Université de Lorraine
TELECOM Nancy

Bases de Données NoSQL

Analyse de Données Protéiques

Architecture NoSQL : MongoDB & Neo4j

Réalisé par :

Maxence AGRA
Lucine GIRAUD
Lina LEKBOURI

17 décembre 2025

Table des matières

1 Introduction

L'augmentation exponentielle des données génomiques nécessite des architectures de stockage et d'analyse capables de gérer cette augmentation de données. Ce projet propose une approche combinant un stockage documentaire pour les données brutes avec MongoDB et une base de graphes pour l'analyse relationnelle avec Neo4J.

Les objectifs principaux sont :

1. stocker efficacement les données UniProt (MongoDB),
2. modéliser les similarités entre protéines (Neo4j),
3. annoter les fonctions protéiques via propagation de labels.

2 Données utilisées

Les fichiers `.tsv` utilisés proviennet de la base de données UniProt et contiennent les informations de protéines de souris et d'humains. Voici les colonnes utilisées :

1. **Entry** : ID UniProt principal (clé unique), servira d'ID dans MongoDB et d'ID de nœud dans Neo4j
2. **Entry Name** : nom de la protéine, utile pour l'affichage et la recherche textuelle
3. **Sequence** : séquence d'acides aminés (AA)
4. **InterPro** : liste de domaines InterPro de la protéine, colonne clé pour construire le graphe de similarité
5. **EC number** : numéro(s) EC, correspond aux fonctions enzymatiques de la protéine, ce sont les labels pour la tâche d'annotation
6. **Protein names** : nom(s) "humains" de la protéine, utile pour la recherche textuelle
7. **Organism** : organisme de la protéine (humain ou souris)

3 Architecture et Infrastructure

L'environnement est entièrement conteneurisé via Docker, garantissant la reproductibilité et l'isolation des services.

Il suffit de télécharger le dossier du code et d'exécuter la commande `docker compose up - -build -d` afin de lancer les 3 services. Le lancement des conteneurs Docker prend plusieurs minutes. On peut ensuite accéder à l'application sur ce lien : `http://127.0.0.1:5000`.

L'architecture repose sur trois conteneurs interconnectés :

— **mongo**, pour la base de données documentaire,
— **neo4j** pour la base de données graphe avec les plugins *APOC* et *Graph Data Science (GDS)* nécessaires aux algorithmes de similarité,
— **app** pour l'application qui orchestre l'ingestion des données via les scripts `load_mongo.py` et `build_graph.py`, et expose l'API de requête.

L'application utilise **Flask** pour l'API, **Bootstrap** pour le style, **JavaScript** pour la logique et **Cytoscape** pour la visualisation du graphe.

4 Tâche 1 : Création de MongoDB

Chaque protéine est transformée en un document JSON. L'identifiant UniProt (`Entry`) est utilisé comme clé primaire pour faciliter la cohérence avec Neo4j. Le schéma complet est disponible en **Annexe A.1**.

L'importation pour MongoDB est réalisée dans le fichier `load_mongo.py` et est optimisée par l'utilisation de batchs de 5000 documents.

5 Tâche 2 : Construction du graphe Neo4J

La modélisation en graphe permet d'expliciter les relations de similarité structurelle entre les protéines, basées sur leurs domaines fonctionnels (InterPro). Les données intégrées à Neo4j viennent directement de MongoDB.

5.1 Schéma du graphe

Le graphe est initialement biparti :

— **Nœuds** : `(:Protein)` et `(:Domain)`.

— **Relation** : `(:Protein)-[:HAS_DOMAIN]->(:Domain)`.

Les noeuds *protéines* comprennent les attributs suivants : *uniprot_id, entry_name, organism, length, ec_numbers, is_labelled, interpro_ids*. Les attributs ne sont pas clés ici comme le but est de visualiser les relations entre les protéines et les domaines (relations *HAS_DOMAIN*), ainsi que la similarité entre protéines (relations *SIMILAR*).

Les relations *HAS_DOMAIN* se créent si une protéine est présente dans un domaine. La partie suivante explique la création des relations *SIMILAR*.

5.2 Création des relations de similarité entre protéines

En premier, une projection du graphe est réalisée à l'aide de la librairie GDS. Puis, l'indice de Jaccard est calculé pour pondérer les arêtes `[:SIMILAR_TO]` entre deux protéines.

L'algorithme écrit les relations des 10 indices de Jaccard les plus élevés au dessus d'un certain seuil (par défaut 0,1) pour chaque protéine. Les paramètres du nombre de relations retenues et du seuil sont modifiables. Cette écriture est aussi réalisée à l'aide de GDS.

En dernier, l'algorithme ajoute deux attributs à chaque relation *SIMILAR* : *shared_domains*, domaines partagés entre ces deux protéines, et *union_domains*, nombre total de domaines de ces deux protéines. La démonstration mathématique permettant d'isoler le nombre de domaines partagés à partir de l'indice de Jaccard est détaillée en **annexe A.2**.

Le calcul et l'écriture des deux attributs *shared_domains* et *union_domains* sont réalisés par deux requêtes identiques, une utilisant la parallélisation pour faire la majorité des relations et l'autre sans parallélisation pour corriger les erreurs inévitables liées à des threads bloqués (deadlocks) sur la première.

Toutes ces étapes sont gérées dans le script `build_graph.py`. Maintenant que les deux bases de données sont créées, l'étape suivante est de créer les requêtes nécessaires pour interroger ces bases de données.

6 Tâche 3 : Requêtes et API

L'accès aux données est unifié via une API Python qui interroge la base la plus appropriée selon la requête. La classe `MongoProteinQueryManager` dans le fichier *mongo_ queries* gère les recherches basées sur le contenu (recherche par ID, nom de gène, nom de protéine, filtrage par `ec_number` ou domaine, agrégations pour les statistiques). La classe `Neo4jProteinQueryManager` dans le fichier *neo4j_ queries* permet la récupération des protéines voisines, le calcul des statistiques sur la similarité ainsi que l'identification des communautés de protéines.

L'application permet la recherche des protéines de différentes manières comme dit précédemment ainsi que la visualisation des protéines et de ses voisins (profondeur 1 ou 2) (voir **annexe A.3**). Les statistiques sont disponibles sur l'onglet statistique (voir **annexe A.4**).

7 Tâche 4 : Identification des communautés et annotation fonctionnelle

L'objectif final est de prédire la fonction (`ec_number`) des protéines non annotées.

7.1 Algorithme : Label Propagation (LPA)

Nous utilisons l'algorithme de **propagation d'étiquettes (LPA)** via la librairie GDS. C'est une méthode rapide qui détecte les communautés en se basant uniquement sur la structure du réseau, sans nécessiter de fonction objective prédéfinie.

Le principe de fonctionnement comporte quatre étapes :

1. **l'initialisation** où chaque nœud (protéine) reçoit une étiquette unique (identifiant de communauté),
2. **la propagation** où à chaque itération, un nœud met à jour son étiquette pour adopter celle qui est majoritaire parmi ses voisins,
3. **la pondération** où l'influence des voisins est modulée par le poids des relations (`jaccard_weight`), un nœud fortement connecté aura plus d'influence,
4. **la convergence** où l'algorithme s'arrête à un consensus local ou après un nombre défini d'itérations.

7.2 Implémentation et Prédiction

L'exécution se déroule en deux phases techniques.

La première phase est la phase de détection des communautés. Une projection de graphe est créée en mémoire incluant les propriétés de poids puis la procédure `gds.labelPropagation.write` écrit l'attribut `community_id` sur chaque nœud à l'aide de la technique expliquée précédemment. La visualisation de cette phase est l'étape 1 de la page "Labellisation" de l'application (voir **annexe A.5**).

La deuxième phase est la phase d'annotation. Pour les communautés mixtes (contenant à la fois des protéines annotées et non-annotées), une prédiction est effectuée selon l'une des deux stratégies suivantes :

1. le **vote majoritaire** par attribution du numéro EC le plus fréquent dans la communauté, ce n'est donc plus problème de multi-labelling,
2. l'**union** des fonctions, après union, un **seuil** est utilisé pour ne garder que les numéros EC les plus fréquents dans la communauté (les 30% les plus fréquents).

Le choix est réalisable dans la deuxième étape de la page "Labellisation" (voir **annexe A.6**). Suite à cette labellisation, la page "Statistiques" est mis à jour (voir **annexe A.4**).

Cette approche permet de propager les annotations fonctionnelles des protéines connues vers les protéines inconnues ayant une forte similarité structurelle.

8 Conclusion

Au terme de ce projet, l'utilisation combinée de Neo4J et de MongoDB avec la propagation de labels a permis de doubler le nombre de protéines annotées (passant de 13 % à 25 %). Au-delà de ces résultats chiffrés, l'architecture conteneurisée permet l'installation de l'application en quelques minutes. Cette base pourrait désormais être enrichie, par exemple en y intégrant des méthodes de Machine Learning ou de nouvelles variables biologiques sur les fonctions de protéines, pour affiner encore davantage les prédictions.

A Annexes

A.1 Modèle de données : document JSON complet

Voici la structure complète d'un document protéine tel qu'inséré dans MongoDB.

```
{
  "_id": "A0A024QYR9",
  "entry_name": "A0A024QYR9_MOUSE",
  "organism": "Mus musculus",
  "protein_names": [
    "Phosphatidylinositol 3-kinase adapter protein",
    "B-cell adapter for PI3K",
    "BCAP"
  ],
  "gene_names": ["Pik3ap1"],
  "sequence": {
    "length": 572,
    "aa": "MERGGEAAAAAAAVGGGGGEVAS..."
  },
  "interpro_ids": [
    "IPR035892",
    "IPR000456"
  ],
  "ec_numbers": [
    "3.1.3.16",
    "2.7.11.1"
  ],
  "is_labelled": true,
  "created_at": "2023-10-27T10:00:00Z"
}
```

Listing 1 – Document JSON complet d'une protéine

A.2 Détails mathématiques : calcul des domaines partagés

Pour obtenir le nombre de domaines partagés (*shared_ domains*) entre deux protéines u et v à partir de l'indice de Jaccard et de la taille de leurs domaines respectifs, nous utilisons la démonstration suivante.

Nous partons de la définition du coefficient de Jaccard :

$$J(u,v) = \frac{|D(u) \cap D(v)|}{|D(u) \cup D(v)|}$$

D'après le principe d'inclusion-exclusion, nous exprimons l'union en fonction de l'intersection :

$$|D(u) \cup D(v)| = |D(u)| + |D(v)| - |D(u) \cap D(v)|$$

En substituant cette expression dans la définition du Jaccard, nous obtenons :

$$J(u,v) = \frac{|D(u) \cap D(v)|}{|D(u)| + |D(v)| - |D(u) \cap D(v)|}$$

Et après isolation du terme de l'intersection $|D(u) \cap D(v)|$:

$$|D(u) \cap D(v)| = \frac{J(u,v) \cdot (|D(u)| + |D(v)|)}{1 + J(u,v)} \tag{1}$$

Cette formule permet de calculer l'attribut `shared_domains` sans avoir à recalculer explicitement l'intersection des listes de domaines dans la base de données.

A.3 Interface utilisateur : accueil, recherche et visualisation protéine

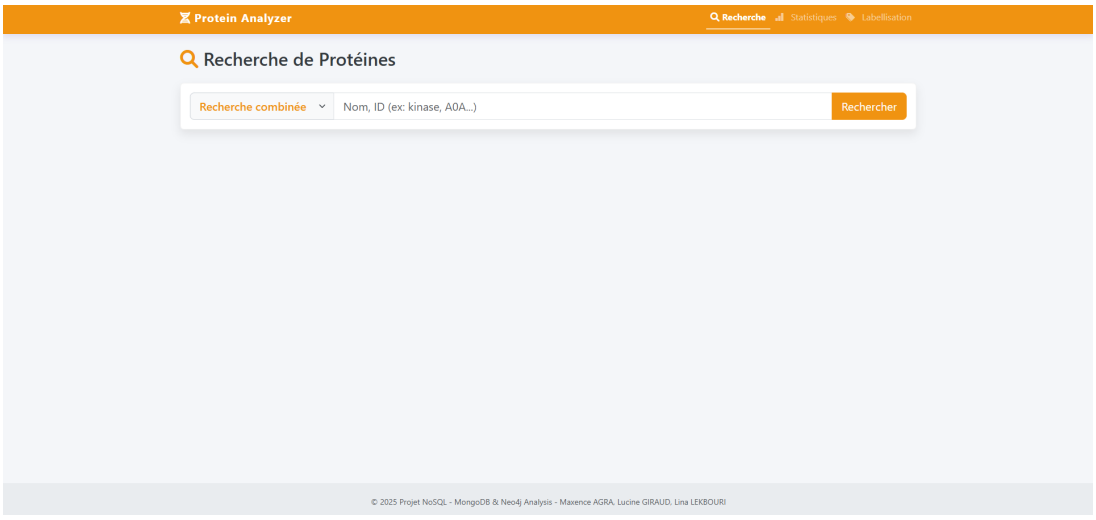

FIGURE 1 – Capture d'écran de la page d'accueil de l'application permettant la recherche de protéines.

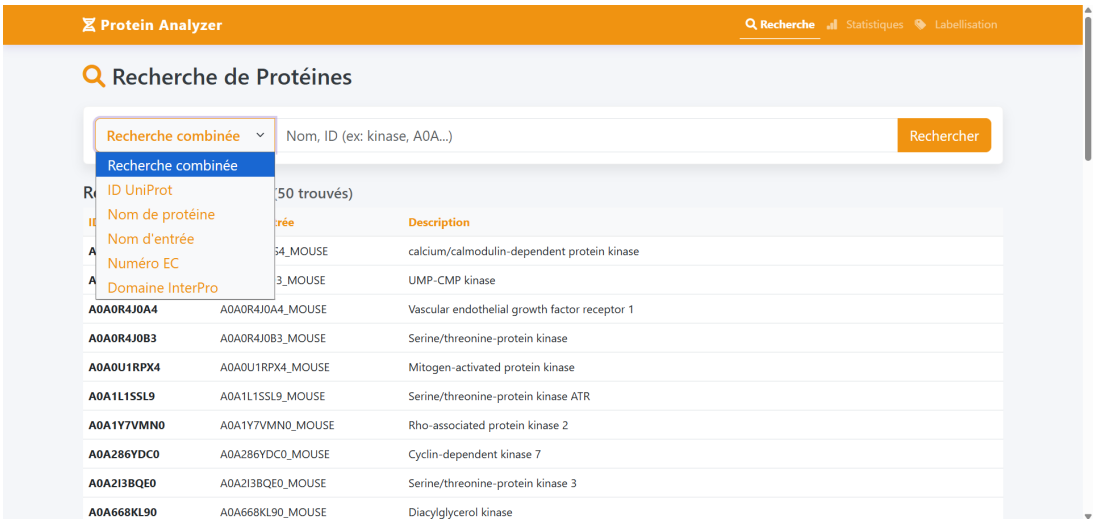

FIGURE 2 – Capture d'écran de la page après recherche.

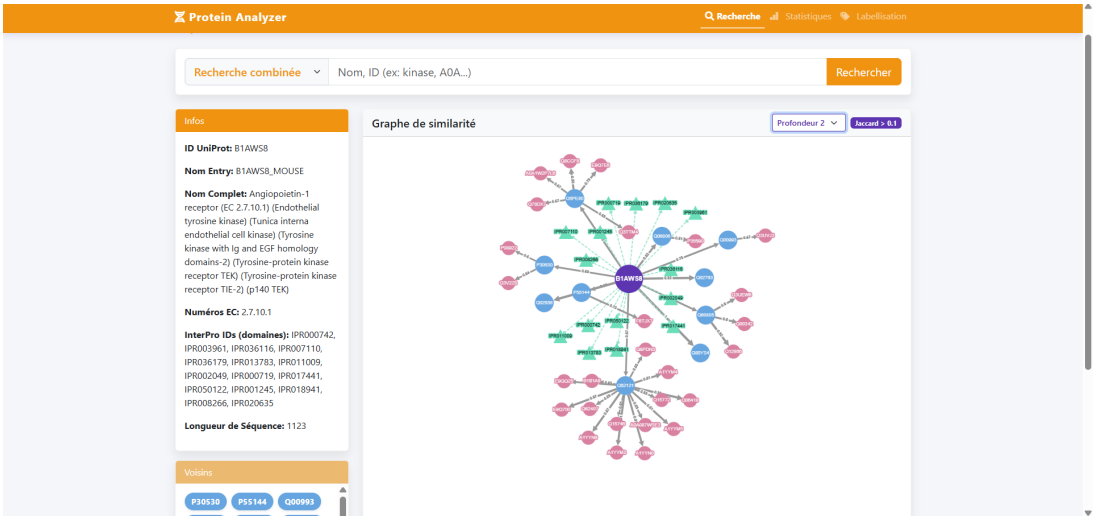

FIGURE 3 – Capture d'écran de la visualisation d'une protéine.

A.4 Interface utilisateur : statistiques

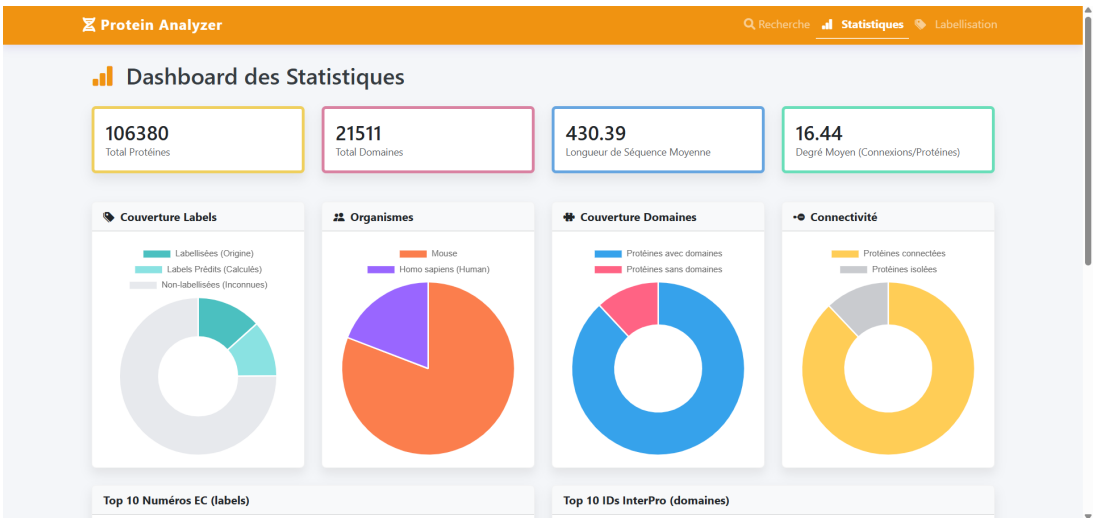

FIGURE 4 – Tableau de bord des statistiques généré à partir des agrégations MongoDB et Neo4j.

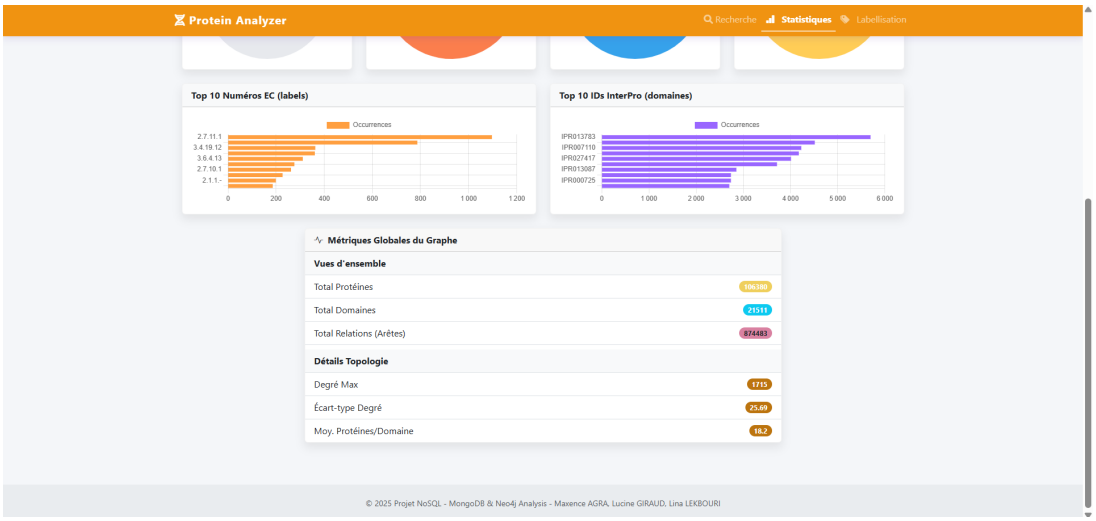

FIGURE 5 – Tableau de bord des statistiques généré à partir des agrégations MongoDB et Neo4j.

A.5 Interface utilisateur : Labellisation - Étape 1

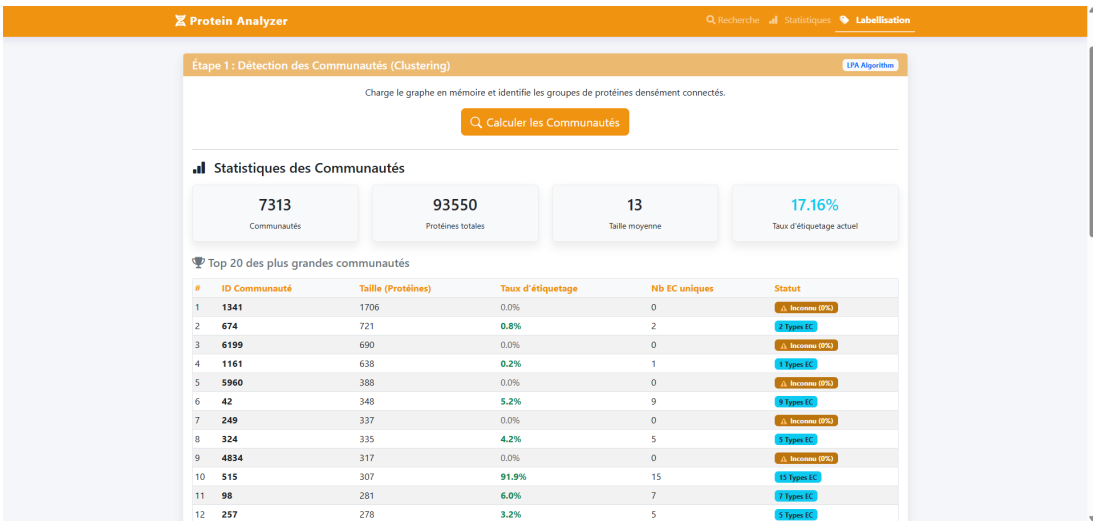

FIGURE 6 – Étape 1 de la labellisation.

A.6 Interface utilisateur : Labellisation - Étape 2

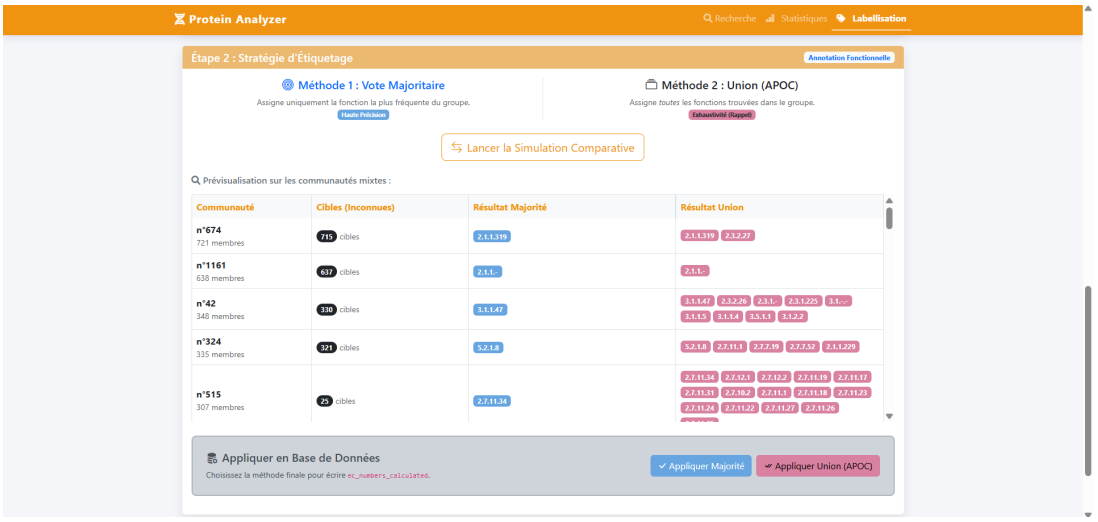

FIGURE 7 – Étape 2 de la labellisation.