UNIVERSITÉ LAVAL

Travail longitudinal

Remise et présentation écrite du produit minimum viable

Équipe 18

Nom	Numéro d'identification
Alec Vincent	111 263 912
Camille Gervais	111 235 227
David Ferland	111 270 836
Dietz-Bénony Awoussi	111 230 262

Bases de données avancées

GLO-4035

David Beauchemin

20 décembre 2022

Table des matières

1. Introduction

Suite à une promenade en vélo entre amis à Cornwall, la présente équipe a eu l'idée d'implémenter une application permettant de planifier des visites en restaurants lors d'un parcours. Pour développer une application conforme aux normes logicielles de maintenabilité, d'extensibilité et de fiabilité, votre financement serait un atout précieux et permettrait de réaliser un projet d'envergure bénéfique pour tous les partis concernés. En effet, étant donné un point de départ géographique, il serait très intéressant de pouvoir spécifier des types de restaurants à visiter et une distance de vélo à parcourir dans le but d'obtenir un parcours épicurien idéal et personnalisé pour nos goûts. Ceci étant dit, une telle application comporte plusieurs aspects technologiques complexes et soumis à différents niveaux de risques. Plus particulièrement, tout ce qui concerne la gestion des données impose une analyse et une planification approfondie car il s'agit de la plaque tournante d'une telle application de ce genre. En effet, que ce soit l'extraction initiale des données, l'acquisition incrémentale pour que notre application soit toujours à jour et conforme à la réalité ou encore la gestion de la sécurité, tous ces aspects de l'application s'accompagnent de risques réels pouvant nuire au bon fonctionnement du logiciel. La solution retenue est une application web (serveur Flask écrit en Python) utilisant plusieurs différents *frameworks.* Plus spécifiquement, deux bases de données distinctes sont utilisées pour gérer la persistance de l'application, à savoir une base de données *MongoDB* pour les restaurants et une *Neo4j* pour simuler le réseau routier de la ville de Cornwall. À l'aide de requêtes webs, l'utilisateur pourra donc, en spécifiant un point de départ et des types de restaurants désirés, obtenir le parcours épicurien optimal à l'aide d'une algorithme de création de parcours qui sera détaillé à la section 5 du présent rapport. En plus de la description de cet algorithme, ce rapport présentera plus en détails la stratégie d'acquisition initiale des données, les technologies utilisées dans l'application, les détails du processus d'extraction, de transformation et de conversion des données (ETL) ainsi qu'une explication du plan d'expansion de l'application pour gérer un éventuel volume d'utilisateur beaucoup plus grand, faisant de votre investissement un atout financier idéal et sécuritaire.

2. Stratégie d'acquisition des données

2.1 Source et méthode d'extraction

La ville sélectionnée pour l'application est Cornwall (Ontario, Canada). Deux jeux de données seront utilisés pour la création du trajet épicurien: un premier jeu pour les restaurants et un deuxième pour les routes. Le jeu de données choisi pour les restaurants est celui du site Yelp.

Les données sont accessibles par l'API de Yelp[1] et sont mises à jour régulièrement. Ce jeu de données comprend tous les champs qui seront nécessaires à la création d'un trajet: nom et adresse du restaurant, coordonnées en latitude et longitude, ainsi que le type culinaire et/ou de service du restaurant.

Le jeu de données choisi pour les routes, lui, est disponible pour téléchargement en format GeoJSON sur le site gouvernemental Ontario GeoHub.[2] Ce sont les routes de l'Ontario au complet qui y sont répertoriées, ceci par le Ministère des Richesses naturelles et des Forêts de l'Ontario. Les données ont été mises à jour pour la dernière fois en septembre 2022. Chaque entrée du jeu représente un segment d'une rue et inclut des coordonnées en latitude et longitude à fréquence régulière. Certains champs qui pourraient s'avérer utiles dans le cadre du projet incluent les champs de la municipalité, le nom complet de la rue, ses coordonnées géospatiales et la longueur du segment en mètres.

2.2 Présentation d'exemples de données sources

La figure 1 (voir l'Annexe) présente deux exemples d'entrées du jeu de données des restaurants. Tous les champs sont obligatoires, donc remplis, et seuls les restaurants ouverts sont retournés par l'API. Ils peuvent donc tous être utilisés. La source d'erreur principale vient de la façon dont l'API de Yelp fonctionne quant à l'extraction des données. Un point géographique et un rayon sont utilisés pour créer un cercle, dans lequel on vient chercher les restaurants. Il est donc possible que certains restaurants plus éloignés, en bordure de la ville, n'aient pas été inclus.

La figure 2 (voir l'Annexe) présente deux exemples d'entrées du jeu de données des routes. Il est possible de constater que comme la plupart des champs liés au nom, numéro, et municipalité d'une route sont optionnels, il est possible de ne pas avoir ces informations. Le champ "geometry" qui inclut les coordonnées, lui, est cependant toujours présent. La source d'erreur principale de ces données vient du fait qu'il est impossible de savoir si une route est sécuritaire et apte à être traversée en vélo. L'application pourrait donc générer un trajet qui inclut une route où il n'est pas possible de voyager en vélo (par exemple une autoroute sans piste cyclable).

---

[1] Yelp (2022). *Yelp Fusion*. https://fusion.yelp.com/ (consulté le 6 novembre 2022).

[2] Ontario Ministry of Natural Resources and Forestry (2022). *Ontario Road Network (ORN) Segment With Address*. https://geohub.lio.gov.on.ca/datasets/mnrf::ontario-road-network-orn-segment-with-address (consulté le 2 octobre 2022).

3. Technologies utilisées

3.1 Langage de programmation et framework choisi

Python et Flask ont été choisis comme langage de programmation et framework respectivement. Incontournable en informatique aujourd'hui, Python est un langage interprété fiable et bien établi qui est utilisé pour faire à peu prêt tout dans le développement logiciel. Flask est apprécié dans la communauté à cause de la variété de structure qu'il offre dans le développement d'un serveur backend. Il offre beaucoup d'extensions très utiles comme des systèmes d'authentification, des formulaires de validation et bien d'autres outils. Il est à noter que Python est très utilisé pour le traitement de données. Vu la nature du projet et la quantité de données à analyser et à traiter, Python offre l'opportunité de gérer le serveur avec Flask et la capacité d'analyser les données avec les fonctions native de python ou avec des librairies comme scikit learn.

3.2 Bases de données

Le présent projet est principalement un projet d'analyse et traitement de données. Pour faire cela, il est évident que le choix des bases de données doit être motivé par le contexte et le but du projet. Les données des restaurants recueillis précédemment seront stockées dans une base de données Mongo. Une raison essentielle qui a motivé ce choix est que les données de restaurants varient beaucoup d'un restaurant à un autre et la structure en document de Mongo nous permet de pallier cela. En ce qui concerne l'acquisition des données de parcours de l'utilisateur, nous ingérons le tout dans Mongo pour utiliser la fonction de GeoNear et ainsi associer les restaurants aux routes. Dans le contexte du projet, Neo4j sera utilisé pour calculer efficacement les parcours désirés par l'utilisateur car il contiendra les données routières et la localisation des restaurants. En effet, grâce à sa structure en graphe il y a un avantage à effectuer les recherches géospatiales avec Neo4j pour réduire le temps de calcul. Donc, l'application Flask interface seulement avec Neo4j car c'est là que tout l'information nécessaire à l'utilisateur se retrouve.

4. Détails du processus d'extraction, de transformation et de conversion (ETL)

4.1 Processus d'acquisition initiale des données

La première étape du processus d'acquisition des données est d'extraire les données des sites Yelp et Geohub sous format JSON. Nous utilisons l'API de Yelp pour sortir les données relatives aux restaurants de notre ville choisie, soit Cornwall. Les données sur la carte des routes de notre ville sont, quant à elles, extraites du site Geohub qui permet de filtrer facilement les

données pour avoir seulement celles qui nous intéressent. Ensuite, le fichier contenant les données des restaurants de la ville de Cornwall est filtré par un script Python pour enlever les informations inutiles pour le projet comme les reviews et autres informations relatives à Yelp. Nous conservons seulement l'identifiant, le nom, le type, l'adresse et les coordonnées des restaurants. Une fois ce problème réglé, nous ingérons les données concernant les informations restaurants dans la base de données Mongo. Ensuite, pour les données cartographiques, afin d'identifier les intersections entre les routes qui sont représentées par des segments de type "Line" ou "MultiLine" nous parcourons chaque route pour regarder si elles ont une intersection avec les autres routes grâce à un script Python. Si oui, nous conservons ce point comme étant une intersection et chargeons toutes les intersections dans leur propre collection Mongo. Une fois cette opération terminée, nous pouvons charger les données routière dans notre base de données. Chaque type de données, soit routières, de restaurants ou intersections est alors contenue dans sa propre collection. Notre base de données Mongo devient alors notre base de données maître qui contient toutes les données utiles pour la réalisation de notre application. Ces données sont alors immuables.

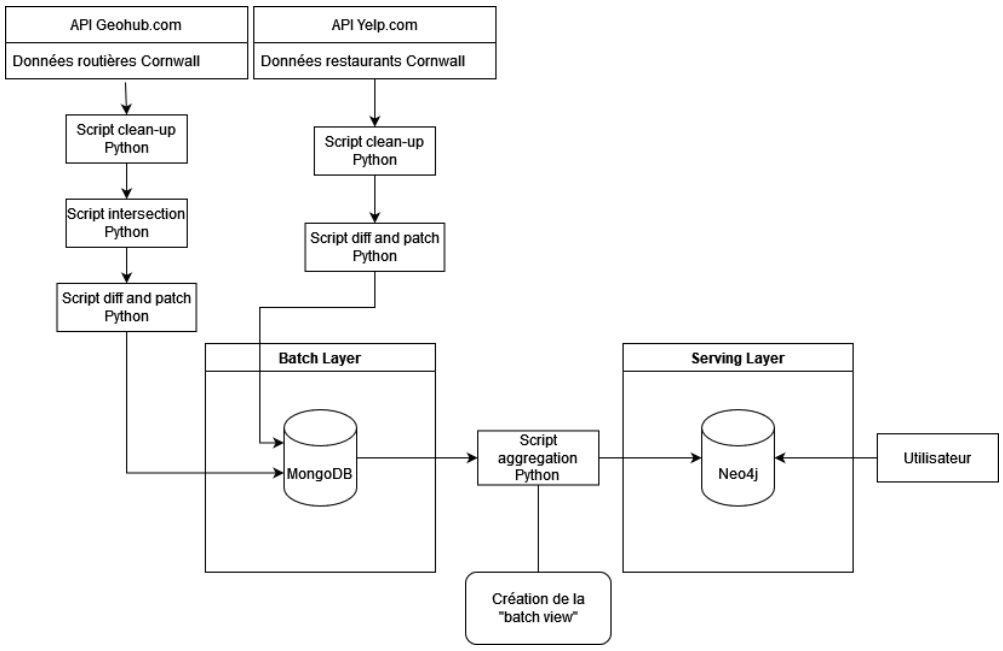
4.2 Processus d'acquisition incrémental des données

Pour le processus d'acquisition incrémental des données, puisque nos deux sources de données sont plutôt statiques (la construction/destruction de routes et de restaurants n'est pas un flot continu de données très contraignant), nous pensons seulement faire des requêtes pour de nouvelles données une fois par semaine. La requête se fait dans le même script Python que le nettoyage de données grâce à un "scheduler". Une fois nettoyées, les données des restaurants sont chargées dans un autre script Python qui s'occupe d'appliquer un algorithme de "diff and patch". Celui-ci passe au travers de tous les objets JSON pour faire une requête vers Mongo avec son identifiant unique et vérifier si les informations entre les deux documents sont identiques, si oui on ne fait rien sinon on met à jour le document dans Mongo. Si la requête ne retourne rien, on peut insérer ce nouveau document dans la base de données. Pour les données cartographiques, le calcul d'intersection est refait au complet et le même type d'algorithme "diff and patch" est appliqué sur les intersections. Pour finir, encore une fois, nous appliquons l'algorithme "diff and patch" sur les données routières pour identifier les différences et mettre à jour Mongo.

4.3 Processus de transformation des données

Le processus de transformation des données commence au niveau des scripts de nettoyage en Python, nous les utilisons pour enlever toutes les données qui ne sont pas utiles à notre application. Bien évidemment nous conservons toutes les données nécessaires à la reconstruction des informations entreposées dans la couche de service (serving layer). Pour les restaurants il s'agit de son identifiant unique, son nom, ses types, son adresse et ses coordonnées géographiques. Alors que pour les routes, il s'agit de son identifiant unique, son nom, sa longueur et ses coordonnées. Une fois que les données sortent du script d'acquisition des données alors elles sont traitées comme étant immuables, elles peuvent seulement être mises à jour ou effacées si les données provenant de nos API sont différentes ou manquantes. Ensuite, une fois intégrés dans la base de données Mongo, les données routières de la ville choisie sont manipulées pour extraire les intersections de celles-ci grâce à un script Python. Les restaurants seront ensuite mis en relation avec ces intersections de routes grâce à la fonctionnalité GeoNear de Mongo. Lorsque cette étape est terminée, notre batch view est alors complète car nous avons maintenant toutes les informations nécessaires pour modéliser notre carte de la ville de Cornwall ainsi que ses restaurants dans Neo4j. La batch view qui contient les informations agrégées est alors sauvegardée dans Mongo. Aussitôt qu'il y a une mise à jour de cette table de données agrégées, un script Python, qui agit en tant qu'abonné à cette table, est notifié et s'occupe de transcrire ces données vers Neo4J.

4.4 Schéma de la pipeline d'ETL



4.5 Avantages du modèle d'ETL

Tel que présentement conçu et envisagé, le modèle d'ETL offre plusieurs avantages, particulièrement au niveau de la flexibilité. Puisque chaque segment comporte certains attributs, il sera éventuellement très facile d'en rajouter pour filtrer les recherches de parcours en fonction de certains critères ciblés. Par exemple, nous pourrions facilement rajouter un attribut ''priorité cycliste'' à certains segments pour obtenir des parcours plus sécuritaires circulant strictement sur ce genre de rues. Cette flexibilité sera cruciale pour le développement éventuel de nouvelles fonctionnalités. En cas de besoin de rapidité, l'ajout de restaurants et de segments est aussi relativement simple, car il suffit d'ajouter rapidement le nœud adéquat dans la base de données Neo4j pour considérer ces nouvelles destinations dans le calcul de chemin. De plus, le fait de limiter la base de données Neo4j au ''simple'' calcul de trajets permet de simplifier grandement la modélisation des données, car celle-ci ne contient pas d'information superflue. L'utilisation de deux bases de données différentes pour la Batch Layer et la Serving Layer rajoute de la robustesse en cas de défaillance d'une des bases de données.

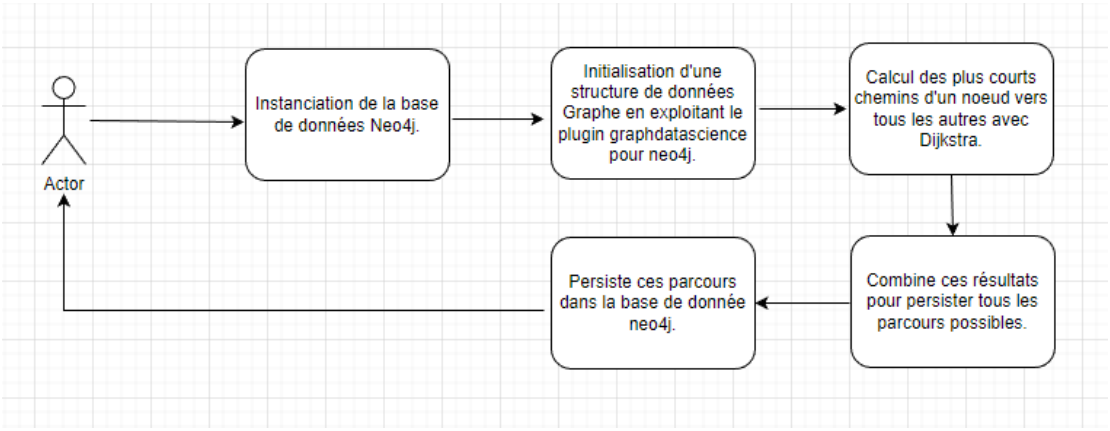
5. Détails de la pipeline de données

5.1 Création d'un parcours

Le point crucial de notre application est que les parcours ne sont pas créés spécifiquement lorsque l'utilisateur en fait une demande. En effet, la taille des données étant actuellement relativement limitée, tous les parcours possibles sont initialement créés dès le lancement de l'application et stockés dans la base de données Neo4j. Cela a pour effet de grandement augmenter le temps de chargement initial, mais permet d'effectuer des requêtes d'obtention de parcours rapidement. Bien sûr, une telle approche serait à réévaluer si le nombre d'intersections devait augmenter rapidement, par exemple en changeant de ville. Ceci étant dit, c'est l'approche que nous avons jugé la plus simple pour un produit minimum viable.

Plus spécifiquement, les bases de données Neo4j étant des bases de données orientées par graphe, cette méthode permet l'exploitation de librairies et de fonctions très utiles pour le calcul de plus courts chemins (les graphes étant une structure de données naturelles à de tels algorithmes). Ainsi, en exploitant le plugin *graphdatascience* pour Neo4j, la méthode pour la création d'un parcours à l'interne est relativement simple et rapide. En effet, nous commençons par créer notre graphe à l'aide de la méthode *project* de cette librairie, puis nous créons/persistons les parcours (ainsi que tous leurs attributs désirés) à l'aide d'une requête Neo4j relativement complexe donnée en annexe (figure 3). Notons notamment l'utilisation de

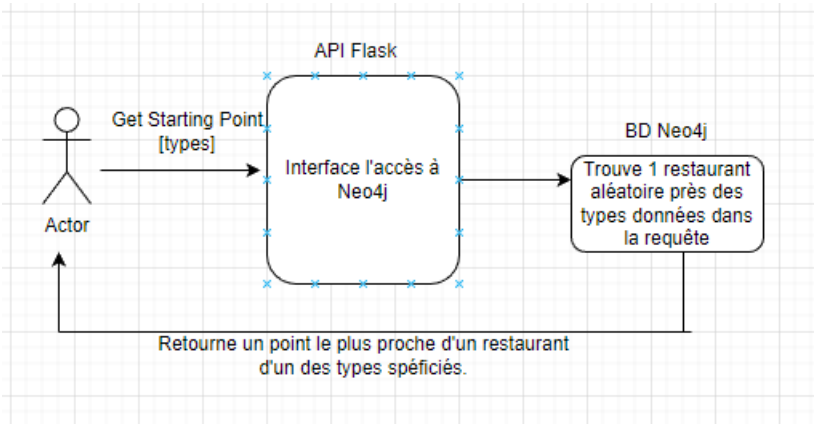
l'algorithme de Dijkstra pour le calcul des plus courts chemins entre les intersections. Le diagramme de flux circulaire suivant explique notre processus de création et de persistance des parcours possibles pour un utilisateur:



5.2 Obtention d'un parcours

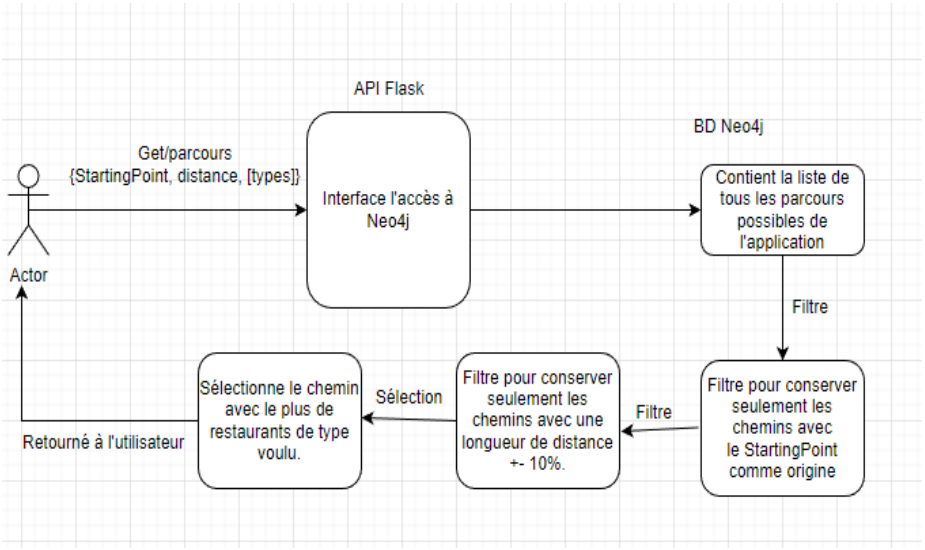
Concernant le processus principal d'obtention d'un parcours, celui-ci est constitué de deux principales étapes, la deuxième étant directement dépendante de la première:

1. Obtenir un point de départ via une requête GET/starting_point sur le serveur web en spécifiant une liste de types de restaurants désirés. Tel que montré sur la figure ci-dessous, le serveur répond avec les coordonnées géographiques correspondants à l'intersection située le plus près d'un des restaurants des types spécifiés.



2. Ayant obtenu ce point de départ, l'utilisateur peut maintenant obtenir un parcours à l'aide d'une requête GET/parcours en fournissant le point de départ, la distance à parcourir ainsi que les types de restaurants désirés. Tel que décrit à la section précédente, à ce moment, l'application contient déjà en mémoire tous les parcours possibles ayant un certain point comme origine et comme destination, ainsi que les intersections parcourus

pour s'y rendre et la longueur totale du parcours. Ainsi, avec un point de départ et une distance à parcourir ± 10%, nous obtenons une longue liste de parcours possibles. La dernière étape consiste à sélectionner parmi ces parcours celui ayant le plus grand nombre de restaurants spécifiés par l'utilisateur et à l'envoyer comme réponse selon le format voulu. La figure ci-dessous résume brièvement ce fonctionnement.



6. Explication du plan d'expansion

6.1 Description

Après le succès de la preuve de concept, on souhaite continuer de développer l'application dans le but de la rendre disponible au plus de personnes possibles. On souhaite ainsi ajouter graduellement des villes au jeu de données, jusqu'à éventuellement couvrir l'Ontario au complet. Le jeu de données des routes étant fourni par le gouvernement ontarien pour la totalité du territoire, il sera aisé d'ajouter à notre jeu sans avoir de divergences entre les données. L'utilisation de plusieurs centres de données sera nécessaire pour héberger cette nouvelle quantité exponentielle de données. Les deux métriques suivantes seront utilisées pour évaluer la performance des algorithmes de trajet avec les nouvelles données: la médiane du temps de réponse d'une demande de trajet; et le débit de la base de données, c'est-à-dire le nombre de requêtes exécutées par seconde. Tel-quel, l'expansion aura un impact négatif sur ces deux métriques. La grande quantité de données ralentira les algorithmes de trajet, qui devront maintenant parcourir plus de données avant d'arriver au même résultat. Le débit et la médiane du temps de réponse seront donc plus bas. C'est pourquoi on ajoute de la réplication et du partitionnement, afin de contrer l'effet négatif de l'ajout de données.

6.2 Réplication

En raison de la vaste étendue géographique des données et de la nécessité d'avoir plusieurs centres de données, la réplication multi-leader a été choisie. Cela permettra de diviser les données par région géographique de l'Ontario, avec un leader par centre de données. Ainsi, lorsqu'un utilisateur fait une demande de trajet, la demande est envoyée au centre de données le plus proche et le leader de celui-ci est utilisé pour les requêtes de lecture. Comme décrit dans la section 4.2, les données ne sont mises à jour qu'une fois par semaine, étant donné qu'elles sont généralement assez statiques. C'est à ce moment que des conflits peuvent survenir. Le fait que les utilisateurs ne peuvent pas faire de requêtes d'écriture simplifie cependant grandement la résolution des conflits. Les conflits lors de la mise à jour des données d'un leader à un autre peuvent être évités en forçant chaque leader à ne mettre à jour que les données qui se situent dans le périmètre qui leur est attribué. Les changements sont ensuite propagés aux autres leaders de façon asynchrone. Il est acceptable de faire cela puisque tous les leaders vont chercher les données à jour des mêmes sources, c'est-à-dire l'API de Yelp et le site GeoHub.

L'utilisation de plusieurs centres de données et d'une réplication multi-leader permet d'avoir une médiane de temps de réponse plus petite. Puisque l'utilisateur est physiquement plus proche d'un leader, les requêtes de lecture prendront moins de temps à se rendre au leader et à s'exécuter.

6.3 Partitionnement

On utilise un index local pour partitionner les données. L'index primaire correspond aux coordonnées des restaurants et routes et l'index secondaire correspond aux types de restaurants. Ce dernier index n'est pas présent pour les partitions de routes. Les partitions sont organisées de façon à ce que tous les restaurants et routes qu'elles contiennent se trouvent dans une même zone géographique prédéterminée, c'est-à-dire qu'ils sont physiquement proches les uns des autres. On peut donc chercher les restaurants dans une zone précise (par exemple proche du point de départ ou d'un arrêt préalable), ce qui retournera une seule partition. Les restaurants peuvent ensuite être filtrés par type grâce à l'index secondaire. Au niveau du rebalancement des fragments, on choisit la fragmentation dynamique. Cette façon de rebalancer est particulièrement utile pour la fragmentation par plage, ce qui est notre cas ici puisque les partitions se suivent selon les coordonnées qu'elles peuvent contenir. La fragmentation dynamique s'adapte également bien à notre souhait d'ajouter des villes et d'augmenter la taille de la base de données graduellement.

En ajoutant des partitions, on rend l'information plus facile d'accès en lecture. Le débit de la base de données est donc augmenté, et la médiane du temps de réponse est réduite. L'inverse est cependant vrai lors de l'écriture mais puisque l'on écrit dans la base de données seulement lors de la mise à jour hebdomadaire, les impacts sont minimes.

6.4 Sécurité

Notre application, une fois déployée, sera utilisée par, bien évidemment, une grande partie de la population de Cornwall. Bien que nous n'ayons pas, pour l'instant, l'intention de récolter des données sensibles sur nos utilisateurs, il serait bien frustrant qu'un individu malsain accède à notre base de données et décide de corrompre nos données. Pour contrer cela, nous avons décidé d'activer l'authentification pour créer des rôles pour séparer les permissions entre les utilisateurs et les administrateurs de notre application. Les utilisateurs ayant seulement accès à la base de données MongoDB, seule celle-ci comportera un rôle utilisateur. Comme rôle de super utilisateur nous aurons le niveau root qui est le rôle suprême et ne sera accessible qu'aux membres de l'équipe 18. Les deux bases de données, MongoDB et Neo4j, auront ce rôle administrateur. Les administrateurs auront tous les droits sur les bases de données, ce qui leur permettra notamment de faire le processus ETL décrit en section 4. Sur l'instance docker, qui roulera comme étant la base de données de production, le rôle attribué par défaut sera d'utilisateur. Ce rôle sera commun à tous les utilisateurs de l'application et permet seulement de lire les données des trajets. En effet, l'utilisateur n'ajoutera jamais de données à notre application, donc il est inutile de lui donner la permission d'écrire sur celle-ci.

Ensuite, il est à noter que l'utilisation de notre application sera ouverte pour toutes les personnes ayant une connexion à internet, un peu comme Google Map. Étant donné ce fait, il se peut que certaines personnes malsaines interceptent les communications entre notre base de données et notre application pour déduire du trajet de nos utilisateurs. Par exemple, cela pourrait arriver si un utilisateur utilise notre service sur un réseau public mal configuré. Cela aurait des effets dévastateurs puisque nos utilisateurs, potentiellement mineurs pourraient se faire suivre par des personnes malsaines. Pour empêcher cela, nous comptons encrypter les communications entrantes et sortantes de MongoDB avec un certificat SSL pour ainsi empêcher les malfaiteurs de les intercepter et de les lire.

Annexe

Tableau 1 - Estimation des différents coûts suite à l'acquisition de nouvelles données

	Coût sur le hardware requis	Temps d'implémentation	Coût sur les performance	Ressources humaines requises
Ajout d'un restaurant	Faible	Faible	Faible	Modéré (ajout manuel des données, pas d'appel aux apis).
Ajout d'un segment cyclable	Élevé,l'algorithme demande beaucoup d'espace de stockage pour les parcours.	Faible, jusqu'au moment où une révision de l'algorithme sera requise. À ce moment, coût élevé.	Très élevé. L'ajout d'un nouveau segment ajoute beaucoup de parcours possibles.	Modéré (ajout manuel des données, pas d'appel aux apis).

Tableau 2 - Estimation des différents coûts de développement de fonctionnalités

	Coût sur le hardware requis	Temps d'implémentation	Coût sur les performance	Ressources humaines requises
Fonctionnalité exploitant les propriétés des restaurants.	Faible	Faible	Faible	Faible
Fonctionnalité exploitant les propriétés des routes.	Faible	Faible, modélisation neo4j rend les segments très extensibles.	Faible car ne demande pas d'écriture,	Faible
Fonctionnalité exploitant les propriétés des parcours.	Modéré	Faible, à moins que cela demande un changement de l'algorithme de création.	Faible car ne demande pas d'écriture, mais peut mener à de longues lectures.	Faible

Figure 1 - Exemples d’entrées du jeu de données des restaurants

{"id": "qphwrH9bpZ_aGI8f6LRHiQ", "alias": "schnitzels-european-flavours-cornwall", "name": "Schnitzels European Flavours", "image_url": "https://s3-media1.fl.yelpcdn.com/bphoto/QMWhSzix_Ccgz5c77wrDig/o.jpg", "is_closed": false, "url": "https://www.yelp.com/biz/schnitzels-european-flavours-cornwall?adjust_creative=wtPKFq7n6RsCDyhLnpYjQg&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&utm_source=wtPKFq7n6RsCDyhLnpYjQg", "review_count": 19, "categories": [{"alias": "modern_european", "title": "Modern European"}, {"alias": "pubs", "title": "Pubs"}], "rating": 4.0, "coordinates": {"latitude": 45.01786543, "longitude": -74.72861036937894}, "transactions": [], "price": "$$", "location": {"address1": "158 Pitt Street", "address2": "", "address3": "", "city": "Cornwall", "zip_code": "K6J 3P4", "country": "CA", "state": "ON", "display_address": ["158 Pitt Street", "Cornwall, ON K6J 3P4", "Canada"]}, "phone": "+16139388844", "display_phone": "+1 613-938-8844", "distance": 2022.0242717481322}

{"id": "pt63PPTYl0PI-W6OedFU8w", "alias": "joeys-seafood-restaurants-cornwall", "name": "Joey's Seafood Restaurants", "image_url": "https://s3-media3.fl.yelpcdn.com/bphoto/OuaHFQV045AylQKxlp3gpg/o.jpg", "is_closed": false, "url": "https://www.yelp.com/biz/joeys-seafood-restaurants-cornwall?adjust_creative=wtPKFq7n6RsCDyhLnpYjQg&utm_campaign=yelp_api_v3&utm_medium=api_v3_business_search&utm_source=wtPKFq7n6RsCDyhLnpYjQg", "review_count": 8, "categories": [{"alias": "seafood", "title": "Seafood"}, {"alias": "salad", "title": "Salad"}, {"alias": "fishnchips", "title": "Fish & Chips"}], "rating": 4.5, "coordinates": {"latitude": 45.01456056879645, "longitude": -74.7492781}, "transactions": [], "price": "$", "location": {"address1": "830 Second Street W", "address2": "", "address3": "", "city": "Cornwall", "zip_code": "K6J 1H7", "country": "CA", "state": "ON", "display_address": ["830 Second Street W", "Cornwall, ON K6J 1H7", "Canada"]}, "phone": "+16139360169", "display_phone": "+1 613-936-0169", "distance": 2546.7592999182652}

Figure 2 - Exemples d’entrées du jeu de données des routes

{ "type": "Feature", "properties": { "OGF_ID": 1944002730, "OFFICIAL_STREET_NAME": "JOHN TABOR TRAIL", "ABBREVIATED_STREET_NAME": "JOHN TABOR TRAIL", "ALTERNATE_STREET_NAME": null, "DIRECTIONAL_PREFIX": null, "STREET_TYPE_PREFIX": null, "STREET_NAME_BODY": "John Tabor", "STREET_TYPE_SUFFIX": "Trail", "DIRECTIONAL_SUFFIX": null, "ROAD_CLASS": "Collector", "ROAD_ELEMENT_TYPE": "ROAD ELEMENT", "L_FIRST_HOUSE_NUMBER": 262, "L_LAST_HOUSE_NUMBER": 276, "L_HOUSE_NUMBER_STRUCTURE": "Even", "L_ORIGINAL_MUNICIPALITY": "Toronto", "L_STANDARD_MUNICIPALITY": "City of Toronto", "R_FIRST_HOUSE_NUMBER": 261, "R_LAST_HOUSE_NUMBER": 271, "R_HOUSE_NUMBER_STRUCTURE": "Odd", "R_ORIGINAL_MUNICIPALITY": "Toronto", "R_STANDARD_MUNICIPALITY": "City of Toronto", "DIRECTION_OF_TRAFFIC_FLOW": "Both", "ROUTE_NAME_1": null, "ROUTE_NAME_2": null, "ROUTE_NAME_3": null, "ROUTE_NAME_4": null, "ROUTE_NUMBER_1": null, "ROUTE_NUMBER_2": null, "ROUTE_NUMBER_3": null, "SHIELD_TYPE": null, "LENGTH": 157.52, "GEOMETRY_UPDATE_DATETIME": "2002-05-01T00:00:00Z", "EFFECTIVE_DATETIME": "2019-10-02T09:57:46Z", "SYSTEM_DATETIME": "2019-10-11T14:27:15Z", "OBJECTID": 25273169, "SHAPELEN": 0 }, "geometry": { "type": "LineString", "coordinates": [[-79.206348941005558, 43.807003320230706], [-79.20644594104391, 43.807085920240652], [-79.206493241066198, 43.807167720252828], [-79.206553241091839, 43.807240620262782], [-79.206588641108766, 43.807304220272307], [-79.206622941128515, 43.807403820288151], [-79.206645641142188, 43.807476220299741], [-79.2066558411518, 43.807548420311946], [-79.206666141161477, 43.807620620324087], [-79.206664241166408, 43.807683620335219], [-79.206649741168093, 43.807755420348464], [-79.20663514116977, 43.807827220361702], [-79.206632441176865, 43.807917220377576], [-79.206589241180467, 43.808114620414138], [-79.206548141178359, 43.808240020437978], [-79.206519841179585, 43.808356620459733]] } }

{ "type": "Feature", "properties": { "OGF_ID": 1946677043, "OFFICIAL_STREET_NAME": null, "ABBREVIATED_STREET_NAME": null, "ALTERNATE_STREET_NAME": null, "DIRECTIONAL_PREFIX": null, "STREET_TYPE_PREFIX": null, "STREET_NAME_BODY": null, "STREET_TYPE_SUFFIX": null, "DIRECTIONAL_SUFFIX": null, "ROAD_CLASS": "Local / Strata", "ROAD_ELEMENT_TYPE": "ROAD ELEMENT", "L_FIRST_HOUSE_NUMBER": null, "L_LAST_HOUSE_NUMBER": null, "L_HOUSE_NUMBER_STRUCTURE": null, "L_ORIGINAL_MUNICIPALITY": null, "L_STANDARD_MUNICIPALITY": null, "R_FIRST_HOUSE_NUMBER": null, "R_LAST_HOUSE_NUMBER": null, "R_HOUSE_NUMBER_STRUCTURE": null, "R_ORIGINAL_MUNICIPALITY": null, "R_STANDARD_MUNICIPALITY": null, "DIRECTION_OF_TRAFFIC_FLOW": "Both", "ROUTE_NAME_1": null, "ROUTE_NAME_2": null, "ROUTE_NAME_3": null, "ROUTE_NAME_4": null, "ROUTE_NUMBER_1": null, "ROUTE_NUMBER_2": null, "ROUTE_NUMBER_3": null, "SHIELD_TYPE": null, "LENGTH": 22.345, "GEOMETRY_UPDATE_DATETIME": "2013-03-21T10:24:22Z", "EFFECTIVE_DATETIME": "2021-07-28T13:52:11Z", "SYSTEM_DATETIME": "2021-07-29T16:34:25Z", "OBJECTID": 37249249, "SHAPELEN": 0 }, "geometry": { "type": "LineString", "coordinates": [[-79.085419936934414, 43.045433990919648], [-79.0851612368578, 43.045500790943301]] } }

Figure 3 - Algorithme pour créer les plus courts chemins à partir d'un point de départ

```
MATCH(source:Intersection {id: '4fa50f31-c5b0-4453-bdec-89abdbb22ec9'})
CALL gds.allShortestPaths.dijkstra.stream('myGraph', {
    sourceNode: source,
    relationshipWeightProperty: 'length'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
MATCH ((source) <- [:is_closest_to] - (sourceRestau:Restaurant))
UNWIND nodeIds AS node
MERGE (i:Intersection {id: gds.util.asNode(node).id})
MERGE (i)<-[:is_closest_to]-(r:Restaurant)
RETURN
    index,
    gds.util.asNode(sourceNode).id AS sourceNodeName, sourceRestau.name AS source_resto,
    gds.util.asNode(targetNode).id AS targetNodeName, [res IN COLLECT(r) | res.name] AS
restaurants,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).id] AS nodeNames,
    costs,
    nodes(path) as path
ORDER BY index
```