Analyse des solutions de spatialisation dans les graphes NoSQL afin þÿd exploiter la topologie arc noeud des réseaux routier

Auteur : Déom, Paul
Promoteur(s) : Donnay, Jean-Paul; Kasprzyk, Jean-Paul
Faculté : Faculté des Sciences
Diplôme : Master en sciences géographiques, orientation géomatique et géométrologie, à finalité spécialisée
Année académique : 2018-2019
URI/URL : http://hdl.handle.net/2268.2/7375

Faculté des sciences
Département de géographie

Analyse des solutions de spatialisation dans les graphes NoSQL afin d'exploiter la topologie arc nœud des réseaux routiers

Quelle est actuellement la solution la plus performante pour réaliser des requêtes spatiales sur un réseau routier important ?

Mémoire présenté par :
Déom Paul
Pour l'obtention du titre de :
Master en sciences géographiques,
orientation géomatique et géomètrologie

Année académique : **2018-2019**
Date de défense : **Septembre 2019**

Promoteur : **Pr. Jean-Paul DONNAY**
Co-promoteur : **Pr. Jean-Paul KASPRZYK**
Jury de lecture : **Pr. Xavier FETTWEIS**

Remerciements

Je tiens d'abord à remercier mon promoteur ainsi que mon co-promoteur : M. Donnay et M. Kasprzyk. Ils m'ont été d'une grande aide et m'ont conseillé durant toute la durée du mémoire. Je remercie également M. Nys pour son aide aussi bien technique que pour ses connaissances.

Je remercie aussi M. Fettweis pour l'attention qu'il porte à ce mémoire.

Enfin, je remercie ma petite amie et ma famille pour leur soutien et leur patience durant la réalisation de ce mémoire.

Résumé

Les opérations géographiques sur un réseau routier sont principalement réalisées sur des bases de données relationnelles, comme PostGIS. Toutefois, celles-ci ne sont pas toujours efficaces. Avec l'essor des bases de données NoSQL, la plateforme Neo4j a développé une base de données NoSQL graphe. L'avantage de la base de données Neo4j pour les applications géographiques simples, comme sélectionner le chemin le plus court, est confirmé par la littérature. Cependant, pour des opérations plus complexes et nécessitant un aspect spatial plus poussé, le modèle graphe n'est plus aussi performant.

Dans ce contexte, le but de ce mémoire est de mettre en avant une solution pour réaliser de façon efficace des opérations géographiques dans une base de données graphe. La solution qui est testée est d'utiliser une extension spatiale afin de fournir une géométrie aux éléments de la base de données et ainsi de pouvoir utiliser des outils spécifiques à la géographie (indexation et requêtes spatiales).

Cette solution est appliquée sur des opérations de sélection du plus court chemin et sur la sélection d'éléments du graphe proche d'un point extérieur au graphe. Elle est ensuite comparée à une base de données graphe sans l'extension spatial et surtout à une base de données relationnelle ayant aussi une extension spatiale : PostGIS.

Les résultats de l'analyse confirment l'hypothèse selon laquelle la base de données Neo4j sans l'extension spatiale est plus rapide pour le plus court chemin que PostGIS. Ils montrent aussi un gain de temps pour la réalisation de la recherche d'éléments proches d'un point mais aussi des pertes de performance dues à l'indexation spatiale de Neo4j. Enfin, des attentes et pistes de solutions sont établies pour des futures recherches.

Abstract

Geographic operations on a road network are mainly carried out on relational databases, such as Postgis. But these are not always effective. With the rise of Nosql databases, the Neo4j platform has developed a Nosql graph database. The advantage of the Neo4j database for simple geographic applications, such as selecting the shortest path, is confirmed by the literature. However, for more complex operations requiring a more extensive spatial aspect, the graph model is no longer efficient.

In this context, the purpose of this research is to find a solution to carry out geographic operations effectively in a graph database. The solution tested is to use a spatial extension to provide a geometry to the elements of the database to be able to use geographic tools like indexing and spatial requests. This solution is applied for selection of the shortest path and for selection of graph's elements close to a point outside graph. It is then compared to a graph database without the spatial extension and especially to a relational database with a spatial extension : Postgis.

The results of the analysis confirm the hypothesis that the Neo4j database without spatial extension is faster for the Postgis shortest path. They also show a time gain in the performance of the selection of elements close to a point but also performance losses due to the spatial indexing of Neo4j. Finally, expectations and possible solutions are established for future research.

Table des matières

Liste des figures

Liste des tableaux

Acronymes

SGBD-R = Système de gestion de base de données relationnelle

BASE = *Basically Available, Soft state, Eventual Consistency*

SQL = *Structured Query Language*

LPG = *Labelled Property Graph*

RDF = *Resource Description Framework*

OSM = *Open Street Map*

1.Introduction

Actuellement, une grande partie des applications géographiques sont réalisées sur des bases de données relationnelles. Un exemple d'application est d'utiliser des données géographiques pour déterminer des itinéraires. Cependant, des études ont démontré que ce type de base de données n'est pas performante pour toutes les applications géographiques.

En même temps, les bases de données NoSQL sont de plus en plus utilisées et de nombreuses plateformes développent leurs bases de données NoSQL. Certains types de bases de données NoSQL peuvent être intéressants pour des applications géographiques.

1.1. Description du modèle relationnel

Les bases de données les plus utilisées de manière générale relèvent du modèle relationnel (DB-ENGINES 2019). De nombreux ouvrages en détaillent toutes les finesses (à titre d'exemple, GARDARIN, 1999). Nous ne faisons figurer ici que les caractéristiques principales permettant de comparer par la suite les bases de données relationnelles à d'autres, relevant d'un modèle différent. Dans le modèle relationnel, l'information est structurée sous forme de tables (« relations » est le terme exact). Au sein d'une table, une entité est représentée par une ligne (ou *tuple*), tandis que les caractéristiques des entités, ou attributs, figurent en colonnes. Au moins un attribut sert de clé à la table, afin d'identifier les entités de manière univoque. Les tables d'une base de données relationnelles sont normalisées (jusqu'à 5 niveaux), afin de garantir la cohérence et d'éviter la redondance des données. Cependant, ces qualités nécessitent d'avoir recours à des jointures entre tables pour rapprocher les données éparpillées dans la base. Dans une base de données, les tables sont susceptibles d'être liées de différentes manières (relations simples, relations de généralisation-spécialisation, collections, etc.). Les cardinalités caractérisant chacun des rôles d'une association entre deux ou plusieurs tables indiquent le nombre d'entités participant à chacun des rôles de l'association. L'organisation des données au sein d'une base de données relationnelle est régie par un modèle de données, ou schéma, où sont spécifiés tous les éléments définissant la structure de la base (noms des tables, noms et domaines des attributs, clés, types d'associations et cardinalités, etc.). On qualifie ce type de base de données de structuré car les tables suivent un schéma rigide fixé *a priori*. Les bases de données relationnelles sont gérées par un système de gestion dédié (SGBD-R ou « *Relationnal Database Management System* » - *RDBMS*). Les SGBD relationnels garantissent le respect de contraintes ACID (atomicité, cohérence, isolation et durabilité) (A. HOLEMANS 2017) lors des transactions, afin de maintenir la base dans un état cohérent tout au long de son existence. Toutes les opérations de création, gestion et interrogation d'une base de données relationnelle sont réglées par les commandes du langage SQL « *Structured Query Language »* largement normalisé. Les bases de données relationnelles sont appropriées pour la gestion de données caractérisées par une structure fixe et prédéterminée, même complexe, et réclamant un degré élevé de sécurité.

1.2. Comparaison du modèle relationnel et NoSQL

Les bases de données relationnelles sont avantageuses sur plusieurs points. Elles sont utilisées depuis plusieurs décennies, sont stables et leurs méthodes de conception et d'implémentation sont bien maîtrisées. Elles ont largement su s'adapter aux innovations techniques (modèle relationnel-étendu ou objet-relationnel), au détriment cependant d'une certaine standardisation du langage SQL, et elles disposent d'outils garantissant les aspects sécuritaires à plusieurs niveaux (gestion des privilèges par exemple) (BATRA, *et al.* 2012).

Pourtant, elles présentent aussi des désavantages qui se sont avérés suffisamment bloquants pour voir apparaître ces dernières années, des bases de données relevant de modèles distincts. La nécessité de définir a priori le schéma détaillé de la base relationnelle rend son évolution parfois difficile (nouvelle spécialisation, nouvelle agrégation, changement de domaine, etc.) et d'autant plus lourde à réaliser que la base contient un grand nombre de données (BATRA, *et al.* 2012). Toutes les innovations liées au monde objet, usuel aujourd'hui en informatique, ne sont pas nécessairement prises en charge par le modèle relationnel (héritage multiple, attributs multivalués, etc.). Mais le principal reproche fait aux SGBDR porte sur la multiplicité des jointures entre tables requises par la normalisation des tables. Le temps de calcul d'un sous-ensemble du produit cartésien de deux tables devient prohibitif lorsque le nombre d'entités par table est fort important, et les performances du système s'en ressentent.

De ce fait, un nouveau modèle de stockage a été développé : le modèle NoSQL. Celui-ci a été imaginé dès les années 70 (YAQOOB, *et al.* 2016), à peine plus tard que le modèle relationnel, mais n'a commencé à être utilisé couramment que depuis une dizaine d'années, en réponse aux besoins des grandes entreprises œuvrant sur Internet (GAFAM et autres géants du Web). Les bases de données dites NoSQL prétendent éviter les limites des SGBDR (rigidité du schéma, jointures multiples, etc.) en garantissant une haute performance face à d'énormes quantités de données hétérogènes (« *Big Data* »). D'autant plus que cette quantité de données augmente encore, des prévisions montrent une augmentation du volume mondial de données d'environ 40% par an et qui irait jusque 50 fois en 2020 (YAQOOB, *et al.* 2016). Les modèles NoSQL utilisent des structures diverses pour le stockage de données, quel que soit leur type, ce qui permet un accès aisé, mais cela se fait évidemment au détriment de certaines caractéristiques qualitatives qui restent l'apanage du modèle relationnel.

1.3. Propriétés des bases de données NoSQL

Les propriétés ACID ne sont pas respectées par les bases de données NoSQL. Cependant, ces dernières respectent les propriétés BASE « *Basically Available, Soft state, Eventual Consistency* » et sont conformes au théorème CAP « *Consistencee, Availability, Partition Tolerance* » (NAYAK, *et al.* 2013), (OUSSOUS, *et al.* 2015).

Le théorème CAP a été mis en place lors du symposium sur les principes d'informatique distribué en 2000. Il est actuellement utilisé par la majorité de la communauté NoSQL. Ce théorème est composé de trois caractéristiques :

- La **cohérence** permet de consulter une base de données de manière simultanée à la modification des données.
- La haute **disponibilité** est une technique qui permet d'avoir accès aux données à tout moment grâce à la redondance des systèmes, même en cas de problème d'une des composantes.
- La **tolérance au partitionnement** est la capacité de pouvoir exécuter des requêtes sur des parties du réseau. Ce principe peut aussi être utilisable dans un but de maintenance.

Pour être cohérents, les modèles de base de données doivent respecter au moins deux des trois propriétés du théorème. En pratique, la disponibilité et la tolérance au partitionnement sont plus importantes qu'une consistance complète (SHARMA, *et al.* 2012).

Les avantages énoncés peuvent rendre les bases de données NoSQL plus intéressantes que les bases relationnelles. La différence est surtout visible pour des applications précises : recherche de données, analyse et stockage de données, *Machine Learning*, technique de visualisation (YAQOOB, *et al.* 2016).

Cette recherche a pour but de cibler une de ces applications et d'évaluer si la base de données NoSQL est effectivement plus performante. Les éléments qui influencent la performance des bases de données seront aussi identifiés afin de proposer des variations de la configuration de notre base de données et des requêtes utilisées.

1.4. Données géographiques et NoSQL

Les données et applications à caractère géographique sont aujourd'hui nombreuses sur le web. À côté des applications désormais classiques de visualisation cartographique auxquelles sont souvent liées les recherches d'itinéraires (Google Maps, Bing Maps, Via Michelin, Mappy, etc.), l'information géographique est présente sur quasiment chaque site : explicitement, dans des applications de *crowsourcing* telles que OpenStreetMap par exemple, et implicitement à travers le suivi de la localisation des utilisateurs via leur adresse postale, essentiellement pour les besoins du « geo-marketing ».

Les données géographiques manipulées sur le web suivent le plus souvent un modèle simple : outre un identifiant, un seul attribut et une géométrie ponctuelle suffisent à la plupart des applications banales. Ainsi limitées, et même si elles sont en très grand nombre, les données géographiques sont candidates à des formes de stockage élémentaires, telles que celles prônées par les modèles NoSQL.

Mais les choses évoluent rapidement et certains modèles NoSQL offrent déjà la capacité de gérer des géométries plus complexes (linéaires, polygonales, structures raster). Des travaux récents de l'Unité de Géomatique de l'Université de Liège y ont d'ailleurs été consacrés (HOLLEMANS 2017 ; JEMINE 2017 ; HOLLEMANS, *et al.* 2018). Il s'avère que l'efficacité relative des modèles NoSQL pour la gestion des données géographiques est, dans l'état actuel des choses, très largement dépendante de la nature des traitements pressentis sur ces données (utilisation ou non de la topologie, modification ou non des géométries, etc.). Il est donc pertinent et important, de sélectionner attentivement le domaine d'application et la forme exacte des données géographiques utilisées pour envisager une évaluation de l'efficacité relative d'un modèle NoSQL face à un modèle relationnel spatialisé traditionnel.

1.5. Question de recherche et structure

Outre cette introduction, le mémoire est subdivisé en cinq parties présentées brièvement ci-dessous :

Premièrement, l'état de l'art résume ce qui existe et a déjà été réalisé dans la comparaison des SGBD-R et NoSQL. La recherche permet d'identifier une application géographique et d'évaluer si une base de données NoSQL est effectivement plus performante. Les éléments qui influencent l'efficacité des bases de données sont aussi identifiés afin de proposer des améliorations de la configuration d'une base de données NoSQL.

Ensuite, la deuxième partie présente l'hypothèse qui constitue la base de la recherche. Celle-ci est émise suite à l'analyse des constats tirés de l'état de l'art.

En troisième partie, la configuration décrit les différentes étapes préalables à l'analyse.

L'analyse est développée dans la quatrième partie, elle reprend toutes les opérations réalisées et présente les résultats obtenus à chaque étape.

Enfin, la conclusion est tirée de l'analyse des résultats.

2.Etat de l'art

2.1. Bases de données NoSQL

Depuis le début de l'utilisation des bases de données NoSQL, il existe quatre catégories de base de données ainsi qu'une grande diversité de modèles : ces catégories sont colonne, document, clé-valeur et graphe. Ceux-ci varient les uns par rapport aux autres selon de nombreux facteurs : les caractéristiques et structures de modélisation, le type de données et les principes théoriques (R. &. ANGLES, *et al.* 2008). Les bases de données NoSQL permettent de gérer de grandes quantités d'informations. Les systèmes modernes utilisent ces nouvelles bases de données pour des applications variées : l'analyse de réseau routier, gestion de vente en ligne, gestion de forum…

La gestion des données se fait de diverses façons, en fonction du modèle NoSQL. Cette différence est un des éléments qui distingue les modèles NoSQL du modèle relationnel. Le modèle relationnel est donc constitué de tables, les relations. Celles-ci peuvent avoir plusieurs entités, les lignes de la table aussi appelées tuple. Au contraire, les modèles NoSQL ont des structures différentes qui permettent de stocker les données différemment des tuples.

Les bases de données **clé-valeurs** fonctionnent grâce à la relation entre une clé unique, en format texte, et les données. Le système de stockage est similaire à des tables dont les clés servent d'index (Figure 1). Ce type de base de données est plus rapide que les SGBD-R (HAN, *et al.* 2011). Le fonctionnement est simple : l'utilisateur recherche une donnée grâce à une clé spécifique. Ce type de modèle peut être utilisé pour des sites de vente en ligne (Amazone), des forums...

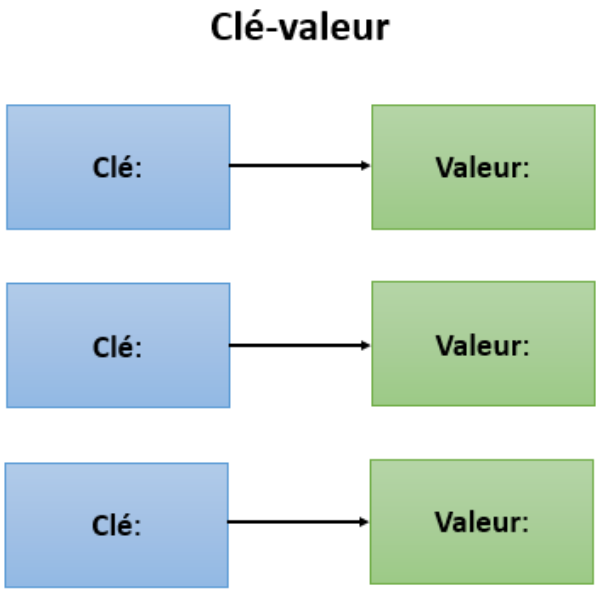


Figure 1 : Exemple de schéma de la structure orientée Clé-Valeur (Digora, s.d. ; modifiée)

Le type de base de données **orientée colonne** est une combinaison de stockage selon les colonnes et selon les lignes (Figure 2). Des clés sont associées aux colonnes et aux lignes ainsi qu'un ou plusieurs

attributs. Ce modèle est utilisé pour les recherches d'informations ainsi que les analyses. Cassandra est un exemple de base de données NoSQL orientée colonne. Elle a été développée par Facebook afin de pouvoir réaliser énormément d'opérations simultanément (LAKSHMAN, *et al.* 2010). Elle est également utilisée par Amazon ou encore Google.

Figure 2 : Exemple de schéma de la structure orientée colonne (Digora, s.d. ; modifiée)

Les bases de données **orientées documents** permettent de gérer une grande quantité d'informations sous forme de document. Un document est assimilé à une table du modèle relationnel et contient des tuples. Ce type de stockage fournit une grande performance (HAN, *et al.* 2011). Il propose également de nombreuses possibilités d'évolution pour les bases de données orientée documents car elles ne possèdent pas de schéma conceptuel. On retrouve cependant des représentations similaires ainsi que deux types de modèles document : Hiérarchique ou « Embedded » (WAGNER, *et al.* 2015). Cette caractéristique rend les bases de données faciles d'utilisation et applicables dans de nombreux cas. Dans les documents, les données sont sous forme d'objets, comme dans une base de données relationnelle (Figure 3). Les documents peuvent être de différents formats standards : XML, PDF, JSON..., ce qui fait varier le comportement des documents. Chaque document possède certaines informations, qui peuvent être différentes, et une clé unique par document. Par exemple, MongoDB et CouchDB sont deux modèles de base de données documents open source.

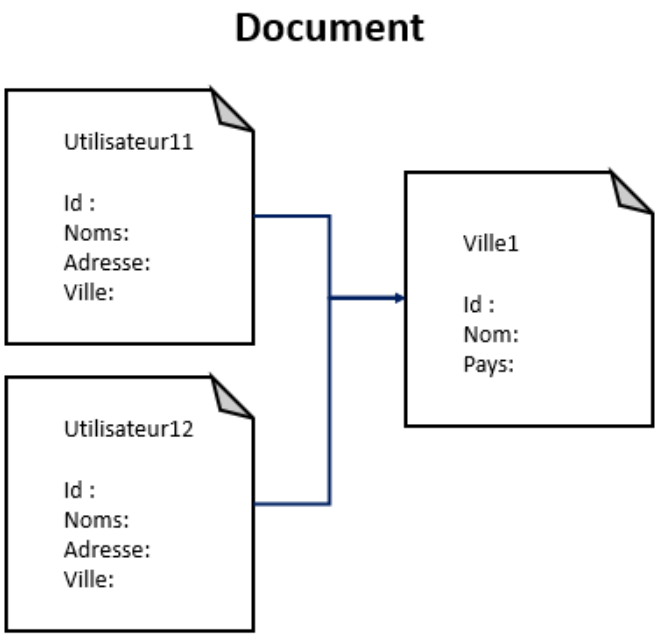


Figure 3 : Exemple de schéma de la structure orientée document (Digora, s.d. ; modifiée)

Enfin, le dernier type est la base de données **orientée graphe**. Elle permet aussi de structurer une grande quantité d'informations mais en suivant la structure d'un graphe. Elle est composée de nœuds et d'arcs (DE SOUZA, *et al.* 2014). Les arcs représentent les relations entre deux objets qui sont représentés par les nœuds (Figure 4). Ce système permet de représenter au maximum les connexions entre les données. Il n'est pas nécessaire d'avoir un schéma pour ce type de base de données. L'avantage des bases de données graphes est qu'elles possèdent une structure inhérente aux différentes situations de la vie réelle : plan de cours, réseaux neuronaux, réseaux de transports en commun, trajets d'avion ou de bateau et leurs ancrages respectifs… (ROBINSON, *et al.* 2015). Ce type de base de données est particulièrement avantageux pour la représentation de données géospatiales telles que les réseaux routiers (MILER, *et al.* 2014). Les réseaux peuvent être représentés par des graphes très différents suivant la quantité d'informations que l'on veut conserver et l'utilisation de ces informations. Il existe donc deux grands types de bases de données orientées graphe : le « Resource Description Framework » (RDF) et le « Labelled Property Graph » (LPG).

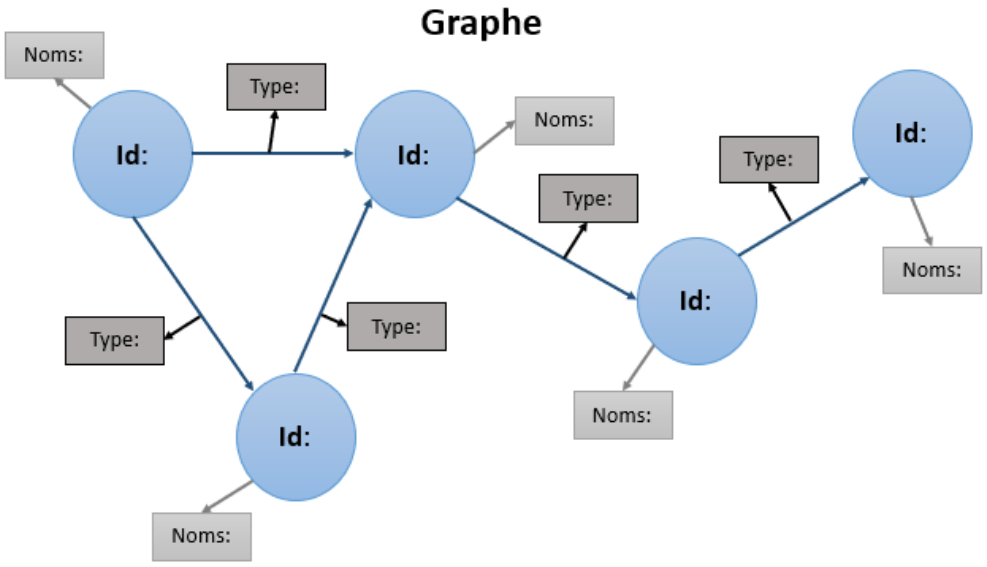


Figure 4 : Exemple de schéma de la structure graphe (Digora, s.d. ; modifiée)

Les types de bases de données NoSQL peuvent être triés suivant trois critères :

- La **flexibilité** est liée à la capacité de lecture des données suivant la variation de quantité de données
- Les **données et modèles de requête** influencent grandement le choix du type de NoSQL à utiliser. En effet, ce choix dépend des caractéristiques des données, de leur utilisation et des requêtes à appliquer.
- La **persistance de la structure**, c'est-à-dire que dans le cas où le système n'est plus capable de lire les données, il réalise un partitionnement, ce qui a des effets négatifs sur la base de données. Des alternatives peuvent être prévues afin d'éviter un partitionnement mais elles ne sont pas toujours mises en place (SHARMA 2012).

Différentes structures possèdent des persistances variables. Une base de données stockée sur la mémoire engendre un modèle rapide mais ayant une taille limitée à la mémoire RAM disponible. Il est aussi possible de conserver les données sous forme de tables et écrites dans la mémoire RAM. Elles sont ensuite transférées sur le disque afin de conserver une performance comparable à l'utilisation unique de la mémoire RAM et sans être limitées par la quantité de données. La dernière structure, sous la forme d'un Btree, fournit un bon support pour une indexation. Les performances sont cependant moins grandes puisqu'il y a une dépendance avec l'efficacité du disque dur.

Le type de base de données utilisé est donc essentiel. Il dépend des besoins de l'utilisateur et des données à traiter.

2.2. Types de bases de données graphes

Comme expliqué précédemment, il existe deux représentations des bases de données graphes (Figure 5). Le RDF reprend l'information de façon complète mais volumineuse. L'autre représentation des graphes est le LPG qui est plus synthétique et donc moins volumineux.

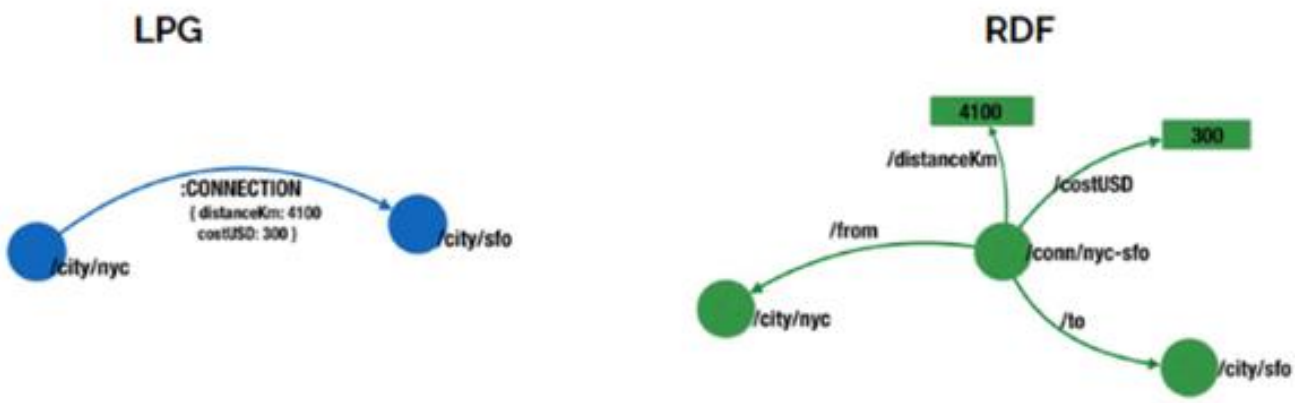


Figure 5 : Représentation du modèle LPG et RDF (Big on Data, 2017).

D'une part, le RDF fait partie d'un ensemble de graphe appelé « Edge-Labelled Graph ». Ce type de graphe est caractérisé par des étiquettes qui identifient chaque arc afin de définir la relation entre deux nœuds. Dans le cas du RDF, les nœuds possèdent un identifiant unique, un URI (Uniform Resource Identifier), et plusieurs attributs numériques ou littéraux, qui constituent les caractéristiques de l'objet (GHRAB, *et al.* 2018). Les URI sont des identifiants uniques liés à une ressource qui permet à n'importe quel utilisateur de les identifier. Ils respectent une règle de syntaxe afin d'être uniformes. Un exemple est l'utilisation de « http:// » avant un lien internet. Les arcs correspondent aux relations entre les différents objets. Ils ne possèdent pas d'attribut mais une caractéristique et un identifiant (un URI). Le RDF suit un schéma qui se répète pour former le graphe. Les arcs et nœuds forment des triplets avec un sujet, un arc et un objet. Le premier nœud est le sujet de la relation, identifié grâce à son URI. Le second nœud est l'objet de la relation, avec une certaine valeur et les deux nœuds sont liés par un arc, caractérisé par un prédicat (THEOHARIS, *et al.* 2005). Cela forme une structure de graphe simple qui n'est soumise à aucune restriction de la quantité de données, ni au niveau du type d'attributs, etc. (ANGLES, *et al.* 2017).

D'autre part, le second type de graphe est le LPG. Un « Property graph » est un graphe où les arcs et les nœuds peuvent chacun avoir une étiquette ainsi qu'un attribut spécifique. Il est possible d'avoir plusieurs étiquettes et attributs dans le cas de graphes qualifiés de « *Multi-Valued Property* » (ANGLES, *et al.* 2017). Dans un graphe LPG, les nœuds sont interprétés comme des entités. Chaque nœud possède une clé-valeur, c'est à dire un identifiant unique à cette entité. Les nœuds ont aussi une ou plusieurs valeurs qui sont les attributs du nœud. Les nœuds peuvent être regroupés sur des couches et assimilés à un type. Dans le cas présenté ci-dessus (Figure 5), il s'agit du type « city » (JAROSLAV 2015). Une fois les entités identifiées, les arcs représentent une relation entre deux entités. Les arcs possèdent une certaine direction, allant d'une entité à l'autre. Elles sont aussi définies par un identifiant unique et peuvent avoir un ou plusieurs attributs, quantitatifs ou qualitatifs. Il est aussi possible de mettre en avant une structure en triplet dans un graphe LPG. En effet, la structure est constituée de deux entités composées d'attributs, un nœud de départ et un nœud d'arrivée. Le nœud de départ est relié au nœud d'arrivée par un arc également constitué d'attributs.

Ce type de graphe possède une structure flexible, il est aisé d'ajouter ou de supprimer des entités ou des arcs. Il stocke la même information qu'un graphe RDF mais de façon plus synthétique, ce qui

permet d'effectuer des recherches de nœuds et d'arcs plus rapidement. On appelle parcours transversal du graphe (VICKNAIR, *et al.* 2010) la capacité de retrouver le nœud voulu en minimisant le nombre d'entités et d'arc consultés. (DOMINGUEZ-SAL, *et al.* 2010)

2.3. Langages de requête

Le langage de requête permet de manipuler les données dans les bases de données. Dans le cas des SGBD-R, un langage standardisé a été mis en place, il s'agit du SQL « *Structured Query Language* ». Pour les bases de données NoSQL, chaque modèle utilise un langage qui lui est propre, pour manipuler les données. Des tests de langage standardisé pour les plateformes NoSQL ont été essayés. Le UnQL « *Unstructured Query Language* » est le résultat de cette standardisation mais peu de plateformes l'utilisent pour le moment (NAYAK, *et al.* 2013).

Pour les bases de données NoSQL orienté graphe, il existe plusieurs langages de requêtes fréquemment utilisés correspondant à différentes plateformes. Le premier permet d'interagir avec un graphe RDF, il s'agit de SPARQL qui a été officiellement standardisé comme langage de requête pour les graphes RDF. GeoSPARQL est une variante de ce langage qui a été développé pour les graphes RDF ayant un caractère géographique. Il s'agit d'une solution proposée par l'OGC pour la représentation et l'accès à des données géospatiales (BATTLE, *et al.* 2011). Le second langage est Gremlin. Il a été créé par la licence Apache dans le but d'être utilisé pour interroger l'ensemble des bases de données de type « Property Graph ». Enfin, le dernier langage est le Cypher, spécialement développé propriétairement par Neo4j pour sa base de données LPG.

Les langages ont des domaines d'applications variés : analyse de structure bio-chimique, machine learning, analyse de réseau, reconnaissance de pattern... (ANGLES, *et al.* 2017)

2.4. Environnement Neo4j

Il existe différents systèmes de LPG (Labelled Property Graph). Ils ont des niveaux de développement plus ou moins aboutis et ils possèdent chacun leurs particularités. Les plus développés sont Sones graphDB, la partie orientée graphe d'ArangoDB, OrientDB et Neo4j. D'autres systèmes moins aboutis sont spécifiques à certaines applications et se basent souvent sur un langage de requête. Des langages de requêtes ont aussi été conçus séparément, comme GraphLog, G ou encore GOOD. Ils permettent de développer des systèmes de base de données graphes en fonction des besoins du gestionnaire des données (WOOD 2012).

Sones graphDB possède peu d'information et de documentation mais il s'agit d'un système de gestion destiné à un réseau routier, avec des données géospatiales (WOOD 2012).

Dans le cas d'ArangoDB, il ne comprend pas uniquement une base de données orientée graphe mais l'ensemble des systèmes NoSQL présentés précédemment : clé-valeur, document et graphe. ArangoDB est bien documenté et utilise son propre langage de requête. La partie graphe d'ArangoDB fonctionne avec une série de nœuds et d'arcs. Les nœuds sont des entités qui doivent être définies par le

gestionnaire de la base de données. Un nœud peut donc soit correspondre à un document ou à un simple nœud. De la même façon, le gestionnaire doit définir de quelle manière les arcs doivent être interprétés. Toutefois, ils possèdent toujours une étiquette pour spécifier le type de connexion dont il s'agit. Les possibilités d'ArangoDB sont donc très poussées mais cela rend son utilisation plus complexe. (ArangoDB s.d.)

OrientDB est un modèle de base de données NoSQL orientée graphe, open source et développée en java, sous la licence Apache. Le langage de requête utilisé est le SQL. Il possède une grande flexibilité et est performant pour l'insertion de données ainsi que pour la réalisation de requête (OrientDB s.d.).

Neo4j est un modèle de graphe LPG avec un langage de requête spécifique, Cypher, ainsi qu'une interface offrant de nombreuses fonctionnalités. Celui-ci est documenté dans de nombreux articles fournis par la communauté ou par les développeurs du modèle (DALTIO, *et al.* 2016). Il est entièrement codé en java et beaucoup de programmes sont en développement autour du modèle, comme des plug-ins, des packages Python... Neo4j permet de visualiser les données sous forme d'entités, avec ou sans les arcs existants, ou sous forme de tableaux, ce qui simplifie l'analyse du résultat obtenu (ANGLES 2012) (DOMINGUEZ-SAL, *et al.* 2010).

Cypher est basé sur le SQL. Il se structure de la même façon que les requêtes SQL par la sélection de données dans une certaine couche, suivant un certain filtre. Cependant, Cypher a aussi une structure visuelle logique, puisque l'écriture des arcs est composée de deux entités de part et d'autre de la relation (JAROSLAV 2015). Dans ce cas, un arc caractérise le lien entre les deux entités, ce qui est différent d'une relation du modèle relationnel, qui a été expliquée précédemment.

2.4.1. Outils externes utilisables

Des outils spatiaux sont fréquemment développés pour les bases de données. Ils peuvent avoir plusieurs buts tels que l'importation de données géospatiales, la création de la géométrie pour des entités ou encore l'application de requêtes spatiales. Ce genre d'extension spatiale existe aussi pour d'autres types de NoSQL, comme GeoCouch[1] pour CouchDB. MongoDB gère aussi les données spatiales mais n'a pas besoin d'extension (BAAS 2012).

Dans le cas de Neo4j, le plug-in est Neo4j Spatial[2] et il est toujours en développement. Il offre des variations intéressantes de Neo4j en fournissant beaucoup d'applications géospatiales. Il permet par exemple de faire une indexation spatiale et il fournit des outils pour réaliser des requêtes spatiales (Neo4j contrib 2017).

D'autres outils et guides sont proposés pour Neo4j par rapport à des éléments de programmation : Python, java, Geoserver. Une relation entre Neo4j et Python peut être faite grâce à différents packages : Py2neo[3], Pypher[4], Neomodel[5]. Cependant, la base de données étant récente, les packages sont en

[1] https://github.com/couchbase/geocouch
[2] https://github.com/neo4j-contrib/spatial
[3] https://py2neo.org/v4/
[4] https://github.com/emehrkay/Pypher
[5] https://neomodel.readthedocs.io/en/latest/

développement. Le plus complet et mieux développé est py2neo. Il permet de réaliser des requêtes Cypher directement sur la base de données graphe.

2.5. Spatialisation

Une notion fondamentale de cette recherche est la spatialisation. La spatialisation est la représentation géospatiale de l'information qui peut se faire de différentes façons. Pour des points, le plus fréquent est de déterminer des coordonnées spécifiques exprimées suivant un certain système de référence. La spatialisation est aussi caractérisée par la création de la géométrie des éléments. Elle peut être exploitée grâce à un index spatial. L'index est construit sur base des coordonnées géographiques des points et reprend une notion de proximité. L'ordre suivant lequel les points proches sont retrouvés varie en fonction de l'index utilisé. Il permet d'accélérer, grâce à la spatialisation d'éléments, le temps de recherche des requêtes spatiales (SKUPIN, *et al.* 2007).

2.5.1. Indexation spatiale

Comme spécifié précédemment, la spatialisation peut être réalisée de différentes façons :

- par l'utilisation de coordonnées,
- par le traitement de géométries,
- et par l'utilisation d'index spatial.

Celui-ci permet, sur base des géométries ou des coordonnées, de structurer l'information lors de requêtes. Les indexations peuvent être spatiales ou non spatiales. L'indexation non spatiale utilise uniquement les clés primaires et les identifiants des éléments pour trier les entités séparément les unes des autres. Dans le cas d'un index spatial, on utilise la position des entités dans l'espace pour créer l'index afin que deux éléments proches spatialement soient proches dans l'arbre d'indexation.

Plusieurs index spatiaux ont été imaginés. Ils varient par leur interprétation des données mais aussi par le nombre de dimensions pris en compte. Le « *Btree* » est un index spécialisé pour une dimension. Il représente les données comme un arbre avec des feuilles. Ces feuilles correspondent aux données stockées et sont regroupées en nœuds. Il est aussi possible d'appliquer cette indexation sur deux dimensions. Pour cela, une technique de lecture des données est mise en place afin d'interpréter les deux dimensions de façon linéaire, suivant un découpage de l'espace et un parcours prédéfini. L'index Btree est par exemple utilisé par MongoDB, une base de données NoSQL orientée document pour structurer les différents documents de la base de données (A. HOLEMANS 2017).

Le deuxième index est spécifique à un espace à deux dimensions, le « *2Dtree* ». Il crée un arbre permettant de retrouver la position de points dans l'espace. Les nœuds sont regroupés suivant une division de l'espace en rectangles.

Le Quadtree est un index qui permet l'indexation de points, polylignes et polygones. Pour cela, il structure la base de données en plaçant chaque point composant la géométrie dans une cellule. La structure de l'arbre se fait en respectant certaines règles. Ces règles varient en fonction du type de Quadtree. Il existe le « *Region Quadtree* », le « *Point Quadtree* » ou encore le « *PM Quadtree* ». Le

« *Region Quadtree* » est utilisé pour indexer des polygones. Pour cela, il divise la zone en quatre cellules et chaque cellule qui n'est pas couverte de façon homogène par le polygone est subdivisé en quatre. Le « *Point Quadtree* » permet d'indexer des points. Il se développe un peu de la même façon que le « *2DTree* », c'est-à-dire que l'espace est divisé par une droite verticale et une horizontale passant par le point, jusqu'à ce qu'il n'y ait plus de point dans les espaces divisés. Le « *PM Quadtree* » est une indexation de polylignes ou de polygones. Il s'agit à nouveau de diviser l'espace en cellules qui ne contiennent qu'un seul point ou partie de droite.

Enfin, le dernier type d'indexation est le Rtree. La structure du Rtree est similaire à celle du Btree, car elle est composée de feuilles dans différents nœuds (Figure 6). Les éléments géométriques sont contenus dans des enveloppes et ce sont les enveloppes qui sont indexées. Les feuilles sont les enveloppes des géométries et elles sont réparties dans des conteneurs : les nœuds. Il existe aussi une variante au Rtree, le Rtree+. Celui-ci permet de mieux gérer les entités de formes et d'étendues très variables, mais il est plus lourd à mettre en œuvre, car il ne permet plus aux conteneurs de se superposer et implique une partition possible d'une même entité entre plusieurs conteneurs. Le Rtree est disponible sur des bases de données NoSQL comme Neo4j et le Rtree+ est utilisé sur la base de données relationnelle PostGIS. (MARTINS, *et al.* 2005).

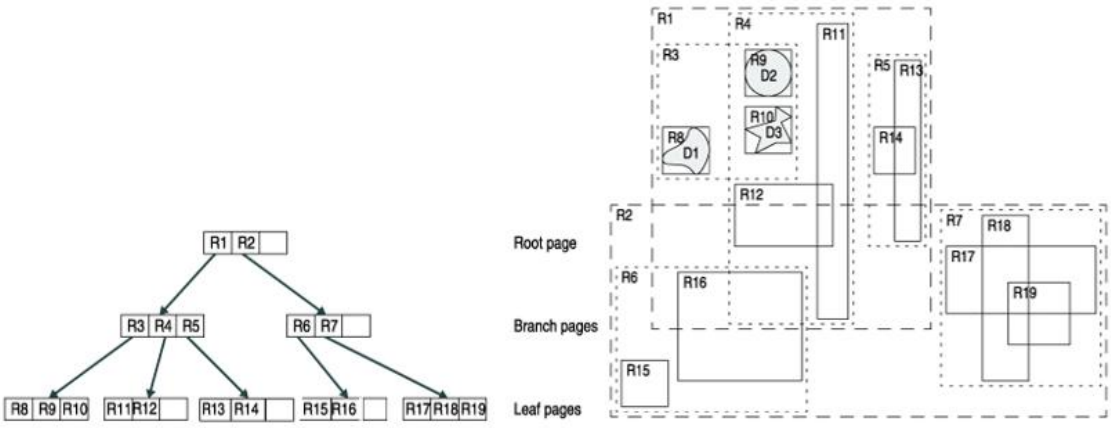


Figure 6 : Structure de l'indexation Rtree

L'indexation spatiale est possible dans les bases de données ayant des valeurs non nulles (GUTTMAN 1984). Une requête spatiale fonctionne en deux étapes. Lors de la première étape, il traverse le graphe pour retrouver le chemin et les entités recherchées. La seconde étape est de calculer le résultat de la requête sur base des entités demandées et des informations liées (ZHAO, *et al.* 2001). Ce texte met en avant l'avantage d'une indexation spatiale pour le parcours transversal du graphe. En effet, l'indexation permet à la base de données de déterminer plus facilement quel arc doit être choisi pour parcourir efficacement le graphe.

2.5.2. Requêtes spatiales

Différentes requêtes spatiales sont appliquées dans la recherche suivante. Le plus court chemin y est testé et permet de vérifier l'intérêt de l'utilisation d'une base de données graphe (GUBICHEV, *et al.* 2010). Il s'agit de l'utilisation du point des arcs afin de définir le trajet optimal entre un nœud de départ et un nœud final. Pour effectuer cette requête, un algorithme spécifique est utilisé. Il s'agit de l'algorithme du plus court chemin de DIJKSTRA (AHUJA, *et al.* 1990). Celui-ci a de nombreux avantages,

outre le fait d'être efficace, il est disponible sur la SGBD-R PostGIS ainsi que sur la base de données graphe Neo4j. Il existe d'autres algorithmes du plus court chemin : « *Bellman-Ford-Moore* », « *Incremental graph* », « *Treshold algorithme* » et « *Topological Ordering* ». Cependant, l'algorithme de DIJKSTRA suit une règle qui le distingue des autres algorithmes. Il permet de définir le plus court chemin dès qu'un nœud possède un label fixe et non quand l'ensemble du réseau est terminé, ce qui le rend plus flexible que les autres. De plus, l'algorithme de DIJKSTRA est un des plus performants (ZHAN, *et al.* 1998).

Il est aussi possible d'utiliser une heuristique au lieu d'un algorithme pour calculer le plus court chemin. Dans ce cas, le résultat n'est pas un plus court chemin fixe mais plusieurs possibilités. Il s'agit du cas le plus fréquent dans la littérature et essentiellement l'heuristique A* (FU, *et al.* 2006).

La requête suivante est la sélection du plus proche voisin d'un point. Le plus proche voisin peut se faire de différentes façons. Il faut faire la distinction lorsque le nœud de départ est situé dans le graphe ou hors du graphe. Dans le premier cas, il s'agit alors de sélectionner le nœud dont l'arc, entre les deux nœuds, a la plus petite valeur.

Dans le second cas, le nœud de départ est qualifié de « hors du graphe ». De ce point, il est possible de sélectionner deux types de points. Un type de sélection est de prendre le nœud le plus proche (Figure 7). Ceci peut se faire de deux façons différentes en fonction de l'information disponible :

- Le nœud le plus proche est calculé directement en prenant la distance euclidienne la plus faible.
- Le nœud est déterminé en fonction d'un poids nécessaire au parcours. Cette méthode demande cependant des informations supplémentaires comme le type de route ou l'occupation du sol (RITSEMA VAN ECK, *et al.* 2002).

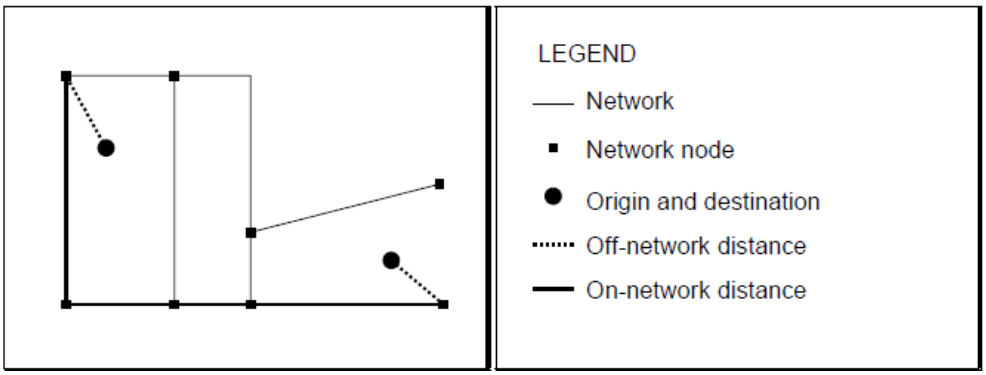


Figure 7 : Nœud le plus proche du point hors graphe (RITSEMA VAN ECK 2002)

L'autre sélection possible est de trouver le point du graphe le plus proche du point hors graphe (Figure 8). Dans ce cas, les mêmes approches peuvent être prises en compte. L'arc du graphe le plus proche du point hors graphe est sélectionné pour ensuite calculer le point de graphe le plus proche. L'arc le plus proche est choisi par distance euclidienne ou grâce à une pondération hors graphe fournie.

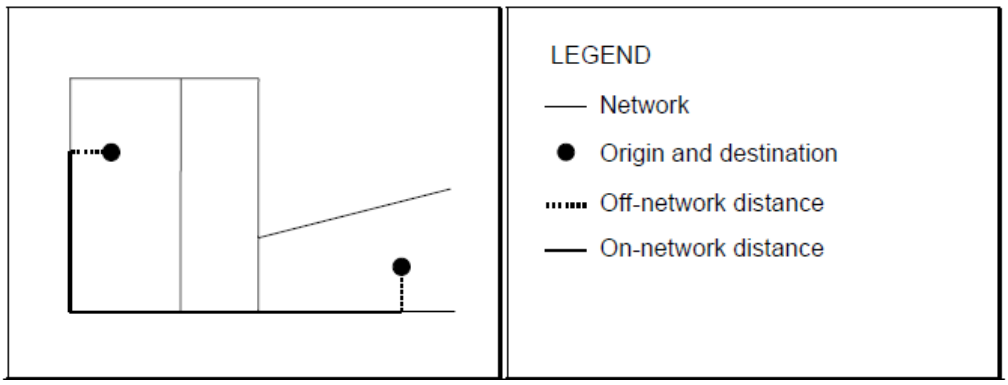


Figure 8 : Point d'un arc le plus proche du point hors graphe (RITSEMA VAN ECK 2002)

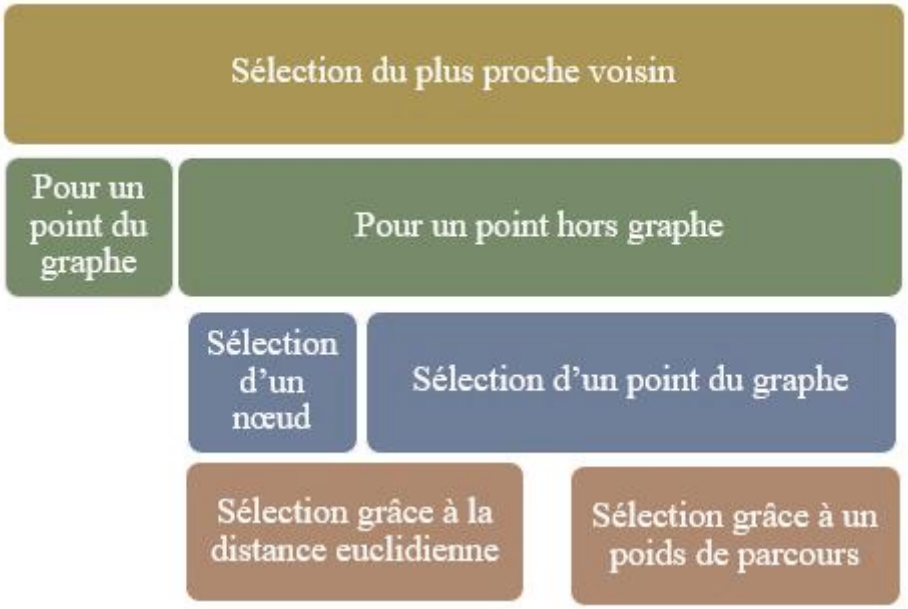


Figure 9 : Schéma récapitulatif des différences entre les plus proches voisins

Il est possible de prendre d'autres éléments en considération comme la notion de barrière naturelle, telle que des ruisseaux, qui ne permettent pas l'accès à une partie de point du graphe.

2.5.3. Application à un réseau

Un réseau possède une structure topologique ayant l'aspect d'un graphe. Ce graphe est composé de nœuds et d'arcs, où les nœuds sont des intersections et les arcs, les connexions entre les nœuds. Les nœuds sont donc connectés entre eux par les arcs. Un nœud peut avoir de multiples arcs qui le relient à plusieurs autres nœuds. Le type de nœuds et d'arcs ne suffit pas toujours à représenter les réseaux, il est possible d'ajouter de l'information sur la structure topologique (CALVERT, *et al.* 1997). Le nombre d'arcs par nœud est appelé le degré de connectivité.

Il existe différents réseaux qui ont des topologies arc-nœud et sur lesquels les requêtes du plus court chemin et de plus proche voisin peuvent être faites ; à titre d'exemples : les réseaux neuronaux, les réseaux sociaux, les réseaux de téléphones ou encore les réseaux routiers.

Les réseaux neuronaux sont composés de nœuds qui représentent les neurones et d'arcs, les axiomes. Un réseau neuronal possède une position relative des neurones et des axiomes mais celle-ci nécessite des connaissances poussées en biologie. Pour les réseaux sociaux, les nœuds sont des entités comme des personnes ou des groupes et les arcs sont les rapports entre ces personnes. Les arcs des réseaux sociaux représentent les relations qu'il y a entre les différentes entités : amitié, relation contractuelle, vente ou un simple échange de messages. Dans ce cas, les nœuds peuvent être associés à un lieu et donc à des coordonnées. Cependant, les informations des réseaux sociaux sont peu fiables et difficiles à obtenir. Les réseaux de téléphonie reprennent chaque téléphone comme un nœud et les messages ou appels sont les arcs entre les différents téléphones. Ceux-ci possèdent aussi une localisation dans l'espace mais la position des téléphones est variable et l'information est difficile à obtenir. Le dernier réseau est le réseau routier. Les nœuds représentent les carrefours et échangeurs, alors que les arcs sont les routes entre ces différents carrefours. Ce type de réseau varie peu et est souvent représenté. Les différents réseaux routiers disponibles en ligne possèdent des coordonnées qui permettent de le rattacher à un système d'information géographique. Les réseaux routiers sont disponibles gratuitement en ligne. De plus, les algorithmes du plus court chemin sont fréquemment appliqués à des réseaux routiers. De nombreuses recherches tentent d'améliorer les performances du plus court chemin en les appliquant à ce type de réseau (ZHAN, *et al.* 1998).

2.6. Questions pendantes

Les analyses suivantes évaluent l'efficacité des bases de données NoSQL orientées graphes en effectuant des comparaisons entre différents modèles graphes existants et par rapport au modèle SGBD-R le plus utilisé : PostgreSQL (AMIRIAN, *et al.* 2014).

Il existe différentes possibilités d'exploitation du parcours transversal d'une base de données graphe. La première est d'utiliser les graphes LPG afin d'enrichir une carte interactive (CATTUTO, *et al.* 2013), (P. B. AMIRIAN, *et al.* 2015), (JOSSE, *et al.* 2017). Le graphe permet d'aller rechercher rapidement de l'information liée à un nœud. L'information fournie dépend du but recherché. Dans le cas d'un outil touristique, l'information cible des données sur les points d'intérêt proches et des connaissances directement pour ces points d'intérêts. Pour une application plus pratique, sur un réseau routier des nœuds peuvent représenter des carrefours, tunnels, ponts et l'information fournie reprendrait des caractéristiques de ces éléments. Il est possible d'imaginer beaucoup d'autres applications qui s'appliquent à un parcours transversal du graphe.

Une approche différente est d'exploiter le parcours transversal afin d'optimiser la rapidité de requêtes spatiales. Ce type d'analyse a été testé de façon plus ou moins complète par d'autres études. Tout d'abord, des comparaisons peu détaillées entre les bases de données graphes et les bases de données relationnelles ont été faites (BATRA, *et al.* 2012) (CIGLAN, *et al.* 2012). Les résultats mettent en avant que le temps d'exécution des requêtes dans la base de données graphe LPG Neo4J est plus courte que dans les RDMBS.

Les performances de requêtes spatiales sont analysées par d'autres recherches. L'une d'elles implique une comparaison entre un graphe LPG et un réseau routier sur PostGIS (MILER, *et al.* 2014). Cette dernière étude se concentre sur la recherche du plus court chemin par rapport à de nombreux critères :

la vitesse d'exécution, l'espace disque utilisé lors du processus ou la température du processeur lors de l'exécution. Pour cette comparaison, l'algorithme du plus court chemin utilisé est celui de DIJKSTRA. C'est celui qui est le plus souvent utilisé dans la documentation car il est flexible et performant. Dans ce cas, il est aussi disponible sur PostGIS et sur Neo4j, ce qui permet de comparer la vitesse d'exécution des bases de données et non de leurs algorithmes. La conclusion est que l'efficacité de Neo4j est supérieure à celle de PostGIS. Cependant, l'étude met aussi en avant une utilisation plus importante du disque dur pour Neo4J.

D'autres auteurs analysent des requêtes spatiales plus variées comme le nombre d'entités contenues dans une zone ou encore le voisin le plus proche (BAAS, *et al.* 2012). La base de données graphe utilisée est Neo4J, elle est comparée à un SGBD-R (PostGIS). Le résultat obtenu est que le plus court chemin réalisé est plus rapide sur Neo4j, peu importe la taille du réseau. Cependant, dans le cas des autres requêtes spatiales réalisées, il n'y a pas d'avantage quant au temps d'exécution pour la base de données graphe. Les éléments qui peuvent ralentir les requêtes spatiales sont :

- La géométrie n'est pas prise en compte par Neo4j, ce qui engendre des étapes supplémentaires pour les requêtes spatiales.
- L'indexation utilisée est Lucene. Il s'agit de l'index par défaut fourni par Neo4j, c'est-à-dire Lucene. Celui-ci n'est pas prévu pour des valeurs numériques et ne prend pas en compte la position des entités dans l'espace.

D'autres études cherchent à tester les performances des requêtes spatiales. Il est possible d'améliorer le temps de recherche du voisin le plus proche de trois façons. Tout d'abord, le processus est plus rapide dans un espace ayant peu de dimensions. Ensuite, une indexation des éléments géométriques permet de diminuer le temps de requête. Enfin, ils montrent que le temps nécessaire à la sélection d'un point le plus proche n'est pas lié au nombre de points totaux présents dans la base de données. (BEIS, *et al.* 1997)

Il existe aussi des indexations spatiales spécifiquement développées pour être utilisées sur des graphes. Le Gtree est l'un d'eux. Il permet un gain important de performance des requêtes spatiales sur un réseau routier (ZHONG, *et al.* 2015).

3.Hypothèse

La littérature met en avant des solutions possibles aux bases de données relationnelles pour réaliser des requêtes spatiales. Une de ces solutions est d'utiliser une base de données NoSQL orientée graphe. Celle qui est la plus mise en avant par les recherches précédentes est la base de données graphe proposée par la plateforme Neo4j. Des études comparatives ont été réalisées entre Neo4j et une base de données relationnelles et montrent un résultat mitigé. En effet, la requête du plus court chemin est la majorité du temps plus performante sur la base de données graphe mais les autres requêtes spatiales n'ont pas de meilleures performances. Une piste serait d'améliorer la configuration de cette base de données pour obtenir des résultats plus rapides que sur une SGBD-R.

La différence de configuration proposée dans ce mémoire est de spatialiser les données. Cela signifie que les nœuds de la base de données auront une géométrie et une position relative connue. De plus, les données pourront être soumises à une indexation spatiale. Pour rappel, une indexation spatiale permet de créer un arbre qui sélectionne les données suivant leur position dans l'espace.

Le choix des requêtes spatiales se base sur les résultats obtenus dans la littérature ainsi que le fonctionnement des requêtes. Deux requêtes seront de ce fait analysées. La première est le plus court chemin entre deux nœuds, car il est intéressant d'analyser les résultats afin d'observer l'influence de la variation de configuration sur un élément connu de la littérature. La seconde requête est la sélection d'éléments proches d'un point extérieur au graphe. Il s'agit d'une requête dont la vitesse d'exécution est proche pour les deux bases de données dans la littérature et ayant un avantage théorique pour l'utilisation d'index spatial.

En basant ma recherche sur ces éléments, on peut donc exprimer une hypothèse. **La référenciation spatiale des points rend la vitesse d'exécution d'une requête du plus court chemin entre deux points extérieurs au graphe inférieure, à celle de PostGIS, dans un grand réseau routier sur Neo4j.**

4.Configuration et traitement préalable

4.1. Choix d'un graphe OSM et d'un échantillon

Le réseau routier utilisé est le réseau fourni par OSM, « Open Street Map ». Ce sont des données conçues par les utilisateurs et contrôlées par des personnes compétentes. Celles-ci sont disponibles en ligne gratuitement, mises à jour régulièrement et complètes pour les régions les plus urbanisées (LUDWIG, *et al.* 2011). En plus de fournir la géométrie, le réseau OSM fournit le type de route (*residential*, *unclassified*, *secondary*, *primary*, *motoway*). Le réseau OSM donne également des informations sur le sens de circulation des routes (ex. route à sens unique). Les sens uniques sont pris en compte en utilisant un coût aberrant pour le sens concerné de l'arc.

Certaines études ont constaté aussi des variations dans le contenu du réseau OSM d'une base de données à une autre. Ces variations sont dues à la façon dont les données OSM sont importées. En effet, il est possible d'obtenir un réseau légèrement différent lorsque les données sont directement intégrées dans la base de données graphe Neo4j et lorsqu'elles sont téléchargées avec le module OSM2PO (MOELLER s.d.), puis importées dans PostgreSQL (BAAS, *et al.* 2012).

Dans ce mémoire, les premières recherches sont réalisées en utilisant une partie du réseau belge afin de faire les tests de requêtes (configuration, intégration et début de requête). Cette étude se fait sur un grand réseau, le réseau est donc étendu à l'ensemble de la Belgique. Le réseau routier est importé d'OpenStreetMap (OSM), il s'agit d'un large réseau puisqu'il y a environ 500 000 nœuds et 600 000 arcs importés. Les données sont dans un système de coordonnées WGS 84 (EPSG : 4326).

Le graphe importé possède plusieurs attributs utiles : id, osm_id, osm_source_id et osm_target_id, code, fclass, distance en km, les coordonnées x1, y1 et x2, y2 et la géométrie. L'identifiant est spécifique à la couche de données et varie d'un jeu à l'autre. Les identifiants OSM sont ceux utilisés dans leur base de données. Il peut donc permettre de retrouver l'objet dans leur base de données, mais il n'est pas forcément unique car un objet sur la base de données OSM peut avoir différentes géométries. Le code et la classe sont utilisés pour identifier le type d'objet dont il s'agit. Cela permet par la suite de déterminer la vitesse à laquelle les routes sont parcourues et de fixer un coût de parcours de la géométrie qui les représente (RAMM 2015).

4.2. Configuration de la comparaison

L'élément principal de ce mémoire est de comparer le temps nécessaire pour faire les requêtes sur des bases de données différentes. Il faut cependant préciser qu'il s'agit d'une comparaison relative. En effet, l'article de (MILER 2014) compare les différences de performances entre Neo4j et PostGIS en faisant varier les configurations et la température du disque dur. Il en résulte que les mesures du temps d'exécution ne sont pas seulement dépendantes des performances des bases de données et des requêtes. Les performances des requêtes sont aussi dépendantes des performances de l'ordinateur et d'autres éléments qui influencent la vitesse du disque dur. Les temps mesurés dans ce mémoire sont

donc pris le même jour, à un court intervalle de temps et sur le même ordinateur. La machine utilisée possède la configuration suivante : un processeur Intel I7 avec 3.2 GHz, 12 Go de mémoire RAM et les opérations sont faites sur un disque dur HDD de 1 Terra.

D'autres éléments justifient qu'il s'agit d'une comparaison relative : le réseau et la version de Neo4j. Bien qu'il soit possible de réimporter exactement le réseau, il est toujours possible qu'il ait subi des modifications. Dans le cas de Neo4j, il s'agit d'une plateforme en plein essor, il y a donc fréquemment des mises à jour qui peuvent avoir des impacts plus ou moins importants sur les résultats obtenus. Les mesures ont été faites sur la version 3.5.0 de Neo4j. Les nombreuses extensions et plug-ins utilisés ont aussi des versions et leurs fonctionnements peuvent varier lorsqu'ils sont mis à jour. L'extension principale qui permet de réaliser la configuration spatiale de Neo4j est la version 0.24 de Neo4j Spatial.

Il est, de ce fait, impossible, pour des raisons techniques, de comparer directement les résultats obtenus dans ce mémoire à d'autres résultats obtenus sur des machines différentes.

4.3. Implémentation et traitement préalable sur PostGIS

Pour télécharger le réseau provenant d'OSM et les importer dans PostGIS, on utilise l'outil OSM2PO (Boston Geographic Information Systems 2019). Un fichier est obtenu et est intégré sur PostgreSQL grâce à une fonction psql[6] dans l'invite de commande. Cette commande permet d'intégrer directement le résultat d'OSM2PO dans la base de données PostgreSQL.

La base de données PostgreSQL doit être préalablement configurée en installant les extensions PostGIS et PgRouting. Une fois les données importées, le réseau est modifié afin de pouvoir l'utiliser pour les requêtes voulues.

Tout d'abord, le graphe est importé sur une couche. On peut sélectionner par la même occasion la partie du graphe qui nous intéresse et les coordonnées peuvent être transformées dans le système de coordonnées voulu. Dans ce cas-ci, l'ensemble du réseau est utilisé et les coordonnées sont conservées en WGS 84. La table est ensuite modifiée pour permettre le calcul du plus court chemin. Pour cela, le graphe possède le type de route correspondant à chaque arc et à chacun de ces types, il est possible d'associer une limitation de vitesse. Les coûts de parcours le long du segment sont calculés en fonction du type de route auquel correspond une limitation de vitesse et grâce à la longueur de l'arc, fournie par OSM. Dans le cas de sens unique, une valeur aberrante est fournie afin qu'elle ne soit pas utilisée par le plus court chemin. Cette valeur permet que si le plus court chemin doit passer par un arc dans un sens interdit, le coût devienne si élevé qu'il est toujours plus intéressant d'emprunter plusieurs autres arcs.

Une deuxième table est créée, dans PostGIS, avec les nœuds du réseau. Les coordonnées de chaque nœud sont calculées. Le réseau est donc configuré et peut être utilisé pour réaliser les requêtes du plus court chemin et de sélection d'un élément proche d'un point extérieur au graphe. Ce réseau est exporté en CSV pour être importé ensuite dans Neo4j. Le fichier CSV ne peut pas être importé directement dans Neo4j car des vérifications sont nécessaires sur le format des données. Les données

[6] https://docs.postgresql.fr/9.4/app-psql.html

doivent d'abord être encodées en UTF-8 pour permettre l'importation. Il est aussi préférable d'utiliser un point comme séparateur de décimale et une virgule entre les différentes données. Une fois ces différents éléments vérifiés, le fichier correspond à la configuration par défaut pour l'importation dans Neo4j.

4.4. Implémentation sur Neo4j

4.4.1. Configuration de Neo4j

Neo4j est disponible de deux manières : grâce à une interface ou à l'invite de commande. La plus simple est l'interface, car elle fournit des informations et des requêtes plus ou moins complexes pour la visualisation des données.

La première étape est de créer un projet sur lequel est stockée la base de données. La seconde étape est d'ajouter un graphe sur ce projet. Le graphe est la base de données. On fournit un nom, un mot de passe et on peut aussi choisir la version de Neo4j. Une fois la base de données créée, il est possible de démarrer le serveur et d'accéder à l'onglet de gestion du graphe (Figure 10).

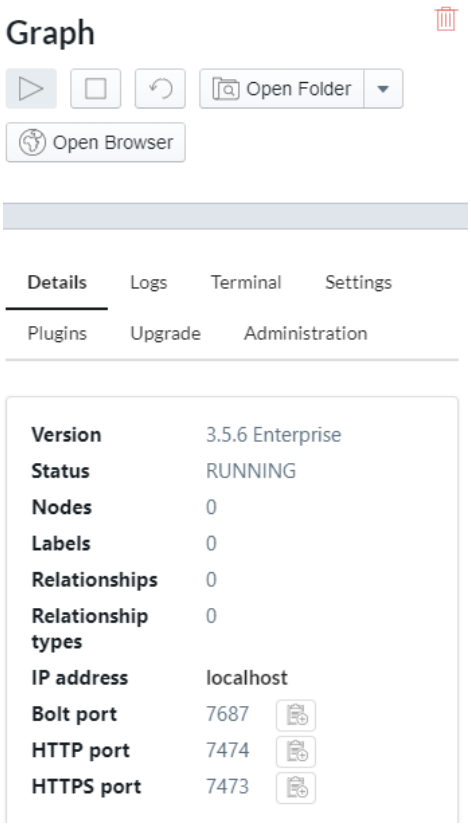


Figure 10 : Interface de gestion d'une base de données Neo4j

Plusieurs informations sont disponibles dans cette interface comme le port d'accès à la base de données, le statut de celle-ci ou encore les erreurs qui peuvent avoir eu lieu dans l'onglet « logs ». L'interface fournit aussi des accès rapides pour installer certains plug-ins, pour mettre à jour la base de données ou simplement faire des requêtes grâce à l'invite de commande.

La troisième et dernière étape est d'accéder au navigateur. Il s'agit de l'interface qui permet de réaliser les requêtes et d'afficher beaucoup d'informations sur le contenu de la base de données (Figure 11).

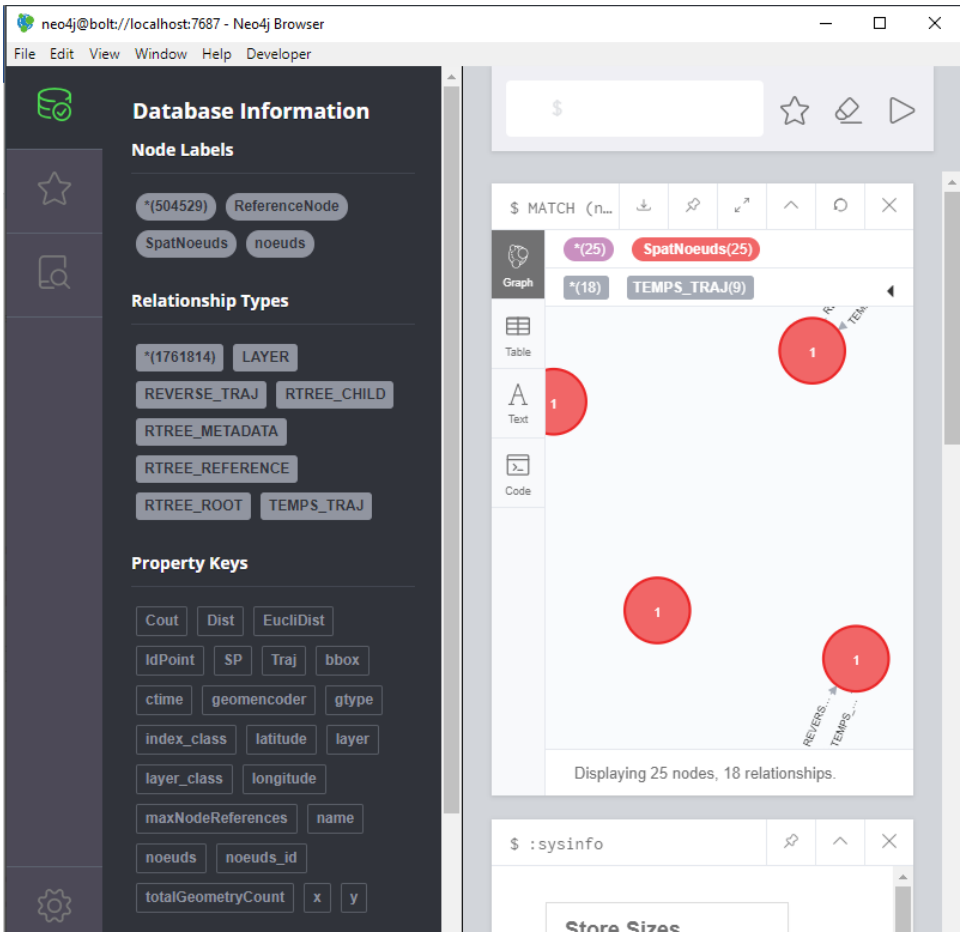


Figure 11 : Navigateur de Neo4j et informations disponibles sur le contenu de la base de données

L'onglet à gauche (Figure 11) indique les informations sur la base de données dont le nombre total de nœuds et les différents types de nœuds. Les mêmes informations sont présentes pour les arcs. Enfin, la liste des attributs présents dans la base de données est aussi disponible. Chaque type et attribut sont sélectionnables et génèrent automatiquement une requête de visualisation des données correspondantes.

4.4.2. Intégration des points dans Neo4j

Comme expliqué précédemment Neo4j possède une documentation complète, ce qui permet de réaliser facilement les requêtes.

Les données sont donc récupérées directement de PostGIS en format CSV afin de minimiser la différence entre les réseaux routiers. Ensuite, une requête (Figure 12) permet d'intégrer des nœuds avec plusieurs labels et attributs sur Neo4j sans l'extension spatiale. Les nœuds sont ajoutés avec l'identifiant OSM et leurs coordonnées en longitude et latitude.

```
USING PERIODIC COMMIT 800
LOAD CSV WITH HEADERS FROM "file:///Spat_Bnode.csv" AS row
CREATE (n:SpatNoeuds) SET n=row,
n.longitude = toFloat(row. longitude),
n.latitude = toFloat(row. latitude)
```

Figure 12 : Requête d'intégration des nœuds sur Neo4j

Ensuite, les arcs sont construits entre les différents nœuds (Figure 13) grâce à leurs identifiants OSM. On fournit à chacun une étiquette dont l'attribut est le coût nécessaire au parcours de cet arc. Il s'agit bien d'arcs et non d'arêtes puisqu'ils sont tous caractérisés par un nœud de départ et un nœud d'arrivée. Les arcs sont réimportés dans l'autre sens en utilisant « REVERSE_TRAJ » comme étiquette et le coût de trajet dans l'autre sens comme attribut de coût. Ceci permet que les deux sens de circulation soient pris en compte lors de la réalisation du plus court chemin.

```
LOAD CSV WITH HEADERS FROM "file:///Vgraphe_db.csv " AS line
MATCH (Source:noeuds {noeuds_id: line.source})
MATCH (Target:noeuds {noeuds_id: line.target})
CREATE (Source)-[r:TEMPS_TRAJ]->(Target)
SET r.Cout = toFloat(line.cost_min);
```

Figure 13 : Requête d'intégration des arcs sur Neo4j

Après l'intégration des données, la base de données Neo4j sans l'extension spatiale est configurée. C'est à ce stade que les requêtes peuvent être réalisées. Les étapes suivantes ne sont pas appliquées dessus puisqu'il s'agit de la configuration de l'aspect spatial de la base de données. Cette base de données Neo4j est similaire à celle qui est utilisée dans la littérature et permet de vérifier les conclusions tirées dans l'état de l'art. Malgré le fait que certaines requêtes soient trop différentes pour être comparées, pour d'autres requêtes, elle fournit une base de comparaison afin de mettre en avant les différences apportées par l'aspect spatial. Les deux bases de données Neo4j sont donc confrontées pour la sélection du nœud le plus proche et pour la réalisation du plus court chemin.

4.4.3. Extensions Neo4j et applications

Cette étude se base sur une modification de la configuration de Neo4j en réalisant une spatialisation différente des nœuds. C'est-à-dire que chaque nœud doit avoir sa géométrie, ce qui permet par la même occasion d'appliquer une indexation spatiale des nœuds. Pour cela, on utilise un plug-in ayant été développé par la communauté de Neo4j : Neo4jSpatial. Celui-ci est assez complet puisqu'il permet l'intégration et la gestion de données géospatiales ainsi que la réalisation d'opérations spécifiques à ces données. Il n'est pas tout à fait abouti, mais il met à disposition beaucoup d'outils de requête spatiale.

L'intégration du plug-in à Neo4j est simple puisqu'il suffit de télécharger le fichier .jar fourni, de l'ajouter dans le dossier plug-in de Neo4j et de redémarrer le serveur (Neo4j contrib 2017). Les

requêtes disponibles sont décrites dans une documentation sur le site de Neo4j, mais il est aussi possible d'obtenir une liste exhaustive grâce à la fonction « *CALL spatial.procedure* ». Cependant, l'extension est complexe à utiliser car les fonctions sont, pour le moment, peu détaillées et en même temps exigeantes au niveau du type et du format des inputs. Tout d'abord, il faut que les données soient intégrées sur la couche spatiale générée par le plug-in (MEDEIROS 1994). Cette couche spatiale doit être créée mais elle ne peut contenir que des nœuds et non les arcs du graphe. Il y a plusieurs outils d'importation de nœuds sur les couches spatiales. Le choix de l'outil dépend surtout du format des données disponibles. Les arcs peuvent être construits entre ces nœuds mais ils ne sont pas spatialisés, ce qui est un élément important pour la suite de ce mémoire.

Pour la visualisation des données, la technique utilisée est d'exporter les nœuds dans un format CSV et d'utiliser des techniques de cartographie en ligne Javascript telle que Leaflet. Une autre technique de cartographie est la mise en relation de Neo4j et de Géoserver. Géoserver est un serveur cartographique permettant la visualisation et la modification de la symbologie appliquée aux données.

Deux autres plug-ins sont disponibles et proposés directement sur l'interface de Neo4j :

- Le premier est « *Graph algorithms* [7] ». Il permet de réaliser des opérations en prenant en compte des valeurs métriques dans le graphe. Dans le cas de cette recherche, il fournit aussi des algorithmes du plus court chemin, toujours basés sur celui de DIJKSTRA (Neo4j 2019). Enfin, il propose aussi une alternative à l'algorithme de DIJKSTRA avec une requête du plus court chemin basée sur l'heuristique A*.
- Le second est « APOC [8] ». Il n'est pas utilisé durant cette recherche mais il a été testé pour exporter les données en Json pour permettre la visualisation. Le résultat n'était pas concluant et une autre technique a été utilisée. Il offre un large éventail de possibilité, par exemple, il propose une fonction pour l'importation et l'exportation de données dans des formats spécifiques comme le Json. Il propose aussi des requêtes comme un plus court chemin avec l'algorithme de DIJKSTRA (Neo4j 2019).

4.4.4. Intégration de l'extension spatiale et spatialisation

L'étape suivante est d'ajouter les nœuds sur la couche spatiale de la base de données. Dans le cas présent, la fonction utilisée nécessite que les nœuds soient déjà présents dans la base de données. Ils sont donc préalablement importés grâce à une requête (Figure 12) puis ils sont intégrés sur la couche spatiale, qui leur fournit aussi une géométrie (Neo4j contrib 2017).

```
MATCH(n:SpatNoeuds)
WHERE NOT (n)<-[:RTREE_REFERENCE]-()
CALL spatial.addNode("geom_noeuds",n)
YIELD node RETURN node LIMIT 1000
```

Figure 14 : Requête d'intégration des nœuds sur la couche spatiale

[7] https://neo4j.com/docs/graph-algorithms/current/
[8] https://neo4j-contrib.github.io/neo4j-apoc-procedures/#overview

Cette requête (Figure 14) utilise les coordonnées en longitude et latitude des nœuds pour ajouter les nœuds à la couche spatiale et créer la géométrie dans la base de données. Elle applique par la même occasion un index spatial Rtree sur les données. Cependant, la requête est longue et le temps d'exécution augmente encore avec le nombre de nœuds considéré par la requête. Un programme Python permet d'automatiser la spatialisation en réalisant une boucle avec un petit nombre de nœuds.

4.4.5. Packages et programmation Python

L'ensemble des étapes qui sont réalisées dans la suite du mémoire sont codées en Python, afin de réaliser plusieurs répétitions et de pouvoir l'appliquer sur différents points extérieurs au graphe. Différents packages sont utilisés pour automatiser l'analyse :

- Le plus important est py2neo. Il existe plusieurs packages pour mettre en relation Python et Neo4j. On utilise py2neo car il s'agit du package le plus complet et le mieux documenté (NIGET 2011-2019). Ce package permet de se connecter à la base de données Neo4j. Il fournit des outils d'édition d'un graphe grâce à des requêtes directes sur la base de données. Enfin, le package récupère le résultat des requêtes et l'affiche suivant le format voulu, dans ce cas, sous forme d'une table.
- Le package « *time* » permet de mesurer le temps durant lequel le programme est en cours. En faisant appel à ce package avant et après chaque requête qui nous intéresse, il est possible de connaître le temps nécessaire à leur réalisation. Le temps total de réalisation du code est aussi mesuré pour les comparaisons entre les bases de données.
- D'autres packages permettent de manipuler les données. De cette façon, on utilise le package CSV pour l'importation et l'exportation des données. Les packages « *pandas* » et « *numpy* » permettent de structurer en matrice les données récupérées par py2neo afin de simplifier leur manipulation. Le package math permet de réaliser les calculs de la détermination du point du graphe le plus proche.
- Enfin, psycopg2 est un package utilisé pour faire la relation entre Python et PostGIS. Il permet de se connecter à la base de données et, comme py2neo, il peut réaliser des requêtes et récupérer le résultat dans Python.

4.5. Définition des paramètres du graphe dans Neo4j

La structure du graphe dans Neo4j est inhérente à celle d'un réseau routier. Les intersections entre les routes sont représentées par des nœuds. Après avoir été importés, les nœuds possèdent plusieurs attributs : un identifiant, un identifiant OSM et des coordonnées en longitude et latitude en WGS 84. Les arcs représentent les routes du réseau. Ces arcs sont créés en reliant les nœuds grâce à leur identifiant OSM. Elles ne sont donc pas prises en compte comme des droites par la plateforme Neo4j mais uniquement par des points de départ et d'arrivée ainsi qu'un attribut de coût qui leur a été attribué.

Grâce à la configuration de l'extension spatiale, les nœuds possèdent une enveloppe appelée « bbox », en plus de leur identifiant et des coordonnées en longitude et latitude. Les arcs ne sont pas influencés par la nouvelle configuration puisqu'ils sont toujours définis par les coordonnées du nœud de départ

et d'arrivée. Cette propriété est prise en compte par la suite pour la détermination du point du graphe le plus proche.

Deux graphes Neo4j sont comparés dans cette étude. Le premier possède une structure de base de données similaire à celle utilisée dans la littérature. C'est-à-dire que les coordonnées ne sont que des attributs, les nœuds n'ont pas de géométrie et l'index utilisé est l'index Lucene, fourni par défaut avec Neo4j. Le second graphe est réalisé grâce à l'extension spatiale de Neo4j : Neo4j Spatial. Les nœuds ont une position dans l'espace et ils sont structurés grâce à un index Rtree.

5.Analyse

5.1. Rattachement d'un point extérieur au graphe

5.1.1. Insertion d'un point hors graphe

Les points extérieurs au graphe sont insérés dans la base de données Neo4j de la même façon que les nœuds. Les coordonnées du point à ajouter sont écrites dans un fichier CSV (Figure 15) et importées grâce à la même requête que pour l'insertion de la totalité des nœuds (Figure 12).

```
LOAD CSV WITH HEADERS FROM
'file:///PtsExt.csv' AS row CREATE
(n:PtsExt) SET n=row,n.x =
toFloat(row.x),n.y =
toFloat(row.y)
```

Figure 15 : Requête pour l'importation des points hors graphe

5.1.2. Recherche du nœud le plus proche

La requête (Figure 15) a pour but de localiser le nœud du graphe le plus proche d'un point extérieur au graphe. Dans une application réelle, ceci correspond au cas où une personne se trouvant dans une forêt ou dans des champs cherche la ville la plus proche.

Deux méthodes sont utilisées dans ce mémoire pour sélectionner le nœud le plus proche du point extérieur au graphe. La première est la méthode qui s'applique à la base de données non spatiale. Il s'agit de calculer la distance entre le point et tous les nœuds du graphe et de les trier suivant cette distance. Cependant, cette requête a une faible performance, même lorsque les données sont soumises à une indexation spatiale. Avec cette méthode, des arcs sont créés dans la base de données pour stocker la distance entre le point extérieur et les nœuds du graphe les plus proches. Un des avantages de Neo4j est la visualisation des données. Elle permet de visualiser les nœuds les plus proches et les arcs associés. Dans le cas de la Figure 17, les 20 plus proches voisins ont été sélectionnés pour mieux illustrer la visualisation de données sur Neo4j.

```
MATCH (Source:PtsExt), (Target:SpatNoeuds)
WITH SQRT((Source.x - Target.x)^2+(Source.y -
Target.y)^2) AS proxi, Source, Target
ORDER BY proxi
LIMIT 1
MERGE (Source)-[r:Proximity]->(Target)
SET r.EucliDist = proxi
```

Figure 16 : Requête de sélection du nœud le plus proche grâce à la distance euclidienne

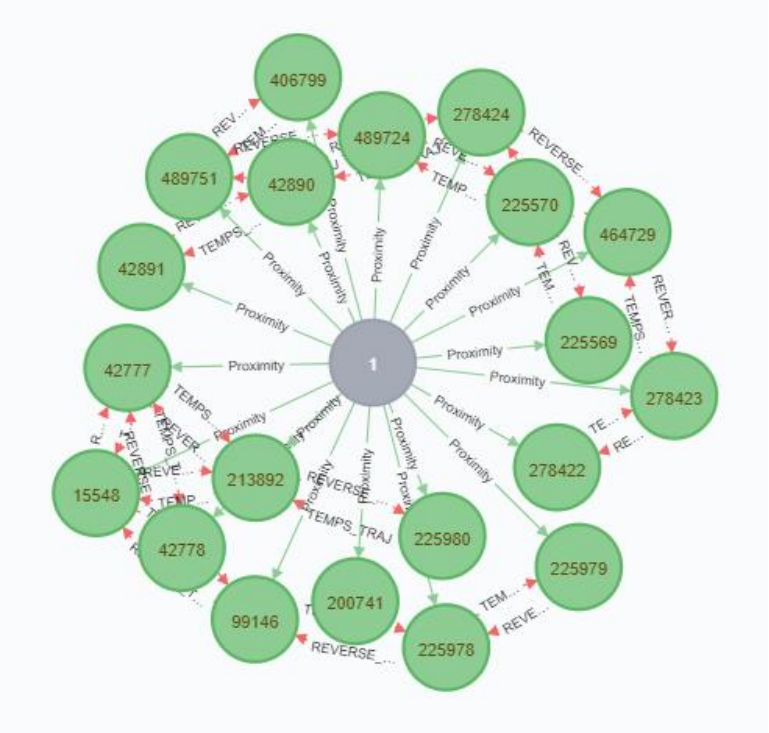


Figure 17 : Sélection de vingt nœuds les plus proches d'un point hors graphe dans Neo4j sans l'extension spatiale

La seconde méthode n'est possible que sur la base de données Neo4j Spatial car elle est fournie par l'extension spatiale. Il s'agit d'une requête de sélection grâce à la position relative du nœud le plus proche d'un point. En effet, la Figure 18 utilise l'indexation spatiale pour sélectionner les nœuds les plus proches dans une certaine zone puis détermine le plus proche. Cette requête est plus performante que l'autre mais n'est applicable que si les nœuds possèdent des géométries.

```
CALL spatial.closest('geom_noeuds',{ longitude:
"+long1+" , latitude:"+lat1+"},0.1) YIELD node
RETURN node, node.longitude, node.latitude LIMIT 1
```

Figure 18 : Requête de sélection du plus proche voisin, fournie avec Neo4j Spatial

Les résultats obtenus pour la première requête (Figure 16) sont comparés pour les deux bases de données Neo4j : l'une d'elle sans l'extension spatiale et l'autre avec. Et les deux types requêtes de sélection d'un nœud sont comparés sur la base de données Neo4j Spatial.

	Neo4j NS (ms)	Neo4j S (ms)
Nœud le plus proche*	2768,36	3067,2
Nœud le plus proche**	/	239,08

Tableau 1 : Résultat de la mesure du plus proche voisin

*Requête de sélection du nœud le plus proche en calculant la distance entre le point extérieur au graphe et chaque nœud du graphe et en sélectionnant le plus proche.

**Requête de sélection du nœud le plus proche spatialement.

Lorsqu'on observe les résultats (Tableau 1), on constate qu'en calculant la distance par rapport à chaque nœud, la base de données spatiale est légèrement plus lente que la base de données sans l'extension spatiale. Ce résultat n'est pas ce qui est attendu. Il faut se rappeler que la base de données sans l'extension spatiale utilise une indexation Lucene qui ne possède pas de notions de position relative des éléments et qui n'est pas prévue pour des valeurs numériques. De ce fait, l'utilisation de l'index spatial devait accélérer l'exécution de la première requête de sélection d'un nœud proche (Figure 16). Cependant, on observe, au contraire, une augmentation du temps nécessaire à la requête. On peut supposer que la différence de rapidité, bien que faible, est due à la surcharge d'attributs des nœuds de la base de données spatiale. De plus, l'indexation spatiale ne peut pas améliorer les performances de cette requête puisqu'elle interagit avec l'ensemble des nœuds de la base de données (Figure 16).

Lorsqu'on compare les deux types de requêtes de sélection sur la base de données Neo4j Spatial, la possibilité de sélection spatiale du nœud est plus rapide que de calculer la distance pour chaque nœud. L'utilisation de l'index spatial permet de trier les données sous forme d'un nouvel arbre. En effet, l'indexation utilise la position relative des nœuds pour former un arbre qui est utilisé lors de la recherche du nœud le plus proche. De plus, la requête spatiale n'a pas besoin de calculer la distance entre le point et tous les nœuds de la base de données, mais seulement avec les nœuds présents dans un certain rayon.

5.1.3. Recherche de l'arc le plus proche

La requête suivante est de sélectionner le point du graphe le plus proche des points importés. La différence avec la requête précédente est qu'il s'agit d'un point quelconque du graphe. Cette notion de plus proche voisin peut être variable en fonction des éléments à disposition. Dans le cas présent, on applique la requête de point le plus proche par distance euclidienne car on ne possède pas d'information sur les zones hors graphes. Cette requête a des applications similaires à la précédente mais avec moins de contraintes. Une personne qui se trouve dans une forêt, ou n'importe quel endroit en dehors du réseau routier, peut chercher à atteindre la route la plus proche.

Pour cela, on sélectionne l'arc le plus proche et on détermine ensuite quel point sur cet arc est le plus proche du point hors graphe (Figure 19). Le point de l'arc ne peut pas être directement sélectionné sur Neo4j. En effet, les arcs n'ont pas de géométrie dans la base de données, l'information est uniquement connue grâce aux coordonnées du nœud de départ et d'arrivée. La requête de sélection de l'arc le plus proche est réalisée par rapport à deux points extérieurs au graphe, pour ensuite calculer le plus court chemin entre ces points.

```
CALL spatial.closest('geom_noeuds',{ longitude: "+str(long1)+",
latitude: "+str(lat1)+"},0.05) yield node MATCH p=(node)-
[r:TEMPS_TRAJ]->(targ) MATCH o=(targ)-[s:REVERSE_TRAJ]->(node)  return
 node.longitude, node.latitude,  targ.longitude,
targ.latitude,r.Cout,s.Cout, node.noeuds_id, targ.noeuds_id limit 2
```

Figure 19 : Requête de sélection d'un arc le plus proche d'un point extérieur au graphe

La requête, ci-dessus, fournit les coordonnées et les identifiants des nœuds caractérisant l'arc (Tableau 4). Ensuite, le résultat de la requête est utilisé dans le code Python (Annexe 2) pour trouver le point de l'arc le plus proche du point hors graphe et en déterminer les coordonnées. Ce point est par la suite intégré dans le graphe, comme un nouveau nœud, pour appliquer le plus court chemin entre les deux nouveaux nœuds proches des deux points extérieurs au graphe. La requête exporte aussi les coûts de l'arc le plus proche, dans les deux sens qui seront utilisés dans la suite du mémoire, pour l'intégration des points les plus proches dans le réseau.

Pour calculer l'intersection entre le point hors graphe et l'arc le plus proche, on utilise les coordonnées des nœuds de départ et d'arrivée des arcs. Ces coordonnées sont utilisées pour déterminer l'équation de la droite sous la forme : $y = a * x + b$. Selon cette formule, a correspond à la pente de la droite et b au décalage par rapport au centre du système de coordonnées. Le chemin le plus court vers un arc est toujours perpendiculaire à celui-ci. Il est donc possible de calculer la pente du trajet le plus court vers un élément grâce à l'équation : $m' = {}^{-1}/_{m}$.

Enfin, l'intersection entre l'arc et le trajet le plus court vers l'arc est le point du graphe le plus proche du point inséré. Les coordonnées de ce point se calculent grâce aux fonctions de la Figure 20.

$$x_C = \frac{b' - b}{a - a'} \qquad y_C = a\frac{b' - b}{a - a'} + b$$

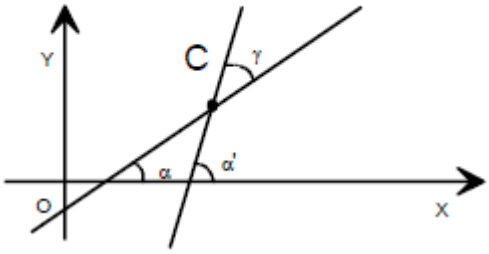


Figure 20 : Équation de calcul de l'intersection entre l'arc le plus proche et la droite la plus courte entre le point hors graphe et le graphe (DONNAY, 2019)

Une fois les coordonnées du point de l'arc connues, il doit être inséré dans le graphe comme un nouveau nœud. Ces nœuds permettent ensuite de calculer le plus court chemin entre les deux points les plus proches des points extérieurs au graphe. L'insertion d'un nœud dans Neo4j se fait facilement. Cependant, il faut recréer des arcs vers les nœuds de départ et d'arrivée de l'arc le plus proche. C'est l'opération représentée sur la Figure 21 avec les nœuds 1000001 et 1000002 qui sont les points les plus proches de deux points extérieurs au graphe.

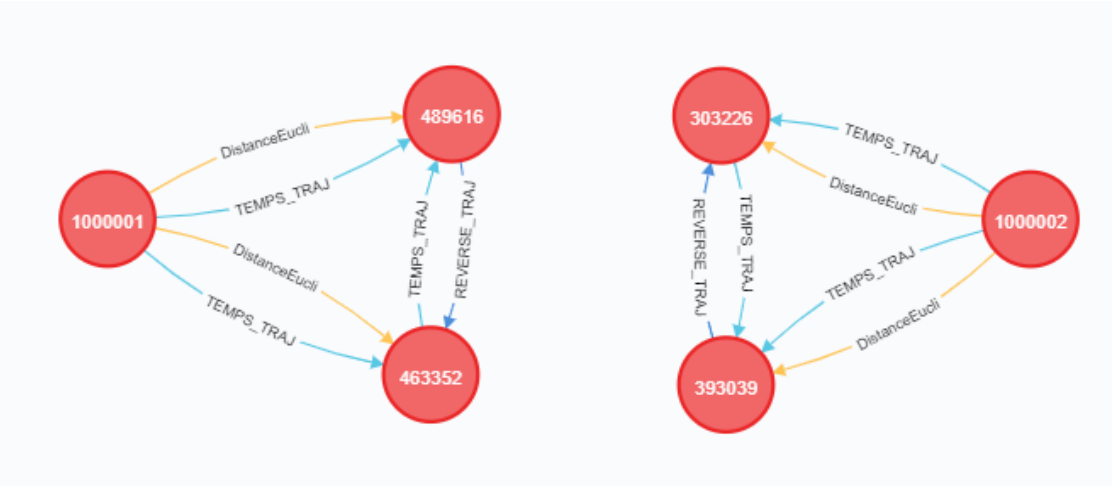


Figure 21 : Représentation Neo4j de l'insertion des nouveaux nœuds dans le réseau

Pour insérer les nouveaux nœuds dans le réseau, les informations de l'arc le plus proche sont extraites. Il s'agit des identifiants des points de départ et d'arrivée ainsi que les coûts de parcours dans un sens et dans l'autre, correspondant à l'attribut de coût des arcs « TEMPS_TRAJ » et « REVERSE_TRAJ » dans la Figure 21. Il faut ensuite calculer la longueur de l'arc en distance euclidienne et la distance euclidienne entre le point le plus proche et les extrémités de l'arc, afin de pondérer le coût des nouveaux arcs. Avec ces différents éléments, le coût de parcours des nouveaux arcs est déterminé puis les arcs et coûts associés sont intégrés dans la base de données. Dans la Figure 21, les nouveaux arcs sont les « TEMPS_TRAJ » qui relient les nœuds 1000001 et 1000002 au réseau. De cette façon, le graphe est prêt pour réaliser le plus court chemin entre deux nouveaux nœuds les plus proches de points extérieurs au graphe.

5.1.4. Résultats de la sélection du nœud et de l'arc le plus proche d'un point extérieur

Deux points extérieurs au graphe sont ajoutés dans la base de données. Leurs coordonnées sont données dans le Tableau 2. Le Tableau 3 représente la requête de sélection spatiale du nœud le plus proche pour le Point 1. Enfin, le Tableau 4 contient les informations obtenues pour la sélection de l'arc le plus proche des deux points extérieurs au graphe.

	Longitude	Latitude
Point 1	4.559136	50.132586
Point 2	5.985007	50.282731

Tableau 2: Coordonnées des points extérieurs au graphe

node.longitude	node.latitude
4.5606976	50.138491

Tableau 3 : Coordonnées du nœud le plus proche du Point 1, extérieur au graphe

	ID_OSM		Coord Départ		Coord Arrivée	
	Nœud Départ	Nœud Arrivée	Longitude	Latitude	Longitude	Latitude
Point 1	413063	413061	4.5547143	50.1368907	4.5606976	50.1384910
Point 2	190228	360253	5.9670677	50.2882347	5.9713767	50.2851894

Tableau 4 : Identifiants et coordonnées de l'arc le plus proche des points extérieurs au graphe

Les résultats sont complexes à analyser avec uniquement les valeurs des coordonnées. Cependant, on sait qu'il ne s'agit pas de valeurs aberrantes pour les éléments les plus proches puisque les coordonnées sont des valeurs proches.

Le temps de réalisation de la sélection d'un arc n'est pas comparé entre les deux bases de données graphes. En effet, les requêtes utilisées sont différentes car la base de données sans l'extension spatiale ne permet pas la sélection d'un arc spatialement. Et si les requêtes ne sont pas les mêmes, la comparaison des temps n'est pas cohérente.

5.2. Recherche du plus court chemin

5.2.1. Parcours du graphe

Après l'insertion des nouveaux nœuds les plus proches des points hors graphe, le plus court chemin peut être réalisé. Cette requête consiste à rechercher, à partir d'un nœud, la suite d'arcs qui mène à un nœud d'arrivée avec une somme des attributs de coût minimale. Les attributs des arcs sont les coûts nécessaires pour parcourir la route qu'ils représentent. Les arcs sont aussi intégrés dans le sens inverse avec un attribut de coût adapté, cela permet de prendre en compte les deux sens de circulation et la présence de sens interdit.

L'utilité de cette requête est connue puisqu'il s'agit d'un calcul d'itinéraire comme le propose Google Maps ainsi que tous les GPS de voitures ou de GSM. Dans ce cas, il s'agit de l'opération la plus simple possible puisqu'aucune information ne vient modifier le trajet proposé, en dehors du temps de parcours d'une route.

Pour réaliser une requête du plus court chemin, Neo4j possède différents algorithmes fournis par les plug-ins et avec des champs d'application variables. Les requêtes du plus court chemin peuvent prendre en compte un ou deux sens de circulation. Elles peuvent aussi avoir d'autres résultats en sortie de la requête, comme la valeur du coût total du trajet, l'ajout d'un label sur les nœuds du trajet ou encore la liste de nœuds à traverser.

L'algorithme de DIJKSTRA est la base de la plupart des requêtes, mais il est aussi possible de réaliser une requête du plus court chemin en se basant sur une heuristique A*. Dans ce cas, il n'y aura pas qu'un seul trajet proposé mais plusieurs. Dans ce mémoire, tous les plus courts chemins utilisés sont basés sur DIKSTRA afin de pouvoir comparer les résultats. Cependant, l'utilisation de Neo4j Spatial

limite les possibilités de requêtes du plus court chemin utilisables. En effet, plusieurs requêtes qui ont été testées et qui fonctionnent pour la base de données sans l'extension spatiale ne sont pas compatibles avec l'extension spatiale de Neo4j. La requête réalisée est donc celle de la Figure 22.

```
MATCH (a:SpatNoeuds {noeuds_id: '1000001'}) MATCH (b:SpatNoeuds {noeuds_id: '1000002'}) CALL
apoc.algo.dijkstra(a, b, 'TEMPS_TRAJ', 'Cout') YIELD path, weight RETURN nodes(path), length(path);
```

Figure 22 : Requête du plus court chemin entre les deux nouveaux nœuds les plus proches des deux points extérieurs au graphe sur Neo4j

La requête (Figure 22) réalise un calcul d'itinéraire entre les deux nœuds les plus proches des points hors graphe. Ces nœuds ont été préalablement insérés dans le graphe avec les identifiants 1000001 pour le nœud de départ et 1000002 pour le nœud d'arrivé. Le résultat est la suite des nœuds traversés par le plus court chemin. Certaines requêtes permettent d'obtenir le temps total nécessaire pour parcourir le trajet en faisant la somme des attributs de coût des arcs. Cependant dans le cadre de ce mémoire, la somme des attributs des arcs est peu intéressante. En effet, l'aspect des plus courts chemins qui nous intéresse est le parcours des nœuds et surtout le temps nécessaire pour leurs exécutions. Le temps calculé pour le plus court chemin peut être intéressant afin de comparer la justesse des résultats avec la requête du plus court chemin sur PostGIS mais la fonction permettant d'obtenir ce type de résultat n'est pas compatible avec l'extension Neo4j Spatial.

5.2.2. Visualisation

La visualisation est un des points forts de Neo4j. Pour la plupart des requêtes, il est possible d'observer le résultat sous forme de nœuds et d'arcs mais sans structure (Figure 17). Cependant, l'interface Neo4j limite le nombre de points observables à 300 afin de limiter l'impact sur les performances.

Neo4j possède aussi beaucoup d'applications et d'outils qui sont développés en parallèle. Une application JavaScript a été développée pour permettre une visualisation non structurée mais complète d'une requête. Il s'agit de Neovis.js qui permet aussi de classer les nœuds et de mettre en avant certains paramètres. D'autres possibilités existent aussi mais n'ont pas été testées. Par exemple, un outil a été développé pour mettre en relation Neo4j et Geoserver, ce qui peut simplifier davatange la visualisation. Il est aussi possible, grâce à un plug-in Neo4j, d'exporter les coordonnées obtenues dans différents formats.

Pour réaliser une visualisation structurée, on exporte les coordonnées des nœuds par lesquels passe notre plus court chemin et on les importe dans un code Leaflet. Leaflet est un JavaScript qui permet de réaliser des cartes sur un fond de carte choisi. Il possède aussi un package « *omnivore* » qui importe automatiquement des points provenant d'un fichier CSV (Figure 23). Chaque point importé par « *omnivore* » est directement défini comme un marker (Figure 24 et Figure 35).

```
var customIconImp = L.icon({     //Variable configurant l'icon affichée
    iconUrl: 'PtsImp.png',       //Choix de l'icon utilisée
    iconSize: [12, 20],          //Taille de l'icon en pixel
    iconAnchor: [11, 19],        //Point d'ancrange de l'icon sur la carte
    popupAnchor: [0, 0]
});

 omnivore.csv('PtsNodesTest1.csv')                  //Importation des données CSV dans omnivore
 .on('ready', function(layer) {
    this.eachLayer(function(marker) {               //Transformation de chaque point en marker

        marker.setIcon(customIconImp);              //Application de la configuration l'icon

        var popupText = 'Le point numero: '         //Message apparaissant en cliquant sur le marker
          +  marker.toGeoJSON().properties.Labels;

        marker.bindPopup(popupText) ;               // Application du message aux markers
    });
}).addTo(map)                                       //Ajout des markers à la carte
```

Figure 23 : Script permettant la visualisation du trajet du plus court chemin (Annexe 1) sur Leaflet

Les points sont donc localisés sur une carte et forment le cheminement à suivre pour avoir le plus court chemin, ce qui permet de comparer le trajet. Pour vérifier notre trajet, on le compare au planificateur le plus connu : Google Maps. Le résultat obtenu est similaire pour le plus court chemin, puisque le trajet passe principalement par les mêmes routes.

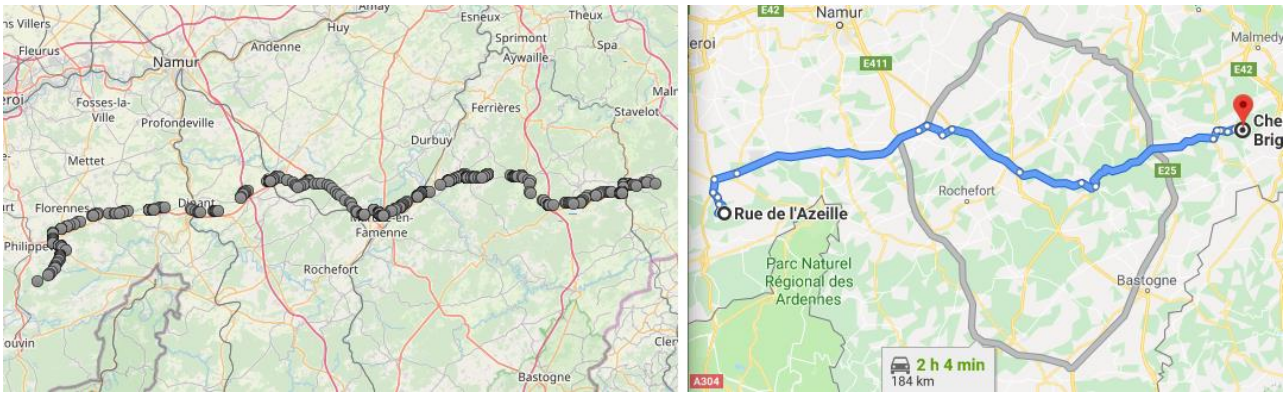


Figure 24 : Comparaison du plus court chemin réalisé avec le résultat obtenu sur Google Maps

Cependant, la comparaison est limitée car on ne connaît pas la technique de détermination du plus court chemin utilisée par Google Maps. De plus, le réseau de Google Maps et celui provenant d'OSM, configuré dans ce mémoire, ne sont pas les mêmes.

Une autre différence importante du résultat visualisé sur la Figure 25 provient des compléments d'information fournis par Google Maps. Ceux-ci permettent de modifier le trajet en fonction des travaux ou des encombrements de la circulation. Ils permettent aussi de proposer des alternatives aux autres trajets. L'ajout et la modification du réseau, en temps réel, sont possibles sur une base de données NoSQL grâce à l'absence de structure. La modification n'a cependant pas d'intérêt dans le cadre de cette comparaison. En effet, il est simple d'ajouter ou de modifier de l'information dans les modèles NoSQL. L'algorithme utilisé engendre aussi des différences de résultat des trajets. Dans ce cas, l'algorithme de DIJKSTRA fournit toujours un résultat unique et l'utilisation d'autres algorithmes, comme l'heuristique A*, permet d'avoir d'autres possibilités de résultat.

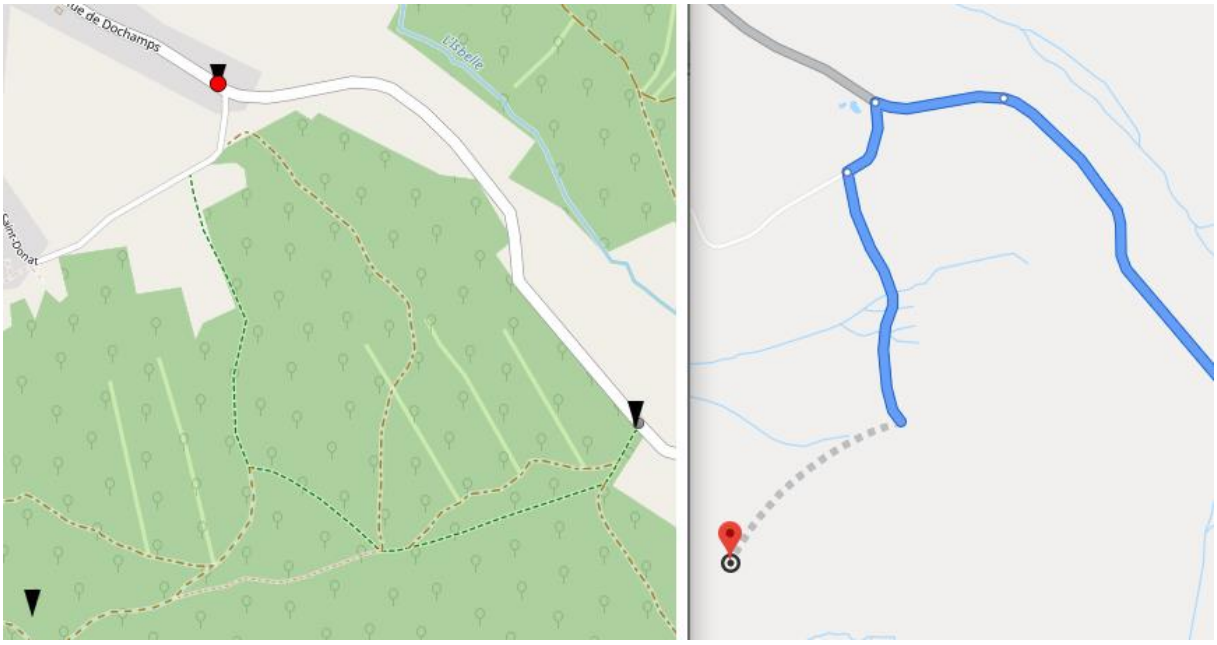


Figure 25 : Représentation de la différence entre le réseau d'OSM et le réseau de Google Map

La Figure 25 montre une différence entre le réseau d'OSM et celui de Google Maps. On observe que dans ce cas, celui de Google Maps intègre un chemin de terre à son réseau, qui n'est pas repris dans celui des bases de données.

On constate, lors de cette visualisation (Figure 24), que la requête du plus proche voisin réalisée par notre code et celle faite par Google Maps donnent des résultats différents. L'explication est que cette différence provient d'une variation entre les deux réseaux. Ces variations sont d'autant plus mises en avant puisque les points extérieurs aux graphes sont situés dans des zones non urbanisées telles que des champs et forêts. Ce sont des zones où l'on trouve le plus fréquemment des sentiers et chemins qui ne sont pas forcément repris dans les réseaux routiers. Le fait que Google Maps intègre certains sentiers peut influencer la sélection des arcs les plus proches d'un point hors graphe, puis le plus court chemin. Ces différences sont dues aux variations du réseau et justifient notamment les premières étapes de ce mémoire de configurer préalablement un réseau commun pour les trois bases de données. En effet, le réseau routier est configuré au préalable sur PostGIS avant de l'importer dans Neo4j.

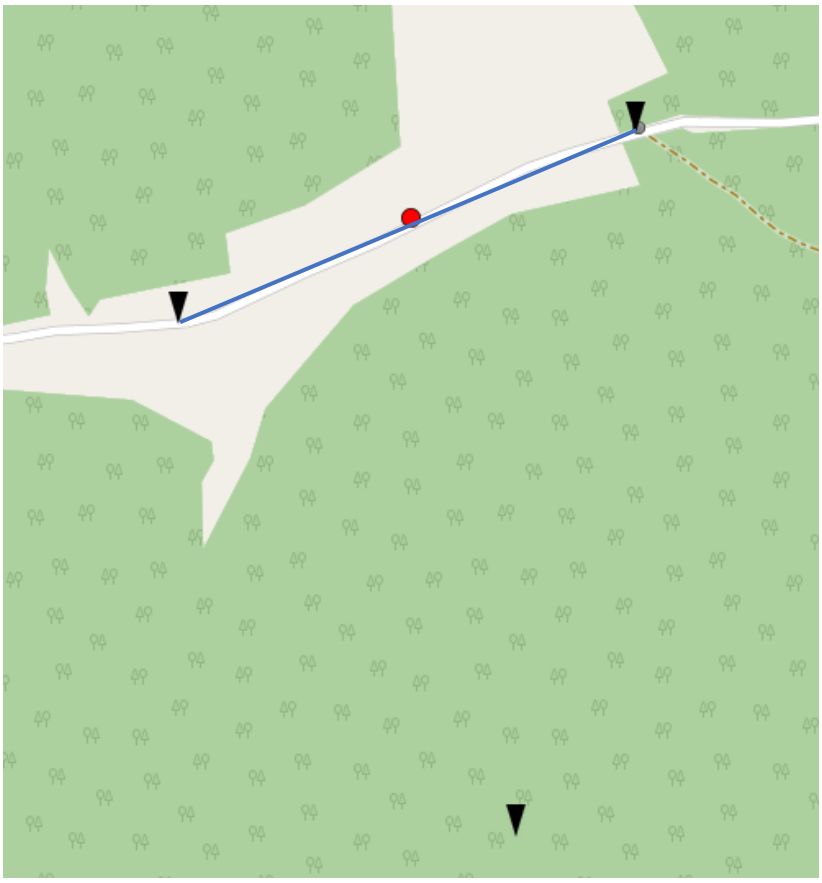


Figure 26 : Représentation de l'arc le plus proche et du nœud inséré

La Figure 26 montre une détermination efficace du point du graphe le plus proche d'un point extérieur au graphe. Le point situé dans la forêt est le point fourni initialement et est symbolisé ici par un triangle noir inversé. La requête de sélection d'un arc le plus proche du point permet de sélectionner l'arc représenté ici par la droite bleue, entre les deux triangles présents sur la route. Enfin, le calcul du point le plus proche donne comme résultat le point rouge.

Représentation Neo4j

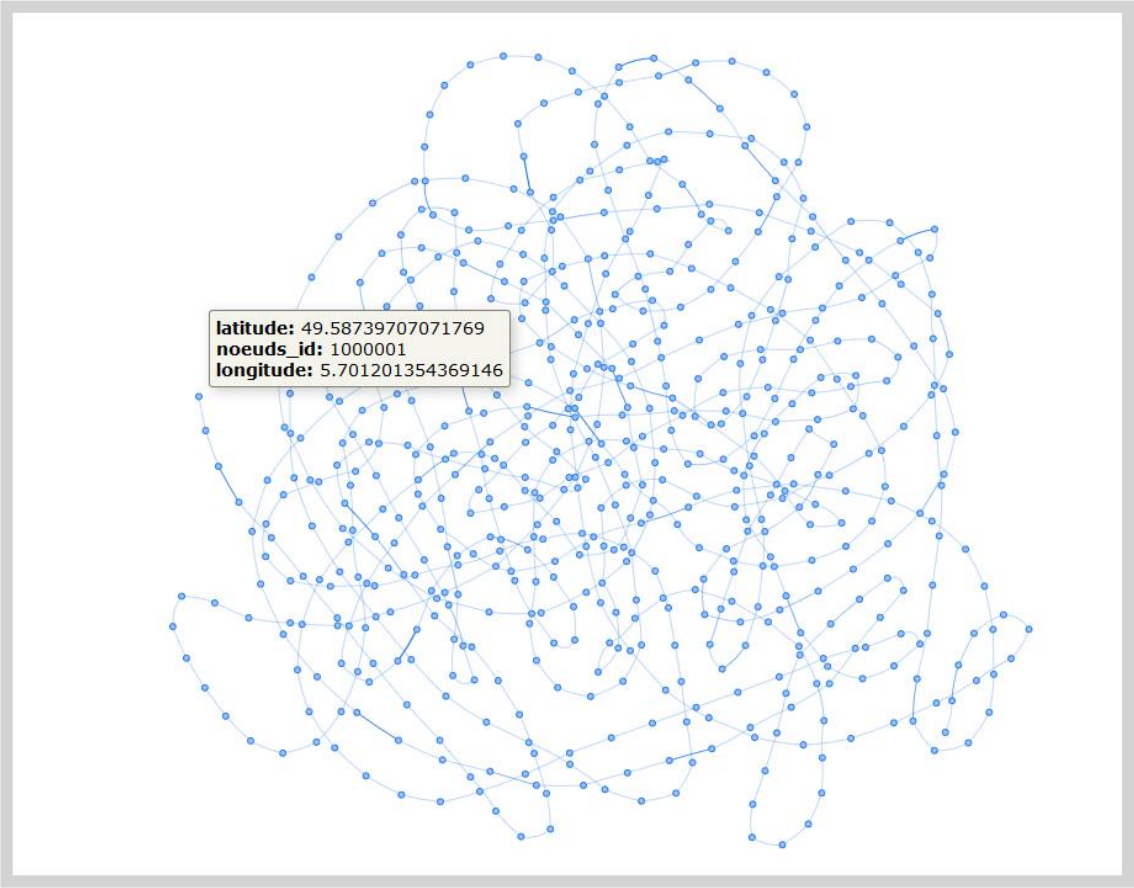


Figure 27 : Visualisation des données du trajet le plus court entre les deux nœuds grâce à Neovis.js

Enfin, la Figure 27 est une dernière représentation faite du plus court chemin calculé. Dans ce cas, on utilise un code javascript « *neovis* ». Bien que cet outil javascript soit peu abouti, il permet d'observer comment fonctionne la base de données graphe sans l'index spatial. On observe les nœuds et les arcs liés de façon non structurée. Dans cette visualisation, les nœuds et les arcs possèdent toujours leurs labels et l'épaisseur des arcs varie en fonction de leur attribut de coût.

5.3. Comparaison avec PostGIS-pgRouting

Actuellement, l'un des modèles relationnels avec aspect spatial les plus utilisé est PostGIS. Il s'agit de l'extension spatiale de la base de données relationnelle PostgreSQL. L'outil pgRouting permet de réaliser un plus court chemin suivant l'algorithme de DIJKSTRA. Il est, dans ce cas, appliqué entre deux nouveaux nœuds, les plus proches des deux points extérieurs au graphe. Cette comparaison permet de savoir si Neo4j peut être plus efficace, en termes de temps d'exécution, que pgRouting.

Pour faire la comparaison entre les deux bases de données, des codes Python sont réalisés avec exactement les mêmes opérations. Il est important de savoir qu'il s'agit bien d'une comparaison relative, c'est-à-dire qu'il n'est pas possible de comparer directement les résultats obtenus avec les

résultats d'une autre recherche. En effet, la puissance de l'ordinateur, sa configuration et la température du disque dur sont des éléments qui justifient le fait que ce soit une comparaison relative. Ensuite, la configuration des bases de données, les requêtes utilisées ou encore le réseau routier peuvent être semblables mais il est toujours possible qu'il y ait de légères différences par rapport à celle utilisée dans ce mémoire.

5.3.1. Configuration et manipulation dans PostgreSQL

Comme expliqué précédemment, les données sont importées dans PostgreSQL avant de passer dans Neo4j. Cette étape permet d'avoir les mêmes informations dans les graphes des deux bases de données.

Il y a une différence importante entre le fonctionnement des deux bases de données. En effet, étant une SGBD-R, PostGIS fonctionne avec deux tables. La première table est celle qui a été importée d'OSM. Elle contient l'ensemble des droites du graphe. Chaque droite est un tuple de la table et représente une route. La seconde table contient tous les nœuds du graphe. Chaque nœud est un tuple de cette table et il représente une intersection du réseau routier.

Il est nécessaire d'appliquer une indexation spatiale pour que la base de données PostGIS puisse être comparée à la base de données Neo4j. Pour cela, les contraintes éventuelles sont retirées des tables puis le nouvel index est créé sur la table des arcs et des nœuds, en fournissant la géométrie comme paramètre (Figure 288).

```
CREATE INDEX spatial_nodeid
  ON cost_roads.osm_node
  USING GIST (the_geom);
```

Figure 28 : Requête de création de l'indexation spatiale sur PostGIS

La requête (Figure 28) crée une indexation Rtree+ sur la base de données PostGIS. Il ne s'agit pas exactement de la même indexation que sur la base de données Neo4j Spatial mais dans ce cas, la différence n'influence pas les performances. La différence entre l'indexation Rtree et Rtree+ engendre une variation dans l'arbre d'indexation, ce qui pourrait influencer ce mémoire. Cependant, l'indexation est réalisée uniquement sur les nœuds et lorsque les enveloppes des entités sont calculées, aucune ne se superpose. Il n'y a donc pas de différence entre l'arbre d'indexation du Rtree et du Rtree+.

La suite des opérations est réalisée à travers une programmation en Python. Les étapes sont les mêmes que celle de Neo4j afin de pouvoir comparer le temps total de réalisation du code. Une différence provient du package qui met en relation PostGIS et Python. Celui-ci est différent du package qui met en relation Neo4j et Python. Leur fonctionnement est cependant similaire puisqu'ils permettent tous les deux d'exécuter une requête directement sur la base de données. Les requêtes utilisées sont aussi différentes puisqu'il ne s'agit pas du même langage de requête. Pour rappel, le langage de requête utilisée dans Neo4j est le Cypher alors que sur PostGIS, il s'agit du SQL. Néanmoins, la différence entre les deux langages n'est pas trop importante puisque le Cypher a été créé en se basant sur le SQL et les différences sont minimisées pour cette recherche. PostGIS possède une requête qui permet de sélectionner directement un point sur une droite, qui est le plus proche du point hors graphe. Cependant, cette requête n'est pas présente sur Neo4j ou Neo4j Spatial. Pour que la comparaison

puisse être faite, on choisit une requête disponible sur Neo4j et sur PostGIS : la sélection de l'arc le plus proche d'un point extérieur au graphe (Figure 29).

```
SELECT
sT_Distance(osm_bgraphe.geom_way,st_setsrid(st_geomfromtext('POINT(3.08441 51.2516)'),4326))/1000.0 AS distance_km,
 osm_bgraphe.x1, osm_bgraphe.y1, osm_bgraphe.x2, osm_bgraphe.y2, osm_bgraphe.cost, osm_bgraphe.reverse_cost
FROM cost_roads.osm_bgraphe, cost_roads.ptsext
ORDER BY distance_km
LIMIT 1
```

Figure 29 : Requête de sélection de l'arc le plus proche d'un point hors graphe sur PostGIS

La requête est appliquée pour les deux points extérieurs au graphe. Les points de l'arcs sont ensuite calculés dans le code Python afin d'être intégrés dans la table de nœuds de PostGIS. La même méthode que sur la base de données Neo4j est appliquée pour définir le nouveau coût de trajet entre le nouveau nœud et les extrémités de l'arc le plus proche. Une fois les deux nouveaux nœuds insérés dans le réseau, il est possible de réaliser le plus court chemin (Figure 3030).

```
SELECT seq, id1 AS node, id2 AS edge, cost
FROM pgr_Dijkstra ('SELECT id, source, target, cost FROM cost_roads.osm_Bgraphe', 1000001, 1000002, false, false);
```

Figure 30 : Requête du plus court chemin sur PostGIS

Le trajet choisi par le plus court chemin est visualisé afin de vérifier que le résultat soit comparable avec le trajet calculé sur Neo4j. Pour cela, on utilise la relation entre PostGIS et Qgis. Elle permet d'afficher les couches de la base de données relationnelles, ainsi que le résultat de la requête du plus court chemin.

5.3.2. Critères de comparaison

La comparaison entre les différents modèles se fait à travers plusieurs éléments. Pour rappel, les bases de données comparées sont une base de données Neo4j sans l'extension spatiale similaire à celle de la littérature, une base de données Neo4j avec le plug-in spatial et une base de données relationnelle PostgreSQL qui possède aussi une extension spatiale : PostGIS.

Pour comparer ces bases de données, on analyse quatre éléments :

- La sélection du nœud du graphe le plus proche d'un point extérieur au graphe. Il s'agit d'une requête SQL simple qui sélectionne le nœud le plus proche spatialement. Cette requête a un caractère spatial important qui permet d'exploiter la géométrie des données et d'utiliser l'indexation spatiale.
- La sélection de l'arc du graphe le plus proche du point extérieur au graphe. Il faut prendre en compte la façon dont les arcs sont interprétés dans Neo4j. Ceux-ci n'ont pas de géométrie mais chaque arc est spatialisé par son nœud de départ et d'arrivée. La sélection se fait donc sur ces nœuds, par rapport à une certaine distance autour du point extérieur au graphe.
- Le plus court chemin entre deux nœuds insérés sur les arcs les plus proches des deux points extérieurs au graphe. La comparaison du plus court chemin sert à vérifier si l'hypothèse est correcte. En effet, on déduit de l'état de l'art que le parcours du graphe dans Neo4j sans l'extension spatiale est plus rapide que dans PostGIS. Notre hypothèse est donc qu'une base

de données Neo4j Spatial permet d'accélérer le plus court chemin ainsi que les requêtes de sélection des éléments les plus proches de points extérieur au graphe.

- Le temps total pour faire l'exécution de l'ensemble des opérations. Les valeurs du temps d'exécution total sont obtenues en faisant la moyenne de plusieurs itérations. Des itérations sont faites au préalable mais elles ne sont pas prises en compte le temps dans le calcul de la moyenne. Ces itérations permettent de ne pas avoir un temps supplémentaire dû au temps nécessaire pour que le disque dur préchauffe.

La comparaison du temps pour exécuter l'ensemble des opérations (Figure 31) est réalisable car celles-ci sont les mêmes pour les différentes bases de données. Il y a, tout d'abord, l'importation des points extérieurs aux graphes. Ensuite, on sélectionne les nœuds les plus proches de ces points, puis les arcs les plus proches. Sur ces arcs sont calculés les points les plus proches des points hors graphe. Ceux-ci sont intégrés dans les graphes sous forme de nœuds. Pour cela, on doit retrouver le poids des nouveaux arcs entre ce nouveau nœud et les extrémités de l'arc le plus proche. Le calcul des nouveaux arcs se base sur le coût et le sens de l'arc le plus proche qui a été déterminé. Enfin, le plus court chemin est réalisé à partir des deux nouveaux nœuds. La requête du plus court chemin n'est pas commencée à partir des points extérieurs au graphe car on ne possède pas de l'information suffisante. En effet, pour commencer le plus court chemin au point extérieur au graphe, il faut connaitre le coût nécessaire pour se déplacer hors du réseau routier, qui est en fonction de l'occupation du sol.

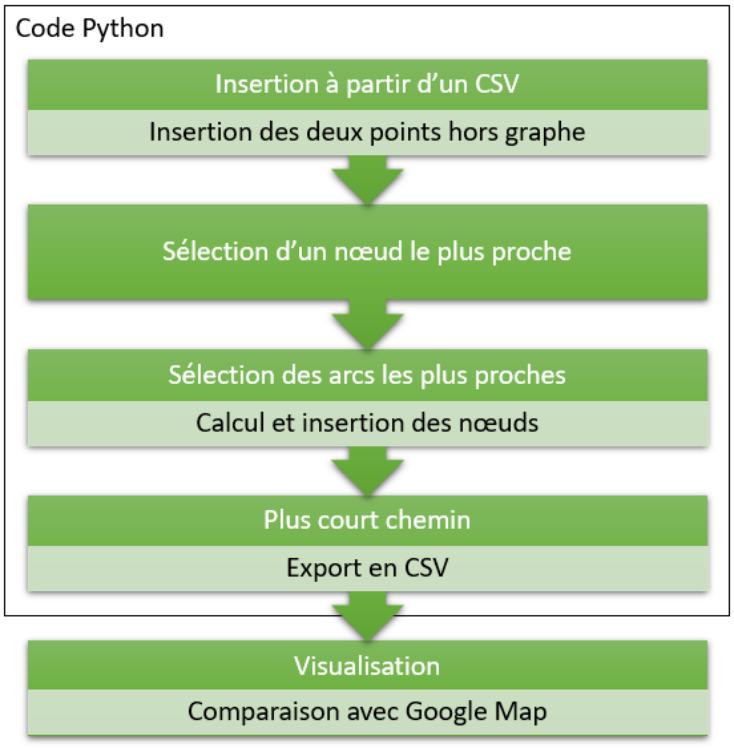


Figure 31 : Résumé des opérations dans le code Python

La réalisation des codes Python est l'étape la plus importante pour la comparaison de l'ensemble des opérations. En effet, il faut que les différentes étapes et requêtes réalisées soient presque identiques. Une problématique majeure pour atteindre cet objectif est l'extraction des données. En effet, Neo4j et PostGIS possèdent des packages de mise en relation avec Python différents et les résultats sont obtenus de façon différente. L'extraction de Neo4j est plus compliquée que dans PostGis car les

tableaux exportés ne sont pas toujours utilisables dans des matrices, ce qui engendre une étape supplémentaire pour extraire les données de manière adéquate.

5.3.3. Résultats obtenus

La première comparaison analyse la sélection du nœud le plus proche et de l'arc le plus proche du point hors graphe. La comparaison ne peut être faite qu'entre Neo4j Spatial et PostGIS car ils utilisent tous les deux une méthode de sélection spatiale d'un nœud. On observe un temps de requête inférieur pour Neo4j Spatial pour les deux types de sélections réalisées (Tableau 5).

	PostGIS (ms)	Neo4j S (ms)
Nœud le plus proche	2322,96	239,08
Arc le plus proche	3484,64	240,24

Tableau 5 : Résultat de temps nécessaire pour la sélection d'éléments les plus proches d'un point extérieur au graphe

On observe que dans les deux cas de sélection d'éléments les plus proches, Neo4j avec l'extension spatiale est plus rapide. La différence est d'environ 10 fois plus rapide pour la sélection du nœud le plus proche et plus de 14 fois pour la sélection de l'arc.

La différence de performance entre les requêtes des deux bases de données peut s'expliquer par plusieurs facteurs. Tout d'abord, le fonctionnement de la base de données graphe permet une lecture plus efficace des données que dans un SGBD-R. La différence ne vient cependant pas du parcours des bases de données puisqu'il se fait de la même façon grâce aux arbres d'indexation spatiale.

La variation est d'autant plus grande pour la sélection d'arcs dans la base de données PostGIS. Ceci peut s'expliquer par une plus grande quantité d'attributs dans la table des arcs de la base de données relationnelle. Une autre cause est que celle-ci nécessite de réaliser une jointure entre la table de nœuds et la table des arcs, avant de pouvoir réaliser la sélection de l'arc. Cette étape n'est évidemment pas nécessaire dans Neo4j grâce à sa structure en graphe et non en table.

On peut aussi constater que les deux temps de sélection sont très similaires dans Neo4j. Ceci peut s'expliquer par la méthode de sélection des éléments. Les arcs dans Neo4j sont considérés uniquement par leurs extrémités. La sélection d'un nœud ou d'un arc ne fait pas de différence pour la base de données.

La seconde comparaison concerne le temps nécessaire à l'exécution de la requête du plus court chemin sur les trois bases de données. L'analyse est faite sur différents trajets dont le nombre de nœuds varie de 250 à 650 nœuds (Tableau 6). Les résultats montrent que la base de données Neo4j Spatial est plus lente pour réaliser la requête de plus court chemin, quel que soit le nombre de nœuds traversé. On observe également que Neo4j sans extension spatiale et PostGIS ont un résultat constant. Au contraire, le temps d'exécution augmente plus ou moins linéairement par rapport au nombre de nœuds traversés pour Neo4j Spatial (Figure 32).

Nombre de nœuds traversés	PostGIS (ms)	Neo4j NS (ms)	Neo4j S (ms)
262	2676,4	1806	3846
401	2713,2	1000,4	5076
500	2674	1298,8	6525
619	2686,8	1696	8388
647	2897	1684,8	8059
Moyenne	2729,48	1497,2	6378,8

Tableau 6 : Résultat du temps nécessaire pour les plus courts chemins entre deux nœuds insérés

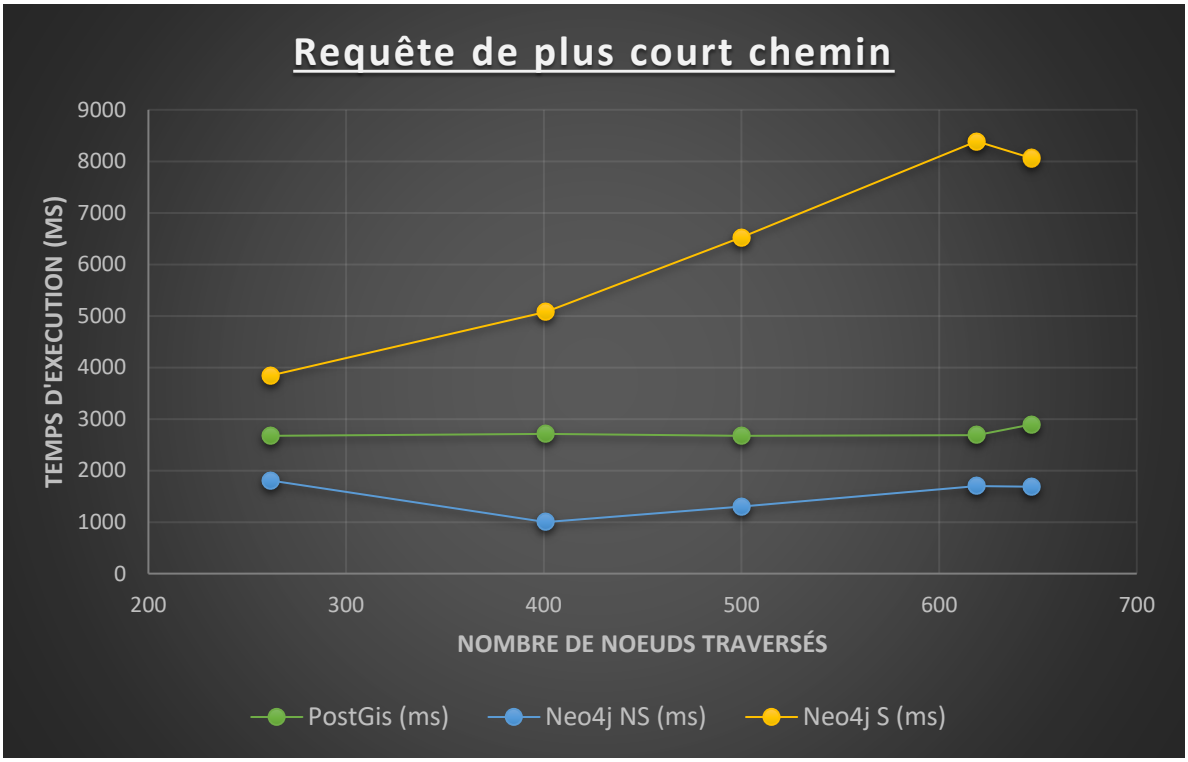


Figure 32 : Graphe représentant le temps nécessaire suivant le nombre de nœuds du parcours

Les résultats obtenus (Tableau 6) par la base de données Neo4j sans indexation spatiale et PostGIS sont cohérents avec ce qui est décrit dans la littérature. PostGIS est plus lent car il utilise un index spatial alors que Neo4j exploite la ressemblance de la base de données inhérente au réseau routier ainsi que le parcours transversal du graphe. PostGIS est cependant nettement plus rapide que la base de données Neo4j avec l'extension spatiale. Les deux bases de données utilisent une indexation spatiale. Cependant, elle engendre sur la base de données graphes des pertes de performance d'autant plus grandes quand le nombre de nœuds augmente.

Il peut y avoir plusieurs raisons qui expliquent les pertes de performance de la base de données Neo4j malgré l'utilisation de l'extension spatiale illustrée avec la Figure 32. Une première cause serait dû à un surplus d'attributs des nœuds dans la base de données Neo4j Spatial. Il y a une importante

redondance d'informations ainsi que des attributs n'ayant pas d'utilité pour le parcours du graphe dans la base de données.

La deuxième cause, et la principale, de cette perte de performance lors du plus court chemin provient de la méthode de lecture des données. Le plus court chemin est une opération profitant de l'efficacité de Neo4j pour le parcours transversal du graphe. Cela signifie que l'analyse des données est faite en minimisant la lecture de données non nécessaires. C'est la raison pour laquelle le parcours du graphe est plus performant pour la base de données sans l'extension spatiale car le graphe a une structure similaire à un arbre d'indexation, ce qui permet d'obtenir le résultat du plus court chemin rapidement. Par contre, l'application de l'index spatial a un effet négatif sur le parcours transversal du graphe. Dans le cas de Neo4j Spatial, l'application d'une indexation spatiale sur le graphe ajoute des contraintes sur le parcours du graphe en classant les entités dans des nœuds suivant leur répartition spatiale. Il faut faire la différence entre les nœuds de l'arbre d'indexation, comme ce cas-ci, et les nœuds de la base de données graphe. L'indexation spatiale engendre donc des surplus d'analyse puisque la base de données cherche d'abord les nœuds proches spatialement puis seulement les arcs ayant un coût plus faible. Il aurait fallu supprimer la contrainte d'indexation pour conserver un temps optimal de parcours transversal.

Une dernière possibilité provient de la façon dont les arcs sont stockés sur Neo4j Spatial. On observe lors de la sélection des éléments des temps presque identiques pour la sélection des arcs et des nœuds. On suppose que ceci est dû au fait que les arcs ne sont pas directement sélectionnés mais qu'on sélectionne uniquement les nœuds de départ et d'arrivé les plus proches. Ces deux constatations laissent penser que lorsque la base de données graphe Neo4j Spatial doit sélectionner des arcs, elle a besoin de mettre en relation les nœuds stockés sur la couche spatiale avec les arcs. Cette étape supplémentaire empêche de réaliser un parcours transversal efficace du graphe.

Pour éviter cette problématique, il est aussi possible d'utiliser d'autres systèmes d'indexation, plus appropriés pour le parcours d'un graphe. En effet, des index spécifiques sont mis en place dans la littérature. Un exemple d'index récemment analysé est le Gtree. Il permet de réaliser efficacement les requêtes spatiales sur des graphes, sans avoir les pertes de performance constatées dans ce mémoire. Le fait de ne pas utiliser d'indexation lors du parcours du graphe est aussi une option puisque la création d'un arbre d'indexation sur un graphe est une étape inutile pour la requête du plus court chemin.

Enfin, le dernier critère comparé est le temps total nécessaire pour la réalisation de l'ensemble des opérations. Pour cela, chaque code doit suivre les mêmes étapes afin que le temps soit comparable. Les étapes du code sont :

- Connexion à la base de données.
- Insertion des deux points hors graphe.
- Sélection du nœud le plus proche d'un point hors graphe.
- Sélection des arcs les plus proches des points hors graphe.
- Détermination du point de l'arc le plus proche du point hors graphe.
- Intégration de nouveaux nœuds dans le graphe, aux coordonnées calculées.
- Création des arcs avec un coût basé sur la longueur des arcs et sur les attributs de coût de l'arc le plus proche.
- Réalisation du plus court chemin entre les deux nœuds intégrés précédemment.

- Export des coordonnées des nœuds du trajet du plus court chemin.

Nombre de nœuds traversés	PostGIS (ms)	Neo4j NS (ms)	Neo4j S (ms)
262	11846,2	14787,6	8914
401	12027,6	14063,4	10025
500	11775,8	14745,4	11637
619	11860,4	14632,8	13033
647	13264,2	14615,4	13477
Moyenne	12154,84	14568,92	11417,2

Tableau 7 : Résultat des temps totaux des codes

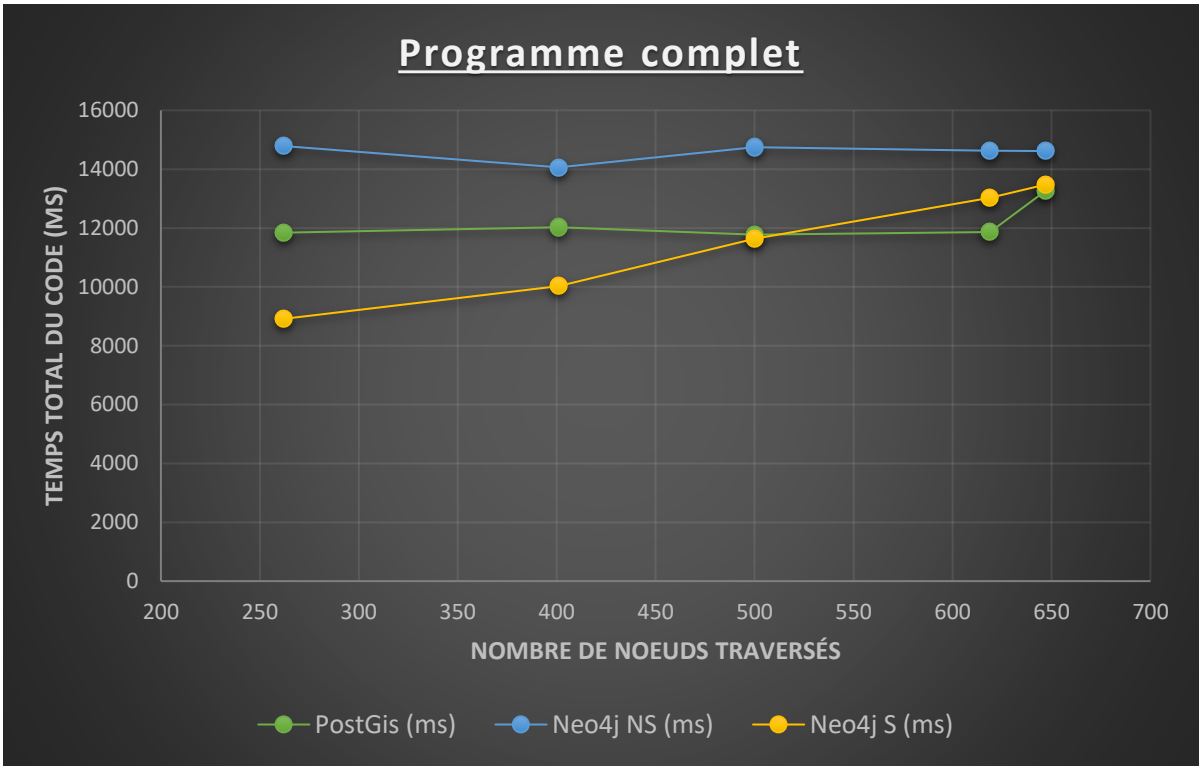


Figure 33 : Graphe comparant les résultats du temps total pour la réalisation des opérations en fonction du nombre de nœuds du plus court chemin

Les résultats montrent que Neo4j avec extension spatiale est globalement plus rapide que PostGIS et que Neo4j sans indexation spatiale. Cependant, il est soumis à une forte croissance qui est corrélée avec le nombre de nœuds traversés lors du plus court chemin. En effet, Neo4j Spatial est plus rapide que PostGIS lorsque le nombre de nœuds traversés est inférieur à 500 nœuds (Figure 33), puis le temps devient plus grand et dépasse le temps nécessaire à PostGIS pour réaliser les opérations. Cette variation est uniquement due aux pertes de performances lors de la réalisation du plus court chemin sur la base de données Neo4j Spatial constatées dans le Tableau 6.

Le trajet résultant du plus court chemin sur PostGIS peut aussi être analysé afin de vérifier qu'il n'y ait pas d'aberrance. Pour cela, on utilise la connexion existante entre PostGIS et Qgis, ce qui permet d'afficher directement le réseau de la base de données ainsi que le résultat de la requête du plus court chemin (Figure 33). Les trajets des deux bases de données donnent un résultat similaire.

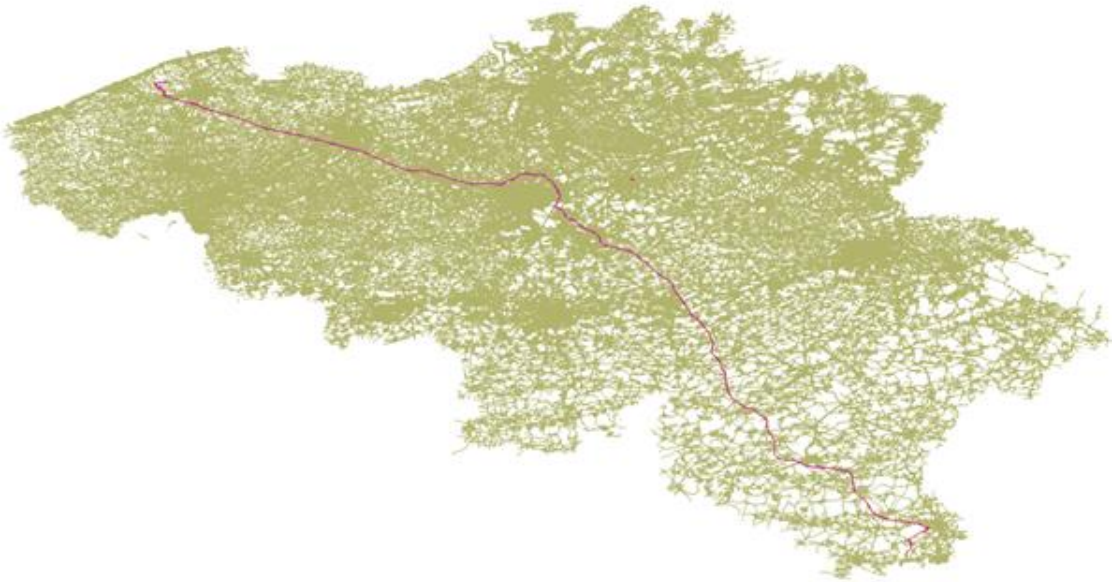


Figure 34 : Trajet résultant du plus court chemin entre deux nouveaux nœuds insérés dans le graphe (Annexe 1)

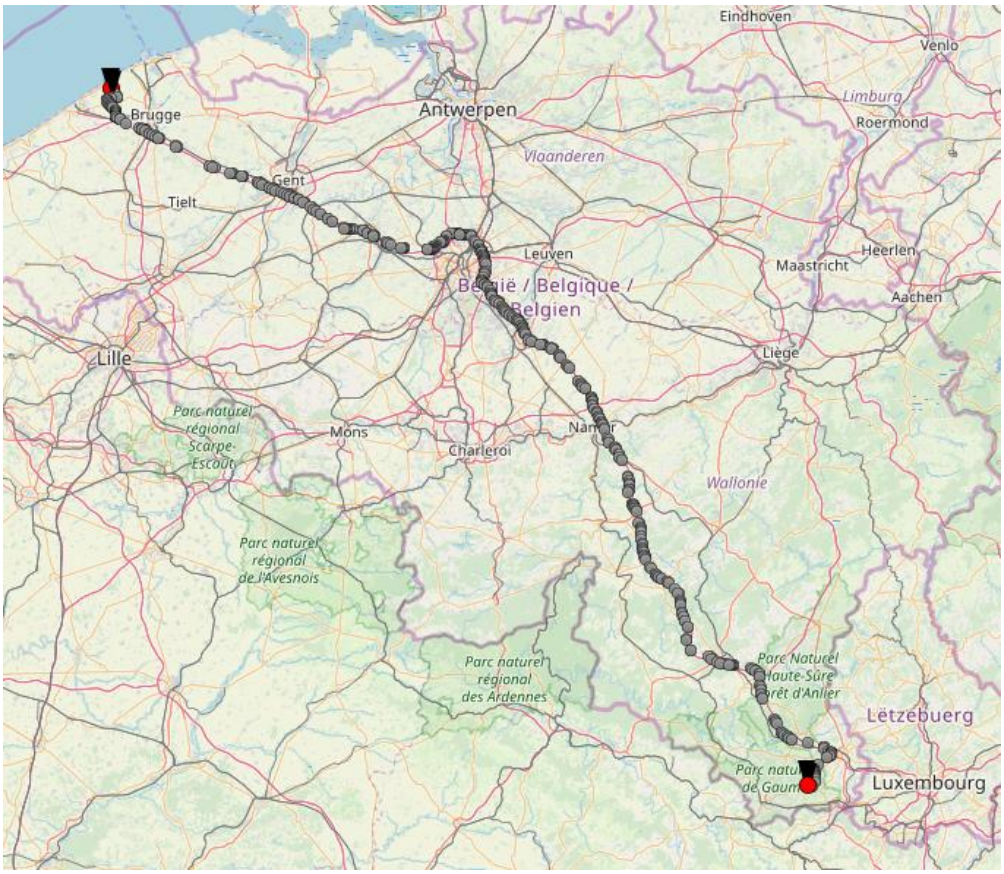


Figure 35 : Suite de nœuds résultant du plus court chemin entre deux nouveaux nœuds insérés dans le graphe (Annexe 1)

5.3.4. Conclusion de la comparaison

Les résultats sont que Neo4j sans le plug-in spatial est plus rapide que les deux autres bases de données pour faire un plus court chemin. Par contre, lorsqu'on configure une base de données sur Neo4j avec l'extension spatiale, le plus court chemin est ralenti proportionnellement au nombre de nœuds parcourus mais la sélection d'un nœud ou d'un arc le plus proche devient nettement plus rapide. Par rapport au temps nécessaire pour réaliser la totalité du code, l'extension spatiale rend donc la base de données Neo4j plus rapide pour des trajets ayant un plus petit nombre de nœuds parcourus mais plus lent dans le cas où le nombre de nœuds devient trop important. Ceci s'explique par les importantes pertes de performance sur Neo4j Spatial pour la réalisation du plus court chemin. Pour les autres étapes, l'extension spatiale de Neo4j donne accès à des requêtes et à l'indexation spatiale qui permettent de gagner du temps sur la globalité du code.

Dans le cas de PostGIS, celui-ci possède un avantage sur la base de données Neo4j sans l'extension spatiale pour réaliser la sélection des éléments proches d'un point extérieur au graphe (Tableau 1 et Tableau 5). Il possède aussi des outils qui lui permettent d'être plus performant sur l'ensemble des opérations. Ces résultats sont en accord avec les conclusions provenant de la littérature.

Enfin, Neo4j sans l'extension spatiale n'est performant que pour faire le plus court chemin car il s'agit d'un simple parcours de graphe sans besoin de gestion de données géospatiales.

Grâce aux résultats de la sélection des arcs les plus proches des points extérieurs au graphe, on peut tirer comme conclusion que l'indexation spatiale et la prise en compte de la géométrie fonctionnent.

En se basant sur la comparaison de la sélection d'un nœud entre les deux bases de données Neo4j, on peut aussi dire que l'extension spatiale donne accès à des requêtes plus efficaces pour réaliser des opérations spatiales. Cependant, l'indexation spatiale a aussi des désavantages puisqu'elle contraint la lecture des données en suivant l'arbre du Rtree, ce qui retire l'avantage initial de parcours transversal des bases de données graphes.

6.Conclusion

6.1. Rappel de la question de recherche et de l'hypothèse

Le but de ce mémoire est de mettre en avant la meilleure solution pour faire des requêtes spatiales sur un réseau routier. Après une analyse de la littérature, la solution qui a été dégagée est d'utiliser une base de données NoSQL orientée graphe. En effet, ce type de base de données possède une structure inhérente au réseau routier qui permettrait de réduire le temps d'exécution.

Dans les différentes bases de données utilisées, pour la réalisation de requêtes spatiales, PostGIS se révèle être une base de données relationnelle fréquente. De la même façon, Neo4j est la base de données graphe la plus performante.

Le choix de la requête spatiale se base sur la littérature. Il est avéré que la requête la plus fréquente est celle du plus court chemin. Plusieurs articles ont déjà établi que les performances de cette requête sur Neo4j sont meilleures que sur PostGIS. Ce mémoire inclut une requête supplémentaire qui est la sélection du plus proche voisin pour laquelle Neo4j ne présente pas d'avantage. Des articles suggèrent qu'une modification de la configuration spatiale de Neo4j permettrait d'améliorer la rapidité d'exécution.

Ces différents constats ont amené à émettre l'hypothèse suivante : **La référenciation spatiale des points rend la vitesse d'exécution d'une requête du plus court chemin entre deux points extérieurs au graphe inférieure, à celle de PostGIS, dans un grand réseau routier sur Neo4j.**

6.2. Résultats de l'analyse

Trois bases de données sont comparées dans le cadre de cette recherche : Neo4j, Neo4j avec l'extension spatiale et PostGIS. Les trois bases de données contiennent le même réseau puisqu'il est préalablement configuré dans PostGIS puis importé dans Neo4j. Neo4j Spatial et PostGIS ont des configurations similaires avec la position relative des éléments ainsi qu'une indexation spatiale de type Rtree ou Rtree+. Les opérations sont toutes réalisées dans un code Python et sont presque identiques dans les trois cas.

6.2.1. Résultats de la sélection d'un nœud le plus proche voisin d'un point extérieur au graphe

La requête de sélection du plus proche voisin est analysée à deux reprises. D'une part, pour comparer les deux bases de données Neo4j (Tableau 1) et, d'autre part, pour comparer les deux sélections spatiales des bases de données Neo4j et PostGIS (Tableau 5).

Dans le premier cas, on constate que l'indexation spatiale ne permet pas d'accélérer la requête calculant la distance euclidienne pour chaque nœud. L'utilisation de l'extension spatiale engendre

même un temps supplémentaire dû à un surplus d'attributs des nœuds. Cependant, lorsqu'on compare les deux méthodes de sélection du plus proche voisin, on observe un net avantage du temps d'exécution en faveur de la sélection du nœud le plus proche spatialement. Le temps de réalisation de la requête est environ 10 fois plus petit. Ces résultats mettent en avant l'avantage d'avoir la position relative des points et que les données soient soumises à une indexation spatiale.

Le second cas compare les deux bases de données spatiales, Neo4j et PostGIS. On observe que Neo4j Spatial prend beaucoup moins de temps pour exécuter la requête de sélection du nœud le plus proche. Le temps pour la sélection d'un nœud est plus de dix fois inférieur au temps nécessaire sur PostGIS. La variation des performances ne peut pas provenir de la différence d'indexation spatiale car, dans ce mémoire, les arbres d'indexation sont les mêmes avec le Rtree et le Rtree+. La différence doit donc venir de la structure même du modèle qui permet une lecture plus efficace des données dans la base de données graphe.

6.2.2. Résultats de la sélection de l'arc le plus proche d'un point extérieur au graphe

Dans le cas de la sélection de l'arc le plus proche, les deux bases de données avec extensions spatiales sont comparées. Les temps obtenus sont environ 14 fois plus longs (Tableau 5) sur PostGIS que sur Neo4j Spatial. Cela peut se justifier de la même manière que pour la sélection du nœud le plus proche, par l'efficacité pour la lecture des données. Cependant, la différence est encore plus grande, ce qui peut être dû à la jointure nécessaire sur la base de données relationnelle entre la table des nœuds et celle des arcs. Au contraire, Neo4j interprète directement l'arc comme un nœud de départ et d'arrivée.

Il est possible de mettre en avant une caractéristique importante de Neo4j Spatial en observant les résultats de la sélection du nœud et de l'arc le plus proches du point. En effet, le temps nécessaire pour les deux requêtes est presque identique. L'explication est que les arcs sont interprétés par leurs extrémités et que la requête porte donc uniquement sur les nœuds proches. Cela rend l'opération de sélection de l'arc particulièrement efficace dans Neo4j Spatial.

6.2.3. Résultats du plus court chemin

La requête du plus court chemin a déjà été testée dans la littérature, mais jamais avec l'extension spatiale de Neo4j. Il est donc intéressant de voir si celle-ci améliore les performances du plus court chemin. Les plus courts chemins sont basés sur le même algorithme : celui de DIJKSTRA. L'analyse se fait sur plusieurs itérations et sur des trajets ayant un nombre de nœuds variable.

Le résultat est que le plus court chemin sur la base de données Neo4j, sans l'extension spatiale, est plus rapide que sur les deux autres bases de données (PostGis et Neo4j Spatial). L'avantage de Neo4j sans l'extension spatiale, par rapport à PostGIS, confirme la conclusion tirée de la littérature sur laquelle se base l'hypothèse de ce mémoire.

Pour le plus court chemin sur Neo4j Spatial, le résultat n'est pas concluant. En effet, celui-ci subit des ralentissements d'autant plus importants que le nombre de nœuds traversés augmente. L'utilisation de l'extension spatiale présente plusieurs aspects qui peuvent engendrer ces pertes de performances. Le premier aspect est que l'insertion sur la couche spatiale engendre un surplus d'attributs pour les

nœuds. Le deuxième aspect est dû à l'application de l'indexation spatiale sur le graphe. La base de données a une structure naturelle sous forme d'arbre qui lui permet d'être performante, sans extension spatiale, pour le parcours du graphe. Cependant, l'indexation spatiale ajoute une contrainte faisant perdre cet avantage en forçant le parcours de la base de données en suivant l'arbre Rtree. Un troisième et dernier aspect est que les arcs ne sont pas intégrés à la couche spatiale, cela engendre une étape supplémentaire durant la requête. C'est-à-dire que le plus court chemin est appliqué sur les deux nœuds présents sur la couche spatiale, qui définisse un arc, puis mis en relation avec l'arc correspondant. Ces différents éléments empêchent la réalisation d'un parcours transversal efficace sur Neo4j Spatial.

Il y a diverses possibilités pour empêcher cette perte de performances, on peut soit ne pas utiliser d'indexation spatiale lors de la réalisation du plus court chemin, soit utiliser un index approprié pour le parcours du graphe. Par exemple, l'index spatial Gtree, proposé dans la littérature a été développé spécialement pour réaliser les requêtes spatiales sur des graphes. Il n'a pas été utilisé, dans le cadre de ce mémoire, car il n'est pas encore disponible sur Neo4j. Il n'y a, actuellement, pas non plus de solution disponible pour intégrer les arcs à la couche spatiale, ce qui risque de ralentir la requête malgré une indexation appropriée.

La comparaison visuelle des trajets obtenus avec le planificateur de trajet de Google Maps permet de vérifier le bon fonctionnement de l'algorithme. Le résultat obtenu bien que légèrement différent n'est pas aberrant. Il y a plusieurs éléments qui peuvent expliquer les variations des trajets : des algorithmes de planification du plus court chemin différents, une variation dans le réseau des bases de données ou encore des modifications dues à des informations supplémentaires fournies par Google Maps.

6.2.4. Résultats de la comparaison des programmes

La dernière comparaison analyse le temps total pour réaliser l'ensemble des opérations.

Neo4j sans l'extension spatiale prend plus de temps pour l'exécution de la totalité du code. Cette comparaison est, néanmoins, peu pertinente puisque les opérations ne sont pas les mêmes que sur les deux autres bases de données. En effet, comme il l'a été mis en avant par la littérature, des opérations supplémentaires doivent être réalisées sur Neo4j sans l'extension spatiale.

La configuration avec l'extension spatiale de Neo4j permet de réaliser directement les requêtes spatiales et ainsi d'accélérer l'ensemble du programme. On obtient dans un premier temps, un temps d'exécution globale inférieur avec Neo4j Spatial par rapport aux deux autres bases de données (Tableau 7). Le temps nécessaire devient supérieur à celui nécessaire sur PostGIS lorsque le nombre de nœuds traversés dépasse environ 500 nœuds. La variation du temps pour réaliser la totalité des opérations de Neo4j Spatial est uniquement due au surplus de temps nécessaire au plus court chemin.

6.3. Conclusion sur la validation de l'hypothèse

La validation de l'hypothèse doit être nuancée. On peut affirmer que le plus court chemin entre deux points extérieurs au graphe sur la base de données Neo4j Spatial est plus rapide que sur PostGIS mais pour un nombre de nœuds traversés inférieur à 500 nœuds.

Cependant, lorsque le nombre de nœuds traversés devient supérieur à 500, l'hypothèse n'est que partiellement validée.

Dans ce cas, les différentes requêtes ont des résultats variables. La sélection spatiale d'un nœud ou d'un arc est plus rapide sur la base de données graphe Neo4j Spatial, mais l'indexation spatiale engendre des ralentissements importants pour l'exécution du plus court chemin.

6.4. Perspective

La conclusion étant que la configuration spatiale testée permet d'être plus rapide. Les prochaines études peuvent chercher, soit à améliorer la configuration testée pour diminuer l'effet négatif de l'indexation sur le parcours transversal, soit continuer l'analyse de cette configuration en l'appliquant à d'autres requêtes spatiales.

Pour modifier la configuration, il peut être intéressant de réaliser son propre système d'indexation spatiale sur Neo4j de façon à ce qu'il soit plus approprié pour un graphe, en se basant sur le Gtree, par exemple. Une autre possibilité est de modifier la configuration de la base de données afin de conserver certains attributs n'étant pas soumis à l'index spatial. Ces paramètres doivent permettre de réaliser un plus court chemin qui bénéficierait du parcours transversal du graphe.

Dans le cas où la configuration ne peut pas être améliorée, on peut analyser d'autres requêtes spatiales. Des tests peuvent être aussi réalisés avec d'autres algorithmes pour réaliser le plus court chemin, comme l'heuristique A*. Il est aussi possible d'étendre l'analyse à d'autres types de requêtes spatiales pour confirmer les résultats obtenus dans ce mémoire, par exemple, la sélection d'éléments dans une certaine zone.

Il est intéressant de suivre l'évolution des capacités de spatialisation de bases de données graphe tel que Neo4j. Elles possèdent un potentiel important de gestion des données géographiques comme un réseau routier.

7.Références

AHUJA, R.K., MEHLHORN, K., ORLIN, J. & TARJAN, R.E. «Faster algorithms for the shortest path problem.» *Journal fo the ACM*, 1990: 2013-223.

AMIRIAN, P., BASIRI, A. & WINSTANLEY, A. «Evaluation of data management systems for geospatial big data.» *International Conference on Computational Science and Its Application*, 2014: 678-690.

AMIRIAN, P., BASIRI, A., GALES, G., WINSTANLEY, A. & MCDONALD, J. «The next generation of navigationnal services using OpenSreetmap data: The integration of augmented reality and graph databases.» Dans *OpenStreetMap in GIScience*, 211-228. 2015.

ANGLES, R. & GUTIERREZ, C. «Survey of graph database models.» Dans *ACM Computing Surveys*, 1. 2008.

ANGLES, R. «A Comparison of Current Graph Database Models.» *IEE 28th International Conference on Data Engineering Workshops.* 2012.

ANGLES, R., ARENAS, M., BARCELO, P., HOGAN, A., REUTTER, J. & VRGOC, D. «Foundations of modern query languages for graph databases.» *ACM Computing Surveys*, 2017.

ArangoDB. *ArangoDB v3.3.23 Documentation.* s.d.

BAAS, B., Geographical Information Management and Applications (GIMA). «NoSQL Spatial : Neo4j versus PostGIS.» Delft, 2012.

BATRA, S. & TYAGI, C. «Comparative analysis of relational and graph databases.» *International Journal of Soft Computing and Engineering, 2*, 2012: 509-5012.

BATTLE, R. & KOLAS, D. «Geosparql: enabling a geospatial semantic web.» *Semantic Web Journal*, 2011: 355-370.

BEIS, J.S. & LOWE, D.G. «Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces.» 1997.

Big on Data. *Graph databases and RDF: It's a family affair.* 2017. https://www.zdnet.com/article/graph-databases-and-rdf-its-a-family-affair/ (accès le juin 2019).

Boston Geographic Information Systems. *pgRouting: Loading OpenStreetMap with Osm2Po and route querying.* 2019. http://www.bostongis.com/PrinterFriendly.aspx?content_name=pgrouting_osm2po_1.

CALVERT, K.L., DOAR, W.B. & ZEGURA, E.W. «Modeling internet topology.» *IEEE Communications magazine* , 1997: 160-163.

CATTUTO, C., QUAGGIOTTO, M., PANISSON, A. & AVERBUCH, A. «Time-varying social networks in a graph database: a Neo4j use case.» *First international workshop on graph data management experiences and systems.* 2013. 11.

CIGLAN, M., AVERBUCH, A. & HLUCHY, L. «Benchmarking Traversal Operations over Graph Databases.» *International Conference on Data Engineering Workshops 28th.* 2012. 186 - 189.

DALTIO, J. & MEDEIROS, C.B. *HydroGraph : Exploring Geographic Data.* Vol. 68, chez *Revista Brasileira de Cartografia*. 2016.

DB-ENGINES. *DB-Engines Ranking.* 2019. https://db-engines.com/en/ranking.

DE SOUZA, B., CLAUDIO, C., DANIEL, F. & MAXWELL, G. «NoSQL geographic databases : on overview.» Dans *Geographic information systems trends and technologies*, 73-103. CRC Press, 2014.

Digora. *Qu'est-ce qu'une base Nosql ? Les cas DataStax et MongoDB.* s.d. https://www.digora.com/fr/blog/definition-base-nosql-datastax-mongodb (accès le Juillet 9, 2019).

DOMINGUEZ-SAL, D., URBON-BAYES, P., GIMENEZ-VANO, A., GOMEZ-VILLAMOR, S., MARTINEZ-BAZAN, N. & LARRIBA-PEY, J.L. «Survey of graph database performance on the hpc scalable graph analysis benchmark.» *International Conference on Web-Age Information Management.* Heidelberg, Berlin, 2010. 37-48.

FU, l., SUN, D. & RILETT, L.R. «Heuristic shortest path algorithms for transportation applications: stat of the art.» *Computers & Operations Research* , 2006: 3324-3343.

GARDARIN, G. *Bases de Données objet & Relationnel.* Paris: Eyriolles, 1999.

GHRAB, A., ROMERO, O., JOUILI, S., & SKHIRI, S. «Graph BI & Analytics: Current Stat and Future Challenges.» *International Conference on Big Data Analytics and Knowledge Discovery.* 2018. 3-18.

GUBICHEV, A., BEDATHUR, S., SEUFERT, S. & WEIKUM, G. «Fast and accurate estimation of shortest paths in large graphs.» *Proceedings of 19th ACM international conference on Information and knnwledge management.* 2010. 499-508.

GUTTMAN, A. «R-trees : a dynamic index structure for spatial researching.» *ACM Sigmod Record*, 1984: 47-57.

HAN, J., HAIHONG, E., LE, G. & DU, J. «Survey on NoSQL database.» *6th international conference on pervasive computing and applications*, 2011: 363-366.

HOLEMANS, A. «Implémentation d'une base de données géospatiale Nosql.» *https://matheo.uliege.be/handle/2268.2/3136.* Université de Liege. 2017.

HOLEMANS, A., KASPRZK, J.P., & DONNAY, J.P. «Coupling an Unstructured NoSQL Database with a Geographic Information System.» *GEOProcesssing Information 2018. The Tenth International Conference on Advanced Geographic Information*, 2018: 23-28.

JAROSLAV, P. «Graph Databases: Their Power and Limitations.» *IFIP International Conference on Computer Information Systems and Industrial Management.* 2015. 58-69.

JEMINE, J. *Conception et implémentation d'un SIG NoSQL.* 2017.

JOSSE, G., SCHMID, K.A., ZUFLE, A., SKOUMAS, G., SCHUBERT, M., RENZ, M. PFOSER, D. & NASCIMENTO, M.A. «Knowledge extraction from crowdsourced data for the enrichment of road networks.» *Geoinformatica, 21*, 2017: 763-795.

LAKSHMAN, A. & PRASHANT, M. «Cassandra: a decentralized structured storage system.» *ACM SIGOPS Operating Systems Review*, 2010: 35-40.

LUDWIG, I., VOSS, A. & KRAUSE-TRAUDES, M. «A Comparison of the Street Network of Navteq and OSM in Germany.» Dans *Advancing geoinformation science for a changing world*, 65-84. 2011.

MARTINS, B., SILVA, M.J. & ANDRADE, L. «Indexing and ranking in Geo IR systems.» *Proceedings of the 2005 workshop on Geographic information retrieval.* 2005. 31-34.

MEDEIROS, C.B. & PIRES, F. «Databases for GIS.» *ACM Sigmod Record*, 1994: 107-115.

MILER, M., DAMIR, M. & DRAZEN, O. «The shortest path algorithm performance comparison in graph and relational database on a transportation network.» *PROMET-Traffic Transport*, 2014: 75-82.

MOELLER, C. *OSM2PO.* s.d. http://osm2po.de/.

NAYAK, A., PORIYA, A., POOJARY, D. «Type of Nosql databases and it's comparison with relational databases.» *International Journal of Applied Information Systems*, 2013: 16-19.

Neo4j. *APOC User Guide 3.5.* 2019. https://neo4j-contrib.github.io/neo4j-apoc-procedures/#overview.

Neo4j contrib. *Neo4j Spatial v0.24-neo4j-3.1.4.* 2017. https://neo4j-contrib.github.io/spatial/0.24-neo4j-3.1/index.html.

Neo4j. *The Neo4j graph algorithms user guide V3.5.* 2019. https://neo4j.com/docs/graph-algorithms/current/.

NIGET, S. *The Py2neo v4 Handbook.* 2011-2019. https://py2neo.org/v4/.

OrientDB. *OrientDB Manual - version 2.2.x.* s.d. http://www.orientdb.com/docs/last/index.html.

OUSSOUS, A., BENJELLOUN, F.Z., LAHCEN, A.A. & BELFKIH, S. «Comparison and classification of nosql databases for big data.» Dans *Proceedings of International Conference on Big Data,Cloud and Applications*. 2015.

RAMM, F. «OpenStreetMap Data in Layered GIS Format.» 2015.

RITSEMA VAN ECK, J.R. & DE JONG, T. «Off the Road: From Data points to the Networks in GIS-Based Network Analysis.» 2002.

ROBINSON, I., WEBBER, JIM. & EIFREM, E. *Graph Databases.* Sebastopol: O'Reilly Media, Inc., 2015.

SHARMA, V. & DAVE, M. «Sql and nosql databases.» 2012.

SKUPIN, A. & FABRIKANT, S., I. «Spatialization.» Dans *The Handbook of Geographic Information Science*, 61-80. 2007.

THEOHARIS, Y., CHRISTOPHIDES, V. & KARVOUNARAKIS, G. «Benchmarkinh database representations of RDF/s stores.» *International Semantic Web Conference*, 2005: 685-701.

VICKNAIR, C., MACIAS, M., ZHAO, Z., NAN, X., CHEN, Y. & WILKINS, D. «A Comparison of a graph database and a relationnal database : A data provenance perspective.» *Proceedings of the 48th Annual Southeast Regional Conference.* New York, 2010. 1-42:6.

WAGNER, B. F., HARLEY, V. O., MARISTELA, H. & ALETEIA, A. F. «Geographic data modeling for NoSQL document-oriented databases.» *GEOProcessing 72*, 2015: 63-68.

WOOD, P.T. «Query languages for graph databases.» *ACM Sigmod Record*, 2012: 50-60.

YAQOOB, I., HASHEM, I., GANI, A., MOKHTAR, S., AHMED, E., ANUAR, N.B. & VASILAKOS, A.V. «Big data: From beginning to future.» Dans *International Journal of Information Management*, 1231-1247. 2016.

ZHAN, F.B. & NOON, C.E. «Shortest path algorithms: an evaluation using real road networks.» *Transportation science*, 1998: 65-73.

ZHAO, J.L. & CHENG, H.K. «Graph indexing for spatial data traversal in road map databases.» Dans *Computers & Operations Research*, 223-241. 2001.

ZHONG, R., LI, G., TAN, K.L., ZHOU, L. & GONG, Z. «G-tree: An efficient and scalable index for spatial search on road networks.» *IEEE Transactions on Knowledge and Data Engineering*, 2015: 2175-2189.

Annexe 1 : Résultats des trajets testés

Cas 1

Coordonnées des points extérieurs au graphe :

	Longitude	Latitude
Point 1	5,69538	49,59692
Point 2	3,08441	51,2516

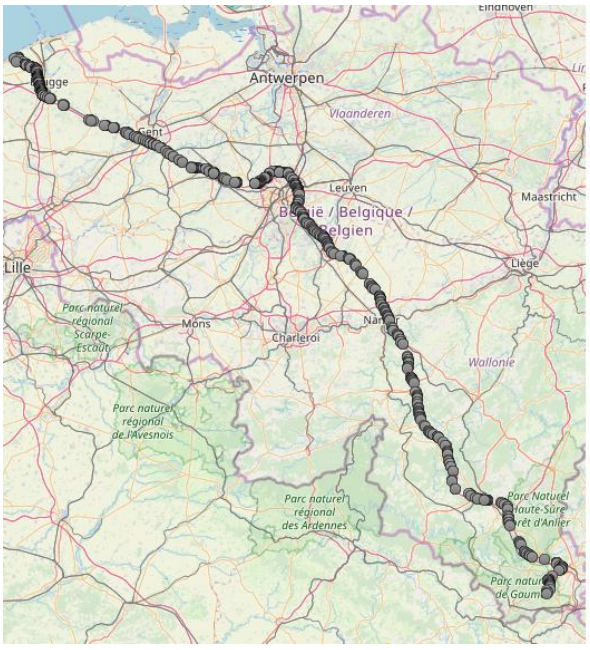
Nœud le plus proche du point 1, hors graphe :

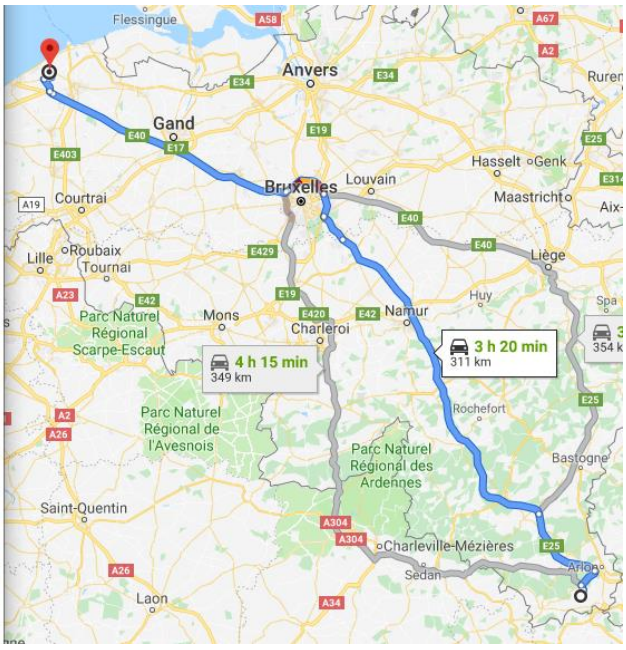
```
node.longitude | node.latitude
---------------|---------------
     5.7010015 |    49.5872749
```

Arc le plus proche des points hors graphe :

	ID_OSM		Coord Départ		Coord Arrivée	
	Nœud Départ	Nœud Arrivée	Longitude	Latitude	Longitude	Latitude
Point 1	463352	489616	5.7028672	49.5884154	5.7010015	49.5872749
Point 2	303226	392940	3.0953979	51.243476	3.0878026	51.252354

Parcours comparés aux parcours Google Maps :





Cas 2

Coordonnées des points extérieurs au graphe :

	Longitude	Latitude
Point 1	4,559136	50,132586
Point 2	5,985007	50,282731

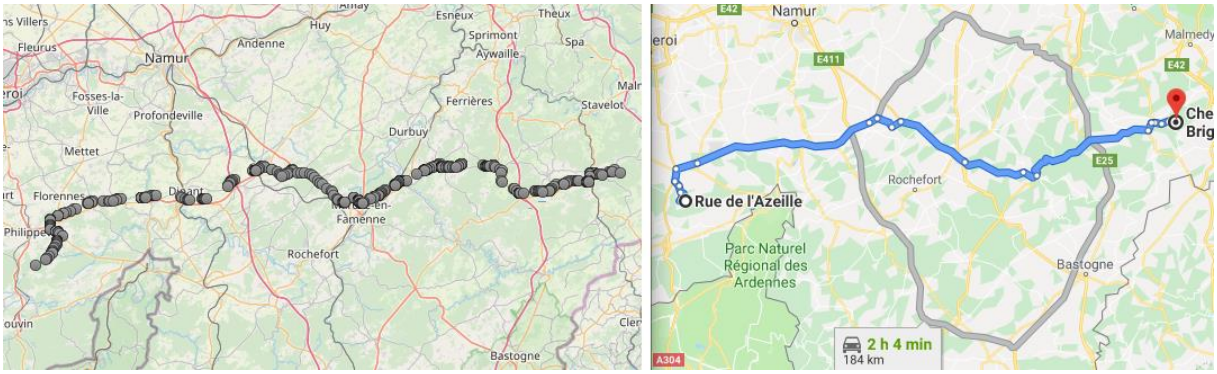
Nœud le plus proche du point 1, hors graphe :

```
node.longitude | node.latitude
---------------|--------------
     4.5606976 |     50.138491
```

Arc le plus proche des points :

	ID_OSM		Coord Départ		Coord Arrivée	
	Nœud Départ	Nœud Arrivée	Longitude	Latitude	Longitude	Latitude
Point 1	413063	413061	4.5547143	50.1368907	4.5606976	50.1384910
Point 2	190228	360253	5.9670677	50.2882347	5.9713767	50.2851894

Parcours comparés aux parcours Google Maps :



Cas 3

Coordonnées des points extérieurs au graphe :

	Longitude	Latitude
Point 1	5,282581	51,191161
Point 2	5,558992	50,227662

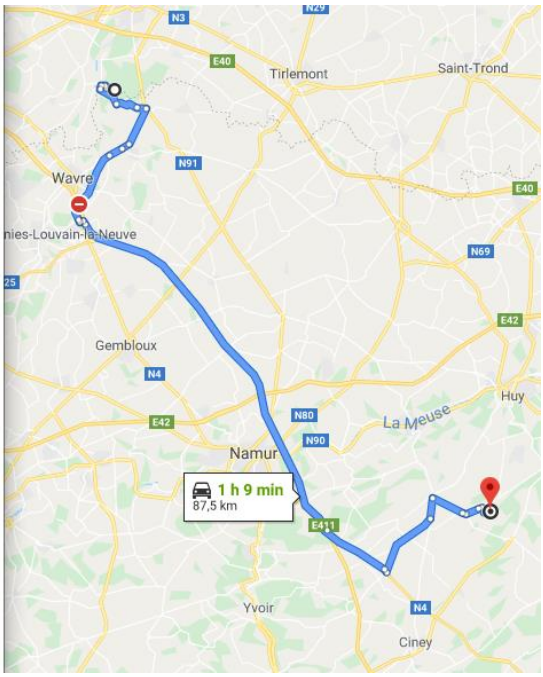
Nœud le plus proche du point 1, hors graphe :

```
node.longitude | node.latitude
---------------|---------------
     5.2774742 |    51.1856916
```

Arc le plus proche des points :

	ID_OSM		Coord Départ		Coord Arrivée	
	Nœud Départ	Nœud Arrivée	Longitude	Latitude	Longitude	Latitude
Point 1	333376	169805	5.2792027	51.184445	5.2789397	51.180162
Point 2	138433	461582	5.5646845	50.2379918	5.5775330	50.2313643

Parcours comparés aux parcours Google Maps :



Cas 4

Coordonnées des points extérieurs au graphe :

	Longitude	Latitude
Point 1	3,599875	51,270393
Point 2	5,354739	49,703617

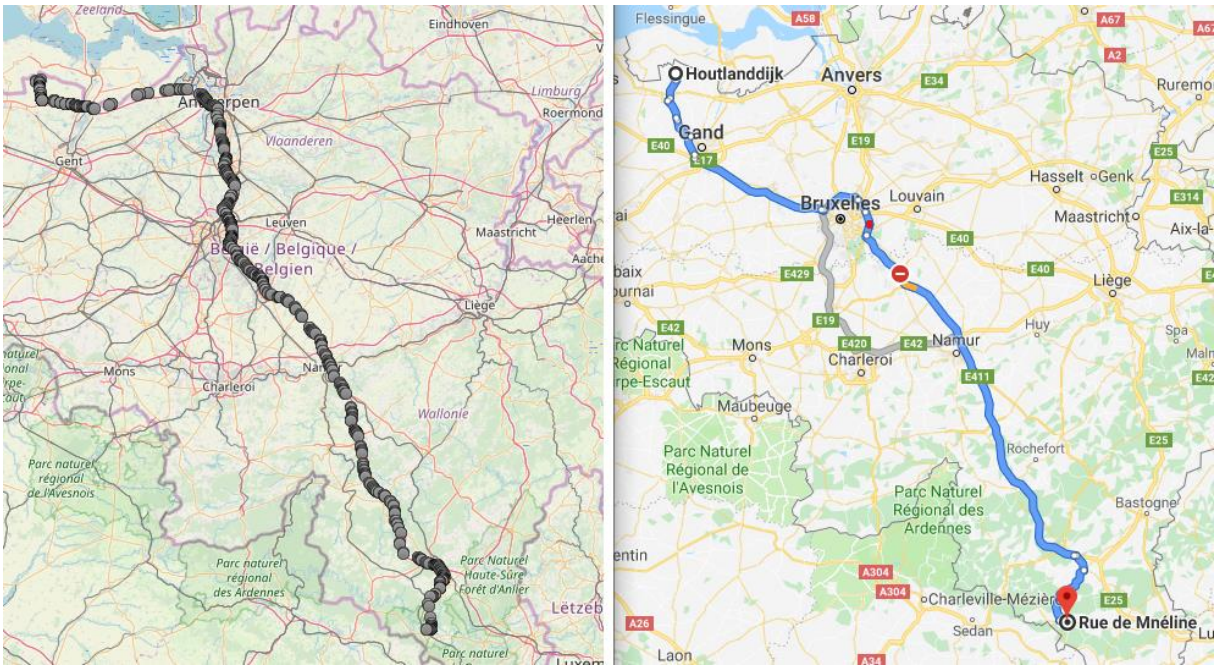
Nœud le plus proche du point 1, hors graphe :

```
 node.longitude | node.latitude
----------------|---------------
      3.5997162 |    51.2630164
```

Arc le plus proche des points :

	ID_OSM		Coord Départ		Coord Arrivée	
	Nœud Départ	Nœud Arrivée	Longitude	Latitude	Longitude	Latitude
Point 1	119456	119457	3.5997162	51.2630164	3.6075242	51.2653967
Point 2	428625	202292	5.3570594	49.6948496	5.3620762	49.6946080

Parcours comparés aux parcours Google Maps :



Cas 5

Coordonnées des points extérieurs au graphe :

	Longitude	Latitude
Point 1	4,672664	50,801012
Point 2	5,207555	50,411976

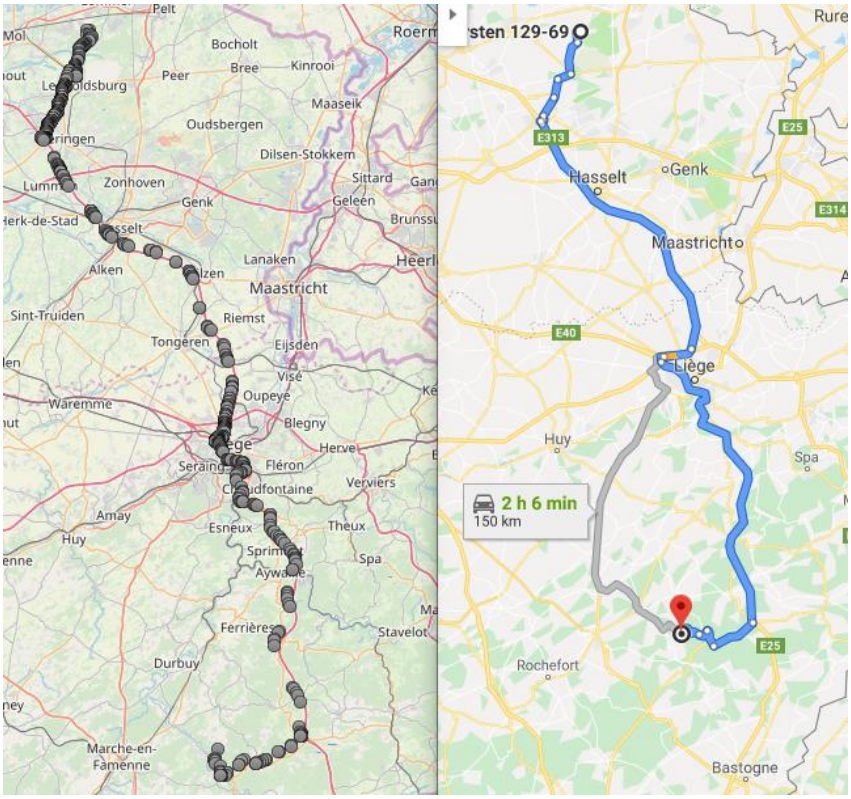
Nœud le plus proche du point 1, hors graphe :

```
node.longitude | node.latitude
---------------|---------------
     4.6663304 |    50.7984656
```

Arc le plus proche des points :

	ID_OSM		Coord Départ		Coord Arrivée	
	Nœud Départ	Nœud Arrivée	Longitude	Latitude	Longitude	Latitude
Point 1	225570	489724	4.665999	50.7985906	4.66556950	50.7991158
Point 2	88341	443333	5.1976614	50.4142692	5.19893680	50.4111978

Parcours comparés aux parcours Google Maps :



Annexe 2 : Code Python des opérations réalisées sur Neo4j avec l'extension spatiale.

```
from py2neo import Graph
from pandas import DataFrame
import numpy as np
import math as math
import csv
import time
import os

#Début du chronomètre pour la mesure du code dans son ensemble
DebutQ1 = time.perf_counter_ns()

#Connexion à la database
NeoGraph = Graph("bolt://localhost:7687", auth=("neo4j", "6720"))

Pts = np.zeros((2,2))
#Insertion d'un point provenant du fichier PtsExt.csv
#Lecture du fichier CSV dans Python
csvfile = "C:/Users/home/.Neo4jDesktop/neo4jDatabases/database-7aa7d067-
3bc4-498e-8491-0a1888107785/installation-3.5.6/import/PtsExt3.csv"
with open(csvfile) as file:
  reader = csv.DictReader(file)
  for row in reader:
    id=int(row['IdPoint'])
    Pts[(id-1),0]=row['longitude']
    Pts[(id-1),1]=row['latitude']
    print(Pts)

#Insertion des noeuds dans la base de données par importation d'un fichier
CSV
insert = NeoGraph.run("LOAD CSV WITH HEADERS FROM 'file:///PtsExt3.csv' AS
row CREATE (n:PtsExt) SET n=row,n.x = toFloat(row.x),n.y = toFloat(row.y)")

#Insertion des données du CSV du premier point extérieur au graphe, dans
une matrice, pour faciliter la manipulation dans Python
long1=float(Pts[0,0])
lat1 =float(Pts[0,1])
Pts1 = np.zeros((1,2))
Pts1[0,0] = float(long1)
Pts1[0,1] = float(lat1)
print("Closest Points")

#Début du chronomètre pour mesurer le temps nécessaire pour la requête de
sélection spatial d'un noeud
AvantClose = time.perf_counter_ns()
#Sélection du noeud le plus proche grâce à une opération spatiale
```

```
ClosestPoint = NeoGraph.run("CALL spatial.closest('geom_noeuds',{
longitude: "+str(long1)+" , latitude:"+str(lat1)+"},0.1) yield node return
node, node.longitude, node.latitude limit 1").to_table()

#Fin du chronomètre, fin du chronomètre et affichage du temps nécessaire
pour le noeud le plus proche
TClose1 = time.perf_counter_ns()
tempsClose1ns = float(TClose1-AvantClose)
tempsClose1ms = float(tempsClose1ns/1000000)
print("Temps d'execution du closest avec la fct en nanoseconde = %d\n"
%tempsClose1ns )
print("Temps d'execution du closest avec la fct en milliseconde = %d\n"
%tempsClose1ms )
print("Le noeud le plus proche est : ")
print(ClosestPoint)

#Début du chronomètre pour mesurer le temps nécessaire pour la requête de
sélection d'un noeud par mesure de la distance
TClose1 = time.perf_counter_ns()
#Sélection d'un noeud le plus proche par calcul de la distance eucledienne
par rapport à tous les noeuds et sélection de la distance minimum
ClosestPointV1 = NeoGraph.run("MATCH (Source:PtsExt), (Target:SpatNoeuds)
WITH SQRT((Source.x - Target.x)^2+(Source.y - Target.y)^2) AS proxi,
Source, Target ORDER BY proxi LIMIT 1 MERGE (Source)-[r:Proximity]-
>(Target) SET r.EucliDist = proxi ")

#Fin du chronomètre, fin du chronomètre et affichage du temps nécessaire
pour le noeud le plus proche
TClose2 = time.perf_counter_ns()
tempsClose2ns = float(TClose2-TClose1)
tempsClose2ms = float(tempsClose2ns/1000000)
print("Temps d'execution du closest par calcul et tri de la distance en
nanoseconde = %d\n" %tempsClose2ns )
print("Temps d'execution du closest par calcul et tri de la distance en
milliseconde = %d\n" %tempsClose2ms )

#Insertion des données du CSV dans une matrice pour le deuxième point
extérieur au graphe
long2=float(Pts[1,0])
lat2 =float(Pts[1,1])
Pts2 = np.zeros((1,2))
Pts2[0,0] = float(long2)
Pts2[0,1] = float(lat2)

#Réalisation et mesure du temps nécessaire pour la sélection des arcs les
plus proches des points extérieurs au graphe
AvantCloseHG = time.perf_counter_ns()
ClosestRel1 = NeoGraph.run("CALL spatial.closest('geom_noeuds',{ longitude:
"+str(long1)+", latitude: "+str(lat1)+"},0.05) yield node MATCH p=(node)-
[r:TEMPS_TRAJ]->(targ) MATCH o=(targ)-[s:REVERSE_TRAJ]->(node) return
node.longitude, node.latitude, targ.longitude, targ.latitude,r.Cout,s.Cout,
node.noeuds_id, targ.noeuds_id limit 2").to_table()
TCloseHG = time.perf_counter_ns()
```

```
ClosestRel2 = NeoGraph.run("CALL spatial.closest('geom_noeuds',{ longitude:
"+str(long2)+", latitude: "+str(lat2)+"},0.05) yield node MATCH p=(node)-
[r:TEMPS_TRAJ]->(targ) MATCH o=(targ)-[s:REVERSE_TRAJ]->(node) return
node.longitude, node.latitude, targ.longitude, targ.latitude,r.Cout,s.Cout,
node.noeuds_id, targ.noeuds_id limit 2").to_table()
tempsCloseHGns = float(TCloseHG-AvantCloseHG)
tempsCloseHGms = float(tempsCloseHGns/1000000)
print("Temps d'execution du closest spatiale en nanoseconde = %d\n"
%tempsCloseHGns )
print("Temps d'execution du closest spatiale en milliseconde = %d\n"
%tempsCloseHGms )

#Insertion des résultats de la requête dans des matrices pour pouvoir des
utiliser
RelsClosest1 = np.array(ClosestRel1)
print(RelsClosest1)
RelsClosest2 = np.array(ClosestRel2)
print(RelsClosest2)

#Code permettant de déterminer la distance entre une droite et un points
#Il commence par déterminer l'équation de droite type y=ax+b
def DistancePtsRel(pts, relation, iteration):
  Xpts1rel = float(relation[iteration,0])
  print(Xpts1rel)
  Ypts1rel = float(relation[iteration,1])
  print(Ypts1rel)
  Xpts2rel = float(relation[iteration,2])
  print(Xpts2rel)
  Ypts2rel = float(relation[iteration,3])
  print(Ypts2rel)

  a = (Ypts2rel - Ypts1rel)/(Xpts2rel - Xpts1rel)
  b = Ypts1rel - (a*Xpts1rel)
    #déterminer la distance à une droite
  Xpts = pts[:,0]
  Ypts = pts[:,1]
  CalcDistRel = math.fabs((Ypts - a*Xpts - b)/math.sqrt(pow(a,2)+1))
  return (CalcDistRel)
print("Selection de l'arc la plus proche")

#Sélection de l'arc la plus proche de notre point et export des
identifiants et coordonnées des noeuds caractérisants l'arc le plus proche
def SelectClosestRel(Pts, RelsClosest):
  DistanceMin = 10
  dim = RelsClosest.shape
  i = dim[0]
  for x in range(i-1):
    Dist = DistancePtsRel(Pts, RelsClosest, x)
    if Dist < DistanceMin:
      DistanceMin = Dist
      i = x
      print("dist")
      print(Dist)
```

```
      print("distmin")
      print(DistanceMin)
    else:
      continue
  print()
  DroiteRelation = RelsClosest[i-1,:]
  Xpts1rel = float(DroiteRelation[0])
  Ypts1rel = float(DroiteRelation[1])
  Xpts2rel = float(DroiteRelation[2])
  Ypts2rel = float(DroiteRelation[3])
  id1 = int(DroiteRelation[6])
  id2 = int(DroiteRelation[7])
  CoutTraj = float(DroiteRelation[4])
  CoutRev = float(DroiteRelation[5])
  return [id1, id2, CoutTraj, CoutRev, Xpts1rel, Ypts1rel, Xpts2rel,
Ypts2rel]

#Applications des méthodes réalisées précédemment et insertion du résultat
dans des matrices
CloseRelPts1 = np.array(SelectClosestRel(Pts1, RelsClosest1))
CloseRelPts2 = np.array(SelectClosestRel(Pts2, RelsClosest2))

#Affichage des résultats obtenus dans la sélection de l'arc le plus proche
des points extérieurs
print("\n Relation la plus proche du point 1 : ")
print(CloseRelPts1)
print("\n Entre les points")
print(CloseRelPts1[0])
print(CloseRelPts1[1])
print("\n Relation la plus proche du point 2 : ")
print(CloseRelPts2)
print("\n Entre les points")
print(CloseRelPts2[0])
print(CloseRelPts2[1])

#Extraction des coordonnées des extrémites des arcs obtenues
LongRel1Ndep = float(CloseRelPts1[4])
LatRel1Ndep = float(CloseRelPts1[5])
IdRel1dep = '1000001N1'
LongRel1Narr = float(CloseRelPts1[6])
LatRel1Narr = float(CloseRelPts1[7])
IdRel1arr = '1000001N2'

LongRel2Ndep = float(CloseRelPts2[4])
LatRel2Ndep = float(CloseRelPts2[5])
LongRel2Narr = float(CloseRelPts2[6])
LatRel2Narr = float(CloseRelPts2[7])
IdRel2dep = '1000002N1'
IdRel2arr = '1000002N2'

#Insertion des coordonnées des points extérieurs au graphe et des
extrémités des arcs dans un csv pour permettre la visualisation
csvfile = "C:/xampp/htdocs/Neo4j/PtsNodesTest3.csv"
```

```
with open(csvfile, "a") as fp:
  wr = csv.writer(fp, dialect='excel')
  wr.writerow(['Labels', 'Lat', 'Long'])

  lat = lat1
  long = long1
  id = 'Pts1'
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)

  lat = lat2
  long = long2
  id = 'Pts2'
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)

  lat=LatRel1Ndep
  long=LongRel1Ndep
  id=IdRel1dep
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)

  lat=LatRel1Narr
  long=LongRel1Narr
  id=IdRel1arr
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)

  lat=LatRel2Ndep
  long=LongRel2Ndep
  id=IdRel2dep
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)

  lat=LatRel2Narr
  long=LongRel2Narr
  id=IdRel2arr
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)



#Calcul des coordonnées du point sur la relation la plus proche
def CoordPtsduGraphe(CloseRel, pts):
  Xpts = pts[:,0]
  Ypts = pts[:,1]
  aRel = (CloseRel[7] - CloseRel[5])/(CloseRel[6] - CloseRel[4])
  bRel = CloseRel[5] - (aRel*CloseRel[4])
  aHG = -1/aRel
  bHG = Ypts - (aHG*Xpts)
  XptsG = (bHG-bRel)/(aRel-aHG)
  YptsG = aRel*((bHG-bRel)/(aRel-aHG))+bRel
```

```
  #Bornage des coordonnées du nouveau noeuds pour limiter aux extrémités
des l'arc
  if CloseRel[4]>CloseRel[6]:
    if XptsG>CloseRel[4]:
      XptsG = CloseRel[4]
    if XptsG<CloseRel[6]:
      XptsG = CloseRel[6]
  if CloseRel[6]>CloseRel[4]:
    if XptsG>CloseRel[6]:
      XptsG = CloseRel[6]
    if XptsG<CloseRel[4]:
      XptsG = CloseRel[4]
  if CloseRel[5]>CloseRel[7]:
    if YptsG>CloseRel[5]:
      YptsG = CloseRel[5]
    if YptsG<CloseRel[7]:
      YptsG = CloseRel[7]
  if CloseRel[7]>CloseRel[5]:
    if YptsG>CloseRel[7]:
      YptsG = CloseRel[7]
    if YptsG<CloseRel[5]:
      YptsG = CloseRel[5]
  return np.array([XptsG, YptsG])

#Coordonnées des nouveaux noeuds insérés dans les arcs les plus proche des
points extérieurs au graphe
Pts1G = CoordPtsduGraphe(CloseRelPts1, Pts1)
pts1G=[1000001, float(Pts1G[0]), float(Pts1G[1])]
Pts2G = CoordPtsduGraphe(CloseRelPts2, Pts2)
pts2G=[1000002, float(Pts2G[0]), float(Pts2G[1])]

delPoints = NeoGraph.run("MATCH (n:PtsExt) detach delete n")

#Insertion des nouveaux noeuds obtenus dans un fichier csv pour permettre
la visualisation
csvfile = "C:/Users/home/.Neo4jDesktop/neo4jDatabases/database-7aa7d067-
3bc4-498e-8491-0a1888107785/installation-3.5.6/import/noeuds.csv"
with open(csvfile, "a") as fp:
  wr = csv.writer(fp, dialect='excel')
  wr.writerow(['noeuds_id', 'longitude', 'latitude'])
  wr.writerow(pts1G)
  wr.writerow(pts2G)
insertPtsG = NeoGraph.run("LOAD CSV WITH HEADERS FROM 'file:///noeuds.csv'
AS row CREATE (n:SpatNoeuds) SET n=row,n.longitude =
toFloat(row.longitude),n.latitude = toFloat(row.latitude)")

#Transformation des données pour leur utilisation dans une requête py2neo
id1pts1 = str(CloseRelPts1[0])[:-2]
id1pts2 = str(CloseRelPts1[1])[:-2]
id2pts1 = str(CloseRelPts2[0])[:-2]
id2pts2 = str(CloseRelPts2[1])[:-2]
print(id1pts1, id1pts2,id2pts1, id2pts2)
```

```
#Insertion des arcs entre les nœuds ajoutés et les extrémités de l'arc le
plus proche. On définit le poids des relations comme étant égale a la
distance euclidienne entre les points pour ensuite pondéré avec le coût de
la l'arc le plus proche.
insertRelation = NeoGraph.run("MATCH(n:SpatNoeuds),(m:SpatNoeuds) WHERE
n.noeuds_id='1000001' AND m.noeuds_id='"+id1pts1+"' WITH SQRT((n.longitude
- m.longitude)^2+(n.latitude - m.latitude)^2) AS proxi, n, m CREATE (n)-
[r:DistanceEucli]->(m) SET r.Dist = proxi")
insertRelation = NeoGraph.run("MATCH(n:SpatNoeuds),(m:SpatNoeuds) WHERE
n.noeuds_id='1000001' AND m.noeuds_id='"+id1pts2+"' WITH SQRT((n.longitude
- m.longitude)^2+(n.latitude - m.latitude)^2) AS proxi, n, m CREATE (n)-
[r:DistanceEucli]->(m) SET r.Dist = proxi")
insertRelation = NeoGraph.run("MATCH(n:SpatNoeuds),(m:SpatNoeuds) WHERE
n.noeuds_id='1000002' AND m.noeuds_id='"+id2pts1+"' WITH SQRT((n.longitude
- m.longitude)^2+(n.latitude - m.latitude)^2) AS proxi, n, m CREATE (n)-
[r:DistanceEucli]->(m) SET r.Dist = proxi")
insertRelation = NeoGraph.run("MATCH(n:SpatNoeuds),(m:SpatNoeuds) WHERE
n.noeuds_id='1000002' AND m.noeuds_id='"+id2pts2+"' WITH SQRT((n.longitude
- m.longitude)^2+(n.latitude - m.latitude)^2) AS proxi, n, m CREATE (n)-
[r:DistanceEucli]->(m) SET r.Dist = proxi")

#Extraction de la valeur de distance eucledienne pour le premier noeud et
transformation de la données pour l'utilisation dans Python
DistEucliN1 = str(NeoGraph.run("MATCH
(Source:SpatNoeuds{noeuds_id:'1000001'})MATCH (Source)-[r1:DistanceEucli]-
>(Target) Return r1").data())
print(DistEucliN1)
part1, part2, dist, ValueDist1, part3, part4, dist2, ValueDist2 =
DistEucliN1.split()
CT1 = float(CloseRelPts1[2])
CRT1 = float(CloseRelPts1[3])
Dist11 = float(ValueDist1.split('}',1)[0])
Dist12 = float(ValueDist2.split('}',1)[0])
print(CT1, CRT1, Dist11, Dist12)

#Extraction de la valeur de distance eucledienne pour le second noeud et
transformation de la données pour l'utilisation dans Python
DistEucliN2 = str(NeoGraph.run("MATCH
(Source:SpatNoeuds{noeuds_id:'1000002'})MATCH (Source)-[r1:DistanceEucli]-
>(Target) Return r1").data())
print(DistEucliN2)
part1, part2, dist, ValueDist3, part3, part4, dist2, ValueDist4 =
DistEucliN2.split()
CT2 = float(CloseRelPts2[2])
CRT2 = float(CloseRelPts2[3])
Dist21 = float(ValueDist3.split('}',1)[0])
Dist22 = float(ValueDist4.split('}',1)[0])
print(CT2, CRT2, Dist21, Dist22)

#Méthode permettant de définir le poid des relations en fonction du Cout de
base et de la distance euclienne mesurée
def CalculPoids(CT, Dist1, Dist2):
  if Dist1 == 0:
```

```
    PoidsTraj1 = 0
    PoidsTraj2 = CT
  if Dist2 == 0:
    PoidsTraj2 = 0
    PoidsTraj1 = CT
  else:
    PoidsTraj1 = CT*(Dist1/(Dist1+Dist2))
    PoidsTraj2 = CT*(Dist2/(Dist1+Dist2))
  return [PoidsTraj1, PoidsTraj2]

#Application de la méthode réalisé au dessus
Traj1 = CalculPoids(CT1, Dist11, Dist12)
RTraj1 =CalculPoids(CRT1, Dist11, Dist12)
Traj2 = CalculPoids(CT2, Dist21, Dist21)
RTraj2 =CalculPoids(CRT2, Dist21, Dist21)

print(Traj1, RTraj1)
print(Traj2, RTraj2)

#Extraction des coordonnées des extrémités des arcs les plus proches des
nouveaux noeuds intégrés
NoeudsRel1 = np.array(NeoGraph.run("MATCH
(N1:SpatNoeuds{noeuds_id:'1000001'})-[d1:DistanceEucli]->(Source)-
[t1:TEMPS_TRAJ]->(Target)<- [d2:DistanceEucli]-
(N2:SpatNoeuds{noeuds_id:'1000001'}) return N1.noeuds_id, Source.noeuds_id,
Target.noeuds_id").to_table())
NoeudsRel2 = np.array(NeoGraph.run("MATCH
(N1:SpatNoeuds{noeuds_id:'1000002'})-[d1:DistanceEucli]->(Source)-
[t1:TEMPS_TRAJ]->(Target)<- [d2:DistanceEucli]-
(N2:SpatNoeuds{noeuds_id:'1000002'}) return N1.noeuds_id, Source.noeuds_id,
Target.noeuds_id").to_table())
print(NoeudsRel1)
print(NoeudsRel2)

#Insertion des identifiants des futurs arcs avec le coup respectif calculé
pour préparer l'insertion
Cout1S =[int(NoeudsRel1[:,0]), int(NoeudsRel1[:,1]), RTraj1[0]]
Cout1T =[int(NoeudsRel1[:,0]), int(NoeudsRel1[:,2]), Traj1[1]]
Cout2S =[int(NoeudsRel2[:,0]), int(NoeudsRel2[:,1]), RTraj2[0]]
Cout2T =[int(NoeudsRel2[:,0]), int(NoeudsRel2[:,2]), Traj2[1]]

#Ecriture des informations précédentes dans un csv afin d'être importées
dans la base de données
csvfile = "C:/Users/home/.Neo4jDesktop/neo4jDatabases/database-7aa7d067-
3bc4-498e-8491-0a1888107785/installation-3.5.6/import/Cost_rel.csv"
with open(csvfile, "a") as fp:
  wr = csv.writer(fp, dialect='excel')
  wr.writerow(['source', 'target', 'Dist'])
  wr.writerow(Cout1S)
  wr.writerow(Cout1T)
  wr.writerow(Cout2S)
  wr.writerow(Cout2T)
```

```
#Insertion des relations entres les nouveaux noeuds et les extrémités des
arcs les plus proches
insertRel = NeoGraph.run("LOAD CSV WITH HEADERS FROM 'file:///Cost_rel.csv'
AS line MATCH (Source:SpatNoeuds {noeuds_id: line.source}) MATCH
(Target:SpatNoeuds {noeuds_id: line.target}) CREATE (Source)-
[r:TEMPS_TRAJ]->(Target) SET r.Cout = toFloat(line.Dist);")

#Calcul et mesure du temps pour réaliser le plus court chemin par rapport
aux deux noeuds insérés dans le graphe et exportation des résultats du plus
court chemin
print("Réalisation du plus court chemin")
AvantPath = time.perf_counter_ns()
ShortestPath = NeoGraph.run("MATCH (a:SpatNoeuds {noeuds_id: '1000001'})
MATCH (b:SpatNoeuds {noeuds_id: '1000002'}) CALL apoc.algo.dijkstra(a, b,
'TEMPS_TRAJ', 'Cout') YIELD path, weight RETURN nodes(path),
length(path);").to_table()
AprPath = time.perf_counter_ns()
extraShortP = np.array(ShortestPath)
tempspathns = float(AprPath-AvantPath)
tempspathms = float(tempspathns/1000000)

print("Temps d'execution du shortestpath en nanoseconde = %d\n"
%tempspathns )
print("Temps d'execution du shortestpath en milliseconde = %d\n"
%tempspathms )

#Suppression des différents fichiers et noeuds ajoutés dans la base de
données pour facilité la répétition de l'exécution du code
delPoints = NeoGraph.run("match (n:SpatNoeuds) where n.noeuds_id='1000002'
or n.noeuds_id='1000001' detach delete n")
os.remove('C:/Users/home/.Neo4jDesktop/neo4jDatabases/database-7aa7d067-
3bc4-498e-8491-0a1888107785/installation-3.5.6/import/noeuds.csv')
os.remove('C:/Users/home/.Neo4jDesktop/neo4jDatabases/database-7aa7d067-
3bc4-498e-8491-0a1888107785/installation-3.5.6/import/Cost_rel.csv')

#Mesure du nombre de noeud dans le plus court chemin
i = (extraShortP[0,1])-1
print(i)
SliceShort = str(extraShortP[0,0]).split(',')

#Exportation des noeuds insérés dans le graphe dans un CSV pour la
visualisation
csvfile = "C:/xampp/htdocs/Neo4j/NewNodesTest3.csv"
with open(csvfile, "a") as fp:
  wr = csv.writer(fp, dialect='excel')
  wr.writerow(['Labels', 'Lat', 'Long'])
  latitude=SliceShort[0]
  longitude=SliceShort[1]
  identifiant=SliceShort[2]
  lat=(latitude)[33:]
  long=(longitude)[12:]
  id=(identifiant)[13:-3]
  Marker = [str(id), float(lat), float(long)]
```

```
  wr.writerow(Marker)

  latitude=SliceShort[(i-1)*10+13]
  longitude=SliceShort[(i-1)*10+14]
  identifiant=SliceShort[(i-1)*10+15]
  lat=(latitude)[33:]
  long=(longitude)[12:]
  id=(identifiant)[13:-4]
  Marker = [str(id), float(lat), float(long)]
  wr.writerow(Marker)

#Exportation du résultat du plus court chemin avec la liste des noeuds. La
liste est insérée dans un CSV pour permettre la visualisation
csvfile = "C:/xampp/htdocs/Neo4j/CoordsNeo4jSTest3.csv"
with open(csvfile, "a") as fp:
  wr = csv.writer(fp, dialect='excel')
  wr.writerow(['Labels', 'Lat', 'Long'])

  for x in range(i):
    latitude=SliceShort[x*10+8]
    longitude=SliceShort[x*10+9]
    identifiant=SliceShort[x*10+10]
    lat=(latitude)[11:]
    long=(longitude)[12:]
    id=(identifiant)[13:-1]
    Marker = [str(id), float(lat), float(long)]
    wr.writerow(Marker)


#Fin du code et du chronomètre pour mesurer le temps total nécessaire à la
réalisation du code
print("Fin du programme")
FinProg = time.perf_counter_ns()
tempsprogns = float(FinProg-DebutQ1)
tempsprogsms = float(tempsprogns/1000000)
print("Temps d'execution du programme complet en nanoseconde = %d\n"
%tempsprogns )
print("Temps d'execution du programme complet en milliseconde = %d\n"
%tempsprogsms )
```