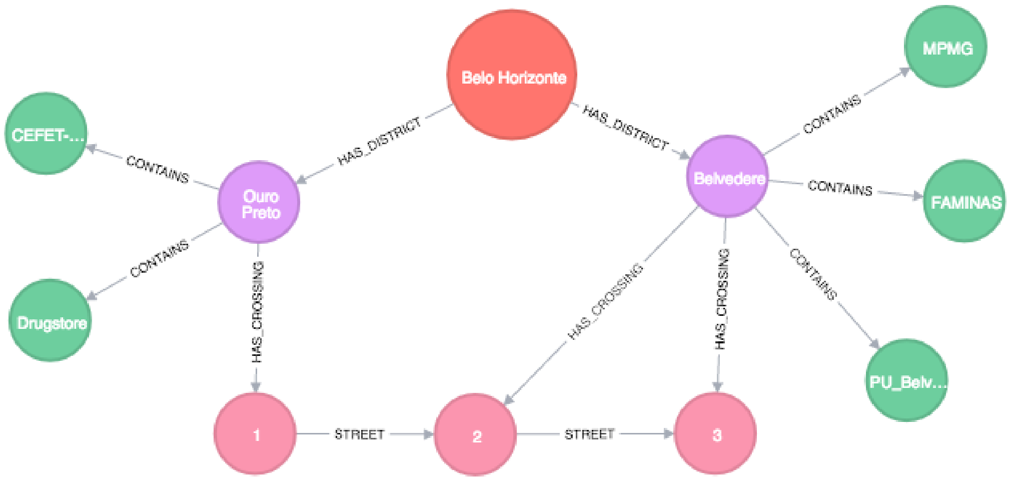


Figure 5.3: Neo4j graph schema

Where tag <*dir*> is the graph directory, tag <*action*> is the action to be done, tag <*layer*> specifies a layer (previously created) and tag <*node*> is a node (specified by its id) to be add to a layer. However, this process must be done for every single node that has a coordinate property, only changing the node id in the end of the REST call. For instance, the command to add the node with id 5 is as follows.

```
:POST /db/data/ext/SpatialPlugin/graphdb/addNodeToLayer {"layer" : "geom", "format" : "node" : "http://localhost:7474/db/data/node/5"}
```

After all of these steps, the sample schema is created in Neo4j as illustrated in Figure 5.3. In the next section, we discuss the physical integrity constraints in Neo4j.

5.2 Physical Integrity Constraints

The consistency of data are ensured by integrity constraints. As shown in Chapter 2, the integrity constraints are: entity, domain, referential, topological, semantic and user-defined. However, Neo4j does not adopt a previous schema and does not have the same concept of triggers and procedures as in relational databases. Thus, certain integrity restrictions might not be verified. Next, we highlight the main aspects of physical integrity constraints.

Entity Integrity Constraint. In Neo4j, each created node has a unique identifier. Thereby, it is possible to create several nodes with the same attributes. However, Neo4j has the unique property constraints to ensure that property values are unique for all nodes with a specific label. Unique constraints do not mean that all nodes must have a unique value for the properties. However, nodes without the property are not subject to this rule. Such constraint is defined as follows:

```
CREATE CONSTRAINT ON (<alias><nodeLabel>) ASSERT <alias>.<nodeProperty> IS UNIQUE
```

Where tag <alias> is an alias for node instances, tag <nodeLabel> is the label from nodes that participate of such constraint, and tag <nodeProperty> is the specific property to be guaranteed. Such constraint works as primary key constraints in relational databases. For instance, the following code applies such constraint to the sample schema on *name* property of *City, District and Building* node labels.

```
CREATE CONSTRAINT ON (c:City) ASSERT c.name IS UNIQUE

CREATE CONSTRAINT ON (d:District) ASSERT d.name IS UNIQUE

CREATE CONSTRAINT ON (e:Building) ASSERT e.name IS UNIQUE
```

As Neo4j creates a unique identifier automatically, there is no need to create constraint *unique* to the *code* property. This constraint ensures that a *Building* cannot be *Private* and *Public* at the same time, for example.

Constraints can be added after a label is already in use. However, it requires that existing data complies with such constraints. Otherwise, the constraints will not be created. When adding unique constraints, Neo4j adds an index on that property implicitly. Thus, there is no need to do that separately. If a constraint is dropped but still has an index on such property, an index must to created explicitly.

Neo4j has a property existence constraint to ensure that a property exists for all nodes with a specific label, or for all relationships with a specific type. However, the property existence constraints are only available in the Neo4j Enterprise Edition. For example, the following code applies such restriction on the City node.

```
CREATE CONSTRAINT ON (c:City) ASSERT exists(c.Name)
```

The property existence constraint can also be applied in relationships. Creating a constraint to guarantee that all relationships with a certain type have a certain property requires the syntax *ASSERT exists(variable.propertyName)* in moment of constraint creation.

Domain Integrity Constraint. In Neo4j, properties are key-value pairs where the key is a string. Property values can be either a primitive or an array of one primitive type: boolean, byte, short, int, long, float, double, char or string. However, there is no way to validate the input data. If this restriction must be used, it must be of application responsibility. Null values are not allowed in Neo4j. If a property has null values, such property must be removed.

Referential Integrity Constraint. Relationships between nodes are a key point of graph databases as they allow finding related data. In Neo4j, a relationship connects exactly two nodes, ensuring a valid start node and an end node. Neo4j does not allow to remove nodes that have connections with other nodes. Removing a node requires to delete their relationships first.

Neo4j does not require the use of primary and foreign keys. The relationships are defined through labeled edges. Relationships are equally well traversed in either direction. Hence, there is no need to add duplicate relationships in the opposite direction (e.g., to improve traversal or performance).

Topological Integrity Constraint. The topological constraints ensure consistency of data regarding their topology. Neo4j offers topological functions for verifying and querying spatial data: *Contain, Cover, Covered By, Cross, Disjoint, Intersect, Intersect Window, Overlap, Touch, Within* and *Within Distance*. However, Neo4j does not have the concept of triggers (as in relational databases) to keep data consistency. Thereby, using such restriction in Neo4j requires extending Neo4j through plugins to solve the topological integrity constraints.

Semantic Integrity Constraint. The semantic of geographic data must also be verified through topological functions. For example, a building cannot be intercepted by a street segment. Thereby, a routine must be implemented to check and validate the data. Such routine (trigger) demands to extend Neo4j through plugins.

User-defined Integrity Constraint. The user-defined constraints act as business rule of a schema. For instance, a gas station must lie farther than 200 meters from any existing school. However, as the other integrity constraints, the only way to validate the user-defined constraints is by extending the Neo4j through plugins.

NoSQL graph databases are designed to be flexible, without concerning about integrity constraints. Such property can be seen in Neo4j, which requires extending its functionalities by building plugins to use the most of integrity constraints (geographical or not). Hence, using integrity constraints in Neo4j becomes complex. However, Neo4j can be used in a hybrid solution combining its advantages (such as explicit network relationships) with the advatantages of relational model (such as the implemented integrity constraints). A hybrid solution can improve the quality of geographical data storing. Next, we discuss a hybrid approach to NoSQL and SQL databases.

5.3 Hybrid Schema

NoSQL databases do not solve all problems and do not replace traditional relational databases. NoSQL databases goal is very specific: filling gaps left by the relational model (such as sharding and a better performance on a huge amount of data). It is also an alternative way to provide persistence, which may be more suitable for specific projects, i.e., not necessarily to all projects. The *one-size-fits-all* no longer exists.

Each database paradigm (relational or NoSQL) is designed to get better performance in distict scenarios (e.g., operations, data structures and data volume). For instance, relational databases ensure the integrity of structured data; key-value databases allow a quick data access using a key (as hash table); document databases store nested data (as JSON); graph databases store data in which their topology and relationships are important (as network relationships); and columnar databases store data in columns (as opposed to relations). However, implementing a hybrid approach can combine benefits of each database paradigm, such as relational and non-relational. Such approach may be called *polyglot persistence* [35].

In case of geographical data, a hybrid approach can improve the data management. Specifically, Santos et al. [38] show that graph databases (Neo4j) get better performance for network queries (as *Urban Routing* and *Position Tracing*) in comparison to relational databases (PostGIS). However, as previously discussed in Section 5.2, making Neo4j to ensure the integrity constraints of spatial data is a hard task, because Neo4j does not have sufficient tools. On the other hand, relational solutions for spatial data are more mature to treat integrity constraints.

In this case, a solution that combines the advantages of graph databases (network relationships) with the advantages of relational databases (data integrity) can make a geographical system faster. For instance, network queries are sent to the graph database and the others to the relational one, and data integrity is verified by the relational database. The aspects of such solution is summarized in Table 5.2 with their pro and cons.

Table 5.2: Comparison between SQL, NoSQL and Hybrid solutions

Solution	**Pro**	**Cons**
Relational	More mature	Lower performance*
	More used	Complex to network relationships
	Spatial integrity constraints	Not ideal to cluster
	ACID	Model-dependent
Graph	Explicit network relationships	Spatial integrity constraints
	Higher performance on network relationships	Less used
	Ideal to cluster	Schemaless
	More flexible	ACID/CAP
Hybrid Approach	Data consistent	
	Higher performance on network relationships	
	Spatial integrity constraints	

5.4 Considerations

In this chapter, we presented the mapping from a geographic conceptual schema (OMT-G) to a physical graph schema using the Neo4j's Cypher query language. The mapping process started from a sample logical graph schema to the physical schema in Neo4j.

The resulting graph of the case study is well connected. However, having a disconnected graph (i.e., a set of smaller graphs instead of only one) is also a possibility, and Neo4j can handle it easily. In this case, a disconnected graph works as disconnected relations (tables) in relational databases. Specifically, Neo4j spatial functions are able to query disconnected spatial nodes that contain geometries. For instance, the spatial function *Contain* retrieves all nodes with a point geometry in a delimited area. However, having disconnected graphs to store spatial data may jeopardize using graph algorithms to traverse the whole graph.

Unlike conceptual schemas, physical schemas depend on a specific DBMS implementation, which may result in challenges for storing and managing data. For instance, integrity constraints on Neo4j require constructing plugins to extend its functionalities, because Neo4j does not have triggers (as relational databases do). Hence, a hybrid approach can be utilized to combine advantages of both relational and graph models.

A previous work of our research group has appointed advantages of NoSQL databases to spatial data [38]. Specifically, Neo4j had higher performance in network queries. On the other hand, other types of spatial queries got better performance when running on SQL solutions (e.g., PostGIS) or even in other NoSQL approaches, as document-oriented databases (MongoDB). Thus, an integration between a schemaless model and a structured schema can provide many benefits, as shown in Section 5.3. However, such polyglot persistence requires a deep research to cover all restrictions and limitations of both models, SQL and NoSQL.

Chapter 6

Conclusion

This work has introduced a methodology to map geographical conceptual schemas (OMT-G) to graph logical schemas. Given a conceptual model, the hardest part of mapping process is to define which class is mapped to a node and how to convert relationships to edges. Another challenge is identifying how to keep the model semantic, respecting both the spatial integrity constraints and non-spatial constraints. Specifically, we presented an algorithm to map an OMT-G schema to a logical graph schema. Such algorithm has a sequence of steps with a formal procedure to reach the mapping as well as the implementation requirements that highlights key aspects to ensure the integrity constraints.

This work has also presented an algorithm to map a logical graph schema to a physical graph schema. To demonstrate such step, Neo4j was chosen as a physical graph database. A logical schema does not depend on a specific database implementation, it depends only on the database type (e.g. relational, graph, document, etc). However, when mapping a logical schema to a specific database, it means adopting the its strengths and limitations. In case the Neo4j, we highlighted the positive aspects as the facility to create network relationships by labeled edges, and the negative aspects as the absence of mechanisms to verify integrity constraints. The initial results of this dissertation are published at [40].

Although Neo4j has a series of limitations to handle spatial data, its use as part of storing solution is not ruled out. Thus, we started a discussion of a hybrid approach considering Neo4j for manage network relationships. The positive and negative points in adopting a hybrid schema are presented, to help a database designers decide which the better storage solution. However, a further study is required to detail the integration process between the NoSQL and SQL architecture.

Ideas for improving and extending this work include:

- **Extend this study to other NoSQL approaches.** Proposes methodologies to mapping geographical conceptual schemas to other NoSQL approaches as document-oriented and column-oriented.
- **Study spatial performance metrics by different modeling techniques.** There are many possibilities to model a NoSQL approaches. However, we need to test if there are performance rise by using different modelling strategies.
- **Develop a Hybrid Spatial DBMS that encapsulates multiple data models/mapping strategies.** We note that systems focused on a single model (for instance, the network model and the urban routing category) usually excel in a particular scenario, but have a underperforming on the rest. Thus, a hybrid approach using NoSQL and SQL solutions may be used to get better performance in different spatial queries.

Bibliography

[1] Amirian, P., Basiri, A., and Winstanley, A. (2013). Efficient Online Sharing of Geospatial Big Data Using NoSQL XML Databases. In *2013 Fourth International Conference on Computing for Geospatial Research and Application (COM. Geo)*, pages 152--152. IEEE.

[2] Angles, R. and Gutierrez, C. (2008). Survey of Graph Database Models. *ACM Computing Surveys (CSUR)*, 40(1):1.

[3] Beard, B. (2016). *Practical Maintenance Plans in SQL Server: Automation for the DBA*, chapter Checking Database Integrity, pages 33--43. Apress, Berkeley, CA.

[4] Boaventura Filho, W., Olivera, H. V., Holanda, M., and Favacho, A. A. (2015). Geographic data modeling for NoSQL document-oriented databases. *The Seventh International Conference on Advanced Geographic Information Systems, Applications, and Services - GEOProcessing 2015*, page 72.

[5] Borges, K. A. V., Davis, C. A., and Laender, A. H. (2001). OMT-G: An Object-Oriented Data Model for Geographic Applications. *GeoInformatica*, 5(3):221--260.

[6] Borges, K. A. V., Davis Jr., C. A., and Laender, A. H. F. (2005). Modelagem conceitual de dados geográficos. In Casanova, M., Câmara, G., Davis Jr., C. A., Vinhas, L., and Ribeiro, G., editors, *Bancos de Dados Geográficos*, pages 83–136. MundoGeo Editora.

[7] Borges, K. A. V., Laender, A. H. F., and Davis, Jr., C. A. (1999). Spatial Data Integrity Constraints in Object Oriented Geographic Data Modeling. In *Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems*, pages 1--6, Kansas City, Missouri, USA.

[8] Bugiotti, F., Cabibbo, L., Atzeni, P., and Torlone, R. (2014). Database Design for NoSQL Systems. In *International Conference on Conceptual Modeling.*, volume 8824, pages 223--231, Atlanta, USA.

[9] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2016). A Data Management Middleware for ITS Services in Smart Cities. *Journal of Universal Computer Science*, 22(2):228--246.

[10] Cattell, R. (2011). Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12--27.

[11] Clementini, E. and Di Felice, P. (1996). A Model for Representing Topological Relationships between Complex Geometric Features in Spatial Databases. *Information sciences*, 90(1):121--136.

[12] Clementini, E., Di Felice, P., and van Oosterom, P. (1993). A Small Set of Formal Topological Relationships Suitable for End-User Interaction. In *3rd International Symposium on Spatial Databases*, pages 277--295, National University of Singapore.

[13] Câmara, G. (1996). Caracterização de Dados Geográficos. In Câmara, G., Casanova, M. A., Hemerly, A. S., Magalhães, G. C., and Medeiros, C. M. B., editors, *Anatomia de Sistemas de Informação Geográfica*, pages 37–48. Instituto de Computação - UNICAMP.

[14] Davis Jr., C. A., Borges, K. A. V., and Laender, A. H. F. (2005). Deriving Spatial Integrity Constraints from Geographic Application Schemas. In Rivero, L. C., Doorn, J. H., and Ferraggine, V. E., editors, *Encyclopedia of Database Technologies and Applications*, pages 176–183. Idea Group.

[15] Egenhofer, M. J. (1989). A Formal Definition of Binary Topological Relationships. In *Third International Conference on Foundations of Data Organization and Algorithms*, pages 457--472, Paris, France.

[16] Egenhofer, M. J. and Franzosa, R. D. (1991). Point-Set Topological Spatial Relations. *International Journal of Geographical Information System*, 5(2):161--174.

[17] Egenhofer, M. J. and Herring, J. (1990). A Mathematical Framework for the Definition of Topological Relationships. In *Fourth international symposium on spatial data handling*, pages 803--813, Zurich, Switzerland.

[18] Elmasri, R. and Navathe, S. (2009). *Fundamentals of Database Systems*. Addison-Wesley, Menlo Park, CA, 6th edition.

[19] Güting, R. H. (1994). GraphDB: Modeling and Querying Graphs in Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 297--308, Santiago de Chile, Chile.

[20] Hashem, H. and Ranc, D. (2015). An Integrative Modeling of BigData Processing. *International Journal of Computer Science and Applications*, 12(1):1–15.

[21] He, C. (2015). Survey on NoSQL Database Technology. *Journal of Applied Science and Engineering Innovation Vol*, 2(2):50--54.

[22] Hora, A. C., Davis Jr, C. A., and Moro, M. M. (2010). Generating XML/GML Schemas from Geographic Conceptual Schemas. In *IV Alberto Mendelzon Workshop on Foundations of Data Management*, Buenos Aires, Argentina.

[23] Hora, A. C., Davis Jr, C. A., and Moro, M. M. (2011). Mapping Network Relationships from Spatial Database Schemas to GML Documents. *Journal of Information and Data Management*, pages 67–74.

[24] Hu, Y. and Dessloch, S. (2014). Defining Temporal Operators for Column Oriented NoSQL Databases. In *18th East-European Conference on Advances in Databases and Information Systems*, pages 39–55, Ohrid, Republic of Macedonia.

[25] Jagadish, H. V., Gehrke, J., Labrinidis, A., Papakonstantinou, Y., Patel, J. M., Ramakrishnan, R., and Shahabi, C. (2014). Big Data and Its Technical Challenges. *Communications of the ACM*, 57(7):86--94.

[26] Junglas, I. A. and Watson, R. T. (2008). Location-Based Services. *Communications of the ACM*, 51(3):65--69.

[27] Kaur, K. and Rani, R. (2013). Modeling and Querying Data in NoSQL Databases. In *2013 IEEE International Conference on Big Data*, pages 1--7. IEEE.

[28] Laurini, R. and Thompson, D. (1992). *Fundamentals of Spatial Information Systems*. Academic Press.

[29] Lee, K. K.-Y., Tang, W.-C., and Choi, K.-S. (2013). Alternatives to Relational Database: Comparison of NoSQL and XML Approaches for Clinical Data Storage. *Computer methods and programs in biomedicine*, 110(1):99--109.

[30] Lisboa F., J. and Iochpe, C. (1999). Specifying Analysis Patterns for Geographic Databases on the Basis of a Conceptual Framework. In *Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems*, pages 7--13, Kansas City, Missouri, USA.

[31] Lizardo, L. E. O. and Davis, C. A. (2014). *OMT-G Designer: A Web Tool for Modeling Geographic Databases in OMT-G*, pages 228--233. Springer International Publishing.

[32] Moro, M. M., Lim, L., and Chang, Y.-C. (2007). Schema Advisor for Hybrid Relational-XML DBMS. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 959--970, Beijing, China.

[33] Neo4j (2016). Neo4j Graph Database: Unlock the Value of Data Relationships. Available at: `http://neo4j.com/product/`. Accessed in June 06, 2016.

[34] Newman, M. E. (2003). The Structure and Function of Complex Networks. *SIAM review*, 45(2):167--256.

[35] Oliveira, F. R. and del Val Cura, L. (2016). Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 230--235, New York, NY, USA.

[36] POSTGIS (2016). Using PostGIS: Data Management and Queries. Available at: `http://postgis.net/docs/using_postgis_dbmanagement.html`. Accessed in May 31, 2016.

[37] Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education.

[38] Santos, P. O., Moro, M. M., and Jr., C. A. D. (2015). Comparative performance evaluation of relational and nosql databases for spatial and mobile applications. In *Proceedings of 26th International Conference Database and Expert Systems Applications - DEXA*, pages 186--200, Valencia, Spain.

[39] Saxena, M., Ali, Z., and Singh, V. K. (2014). NoSQL Databases- Analysis, Techniques, and Classification. *Journal of Advanced Database Management & Systems*, 1(2):13--24.

[40] Seufitelli, D. B., Moro, M. M., and Davis Jr, C. A. (2015). Desafios no Mapeamento de Esquemas Conceituais Geográficos para Esquemas Físicos Híbridos SQL/NoSQL. In *GeoInfo*, pages 119--124, Campos do Jordão, São Paulo.

[41] Shekhar, S., Coyle, M., Goyal, B., Liu, D.-R., and Sarkar, S. (1997). Data Models in Geographic Information Systems. *Communications of the ACM*, 40(4):103--111.

Appendix A

Commands of The Neo4j Schema

Here we provide the commands to create the nodes and relationships of the example.

Creating nodes:

```
:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (bh:City {data}) return id(bh)", "parameters" : { "data" : { "name" : "Belo Horizonte", "coordinates" : "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"}}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (op:District {data}) return id(op)", "parameters" : { "data" : { "name" : "Ouro Preto", "coordinates" : "POLYGON ((15 20, 18 25, 15 25, 17 14, 15 40))" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (cas:District {data}) return id(cas)", "parameters" : { "data" : { "name" : "Belvedere", "coordinates": "POLYGON ((20 35, 25 38, 35 33, 30 30, 24 25, 20 35))" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (mpmg:Edification:Public {data}) return id(mpmg)", "parameters" : { "data" : { "name" : "MPMG", "gov" : "State", "coordinates": "POINT (28 35)" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (pu:Edification:Public {data}) return id(pu)", "parameters" : { "data" : { "name" : "PU_Belvedere", "gov" : "Municipal", "coordinates": "POINT (26 30)" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (fam:Edification:Private {data}) return id(fam)", "parameters" : { "data" : { "name" : "FAMINAS", "type" : "University", "coordinates": "POINT (23 33)" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (dru:Edification:Private {data}) return id(dru)", "parameters" : { "data" : { "name" : "Drugstore Araujo", "type" : "Pharmacy", "coordinates": "POINT (18 20)" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (cef:Edification:Public {data}) return id(cef)", "parameters" : { "data" : { "name" : "CEFET-MG", "gov" : "Federal", "coordinates": "POINT (20 16)" }}}]}
```

```
:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (cros1:Crossing {data}) return id(cros1)", "parameters" : { "data" : { "name" : "1", "coordinates": "POINT (40 50)" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (cros2:Crossing {data}) return id(cros2)", "parameters" : { "data" : { "name" : "2", "coordinates": "POINT (45 35)" }}}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "create (cros3:Crossing {data}) return id(cros3)", "parameters" : { "data" : { "name" : "3", "coordinates": "POINT (25 20)" }}}]}
```

Creating relationships:

```
:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{ name: 'Ouro Preto' }), (b:Edification{ name: 'Drugstore Araujo }), CREATE (a)-[:CONTAINS]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{ name: 'Ouro Preto' }), (b:Edification{ name: 'CEFET-MG' }), CREATE (a)-[:CONTAINS]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{ name: 'Belvedere' }), (b:Edification{ name: 'MPMG' }), CREATE (a)-[:CONTAINS]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{ name: 'Belvedere' }), (b:Edification{ name: 'PU_Belvedere' }), CREATE (a)-[:CONTAINS]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{ name: 'Belvedere' }), (b:Edification{ name: 'FAMINAS' }), CREATE (a)-[:CONTAINS]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:Crossing{name:'1'}), (b:Crossing{name:'2'}), CREATE (a)-[:STREET {name: 'X Street', coordinates: 'LINESTRING(40 50, 42 38, 45 35)'}]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:Crossing{name:'2'}), (b:Crossing{name:'3'}), CREATE (a)-[:STREET {name: 'Y Street', coordinates: 'LINESTRING(25 20, 35 30, 45 35)'}]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{name: 'Ouro Preto'}), (b:Crossing{name: '1'}) CREATE (a)-[:HAS_CROSSING]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{name: 'Belvedere'}), (b:Crossing{name: '2'}) CREATE (a)-[:HAS_CROSSING]->(b)"}]}

:POST /db/data/transaction/commit { "statements" : [ { "statement" : "MATCH (a:District{name: 'Belvedere'}), (b:Crossing{name: '3'}), CREATE (a)-[:HAS_CROSSING]->(b)"}]}
```