

# Projet NoSQL

Système de base de données polyglotte

Master 2 MIAGE – Informatique pour la Finance

Université Paris Dauphine

Hakim IGUENI, Elias KESRAOUI

Juin 2025

## 1 Introduction

Dans un contexte où les systèmes d'information doivent gérer des données de plus en plus hétérogènes, volumineuses et complexes, les bases de données polyglottes s'imposent comme une solution pertinente. Elles consistent à combiner plusieurs technologies de stockage, chacune étant optimisée pour un type de données ou de requêtes spécifique, afin de former un système globalement plus performant et plus souple.

Dans le cadre de ce projet, nous avons choisi de concevoir, modéliser et interroger une **base de données polyglotte** destinée à un système de gestion touristique. L'objectif est de démontrer comment des technologies complémentaires peuvent être articulées pour répondre à des besoins variés tels que la gestion de points d'intérêt, d'avis, de commentaires riches et de données géographiques.

Pour cela, nous avons mobilisé trois technologies principales :

- **PostgreSQL** pour les données relationnelles structurées telles que les hébergements, les utilisateurs ou les avis.
- **MongoDB** pour le stockage des commentaires et des réponses imbriquées, sous forme de documents semi-structurés.
- **Neo4j** pour la représentation des villes et de leurs connexions, en exploitant les capacités de traversée de graphe.

Chaque technologie a été sélectionnée pour ses avantages spécifiques, et nous avons cherché à illustrer leur complémentarité dans un système cohérent. En parallèle, nous avons également mis en place un script **Python** afin de tester certaines interactions entre les bases, notamment l'ajout d'une ville ou d'un avis de manière simultanée. Ce volet applicatif reste optionnel, mais il démontre la faisabilité d'une intégration réelle.

Ce rapport a ainsi pour vocation de présenter l'ensemble de notre démarche : de la modélisation à la mise en œuvre technique, en passant par l'insertion des données, la réalisation de requêtes, et l'intégration partielle entre les composants. Il illustre, à travers un cas d'usage concret, les bénéfices d'une architecture polyglotte dans la conception de systèmes de données modernes.

## 2 Architecture du système

### 2.1 Présentation générale

Dans le cadre de ce projet, nous avons choisi de concevoir un système d'information touristique selon une **architecture polyglotte**, c'est-à-dire reposant sur l'utilisation conjointe de plusieurs technologies de bases de données, chacune étant spécialisée dans un type de données spécifique.

Cette approche nous a permis de séparer clairement les responsabilités des différentes couches du système tout en tirant parti des avantages spécifiques offerts par chaque technologie. Notre objectif était ainsi d'assurer à la fois performance, flexibilité, et cohérence dans la gestion et la consultation des données.

### 2.2 Rôle de chaque technologie

**PostgreSQL :** Nous avons utilisé PostgreSQL pour gérer les entités fortement structurées qui présentent des relations claires et des contraintes d'intégrité :

- Points d'intérêt
- Hébergements
- Activités
- Villes
- Utilisateurs
- Avis

Ce système relationnel garantit la cohérence des données grâce à ses mécanismes de clés étrangères, de transactions, et de déclencheurs.

**MongoDB :** MongoDB a été utilisé pour stocker les commentaires associés aux avis. Ces données, moins structurées et susceptibles de varier d'un enregistrement à l'autre (commentaires imbriqués, réponses optionnelles, etc.), s'intègrent naturellement dans le modèle document de MongoDB.

**Neo4j :** Nous avons choisi Neo4j pour représenter le graphe des villes reliées par des distances et des durées de trajet. Ce modèle permet d'exprimer efficacement des requêtes de type « trouver les villes voisines », « chemin le plus court », ou « villes isolées », qui seraient complexes dans un modèle relationnel classique.

### 2.3 Résumé des interactions entre les bases

Afin de démontrer la faisabilité d'un système polyglotte cohérent, nous avons également testé certaines interactions entre les bases :

- Lors de l'insertion d'un **commentaire** via Python, celui-ci est ajouté dans MongoDB, puis un avis référencé est inséré dans PostgreSQL.
- L'ajout d'une **ville** via la couche Python permet de la créer à la fois dans PostgreSQL (structure) et dans Neo4j (topologie).

Ces cas d'usage montrent la cohérence du découpage choisi : chaque technologie est exploitée pour ce qu'elle fait de mieux, tout en permettant des croisements via des identifiants communs (`ville_id`, `avis_id`).

## 3 Modèles de données et commandes de manipulation

Dans cette section, nous présentons les différents modèles de données mis en œuvre dans notre projet polyglotte, ainsi que les principales commandes de création, suppression et mise à jour associées à chaque technologie.

### 3.1 PostgreSQL – Modèle relationnel

Le modèle relationnel gère les entités fortement structurées du système. Il repose sur les tables suivantes :

- **Ville** : identifiant, nom unique, latitude, longitude.
- **Hebergement** : nom, type, prix, géolocalisation, note, rattaché à une ville.
- **PointInteret** : nom, description courte, géolocalisation, rattaché à une ville.
- **Activite** : nom, description, prix, durée, dates, rattachée à un POI.
- **Avis** : note sur 5, utilisateur, date, identifiant du commentaire MongoDB, rattaché à un hébergement.

#### Création des tables

Extrait du script `create.sql` :

```
CREATE TABLE Ville (  
    id SERIAL PRIMARY KEY,  
    nom TEXT NOT NULL UNIQUE,  
    latitude FLOAT,  
    longitude FLOAT  
);  
  
CREATE TABLE Hebergement (  
    id SERIAL PRIMARY KEY,  
    nom TEXT NOT NULL,  
    ...  
    ville_id INT REFERENCES Ville(id) ON DELETE CASCADE  
);
```

#### Insertion de données

Extrait du script `insert.sql` :

```
INSERT INTO Ville (nom, latitude, longitude) VALUES ('Montreal', 45.5017, -73.5673);  
INSERT INTO Hebergement (nom, type, prix, ville_id) VALUES ('Hotel Bleu', 'Hôtel', 120, 1);
```

#### Suppression et mise à jour

Suppression d'un hébergement (avec cascade sur les avis):

```
DELETE FROM Hebergement WHERE id = 1;
```

Mise à jour du prix :

```
UPDATE Hebergement SET prix = 135 WHERE id = 1;
```

## MongoDB – Modèle document

Nous avons utilisé MongoDB pour stocker les `commentaires`, potentiellement imbriqués, associés à un identifiant d'avis PostgreSQL.

**Création de la base et de la collection** Fichier `create-db.js` :

```
use projet_bdd
db.createCollection("commentaires")
```

**Insertion de documents** Fichier `insert_commentaires.js` :

```
db.commentaires.insertOne({
  avis_id: "1",
  contenu: "Très bon séjour.",
  reponses: [
    { utilisateur: "Admin", contenu: "Merci !" }
  ]
});
```

### Mise à jour et suppression

Exemple de mise à jour :

```
db.commentaires.updateOne(
  { avis_id: "1" },
  { $set: { contenu: "Excellent séjour." } }
);
```

Suppression :

```
db.commentaires.deleteOne({ avis_id: "1" });
```

## 3.2 Neo4j – Modèle graphe

La base Neo4j est utilisée pour représenter un réseau de villes canadiennes connectées par des relations orientées de type `RELIE_A`, enrichies de deux attributs : `distance` (en km) et `temps` (en minutes).

**Création des nœuds et relations** Les villes sont créées une seule fois grâce à l'opérateur `MERGE` :

```
MERGE (:Ville {nom: 'Montreal'});
MERGE (:Ville {nom: 'Quebec'});
MERGE (:Ville {nom: 'Ottawa'});
MERGE (:Ville {nom: 'Toronto'});
MERGE (:Ville {nom: 'Vancouver'});
MERGE (:Ville {nom: 'Halifax'});
MERGE (:Ville {nom: 'Longueuil'});
MERGE (:Ville {nom: 'Laval'});
MERGE (:Ville {nom: 'Gatineau'});
MERGE (:Ville {nom: 'Mississauga'});
MERGE (:Ville {nom: 'Burnaby'});
MERGE (:Ville {nom: 'Dartmouth'});
```

Création de relations avec distances et temps :

```
MATCH (m:Ville {nom: 'Montreal'}), (q:Ville {nom: 'Quebec'})
CREATE (m)-[:RELIE_A {distance: 250, temps: 180}]->(q);

MATCH (o:Ville {nom: 'Ottawa'}), (t:Ville {nom: 'Toronto'})
CREATE (o)-[:RELIE_A {distance: 450, temps: 270}]->(t);

MATCH (v:Ville {nom: 'Vancouver'}), (h:Ville {nom: 'Halifax'})
CREATE (v)-[:RELIE_A {distance: 6030, temps: 4200}]->(h);
```

### Mise à jour d'une relation

Exemple : modifier le trajet entre Ottawa et Toronto :

```
MATCH (v1:Ville {nom: 'Ottawa'})-[r:RELIE_A]-(v2:Ville {nom: 'Toronto'})
SET r.distance = 430, r.temps = 260
```

### Suppression d'une ville et de ses connexions

```
MATCH (v:Ville {nom: 'Halifax'}) DETACH DELETE v
```

### Autres requêtes pertinentes

Trouver les villes proches de Montreal (moins de 10 km) :

```
MATCH (v1:Ville {nom: 'Montreal'})-[r:RELIE_A]-(v2:Ville)
WHERE r.distance < 10
RETURN v2.nom AS ville_proche, r.distance
```

Trouver l'itinéraire optimal entre Ottawa et Vancouver :

```
MATCH path = (start:Ville {nom: 'Ottawa'})-[:RELIE_A*]-(end:Ville {nom: 'Vancouver'})
RETURN [v IN nodes(path) | v.nom] AS villes,
       reduce(total = 0, rel IN relationships(path) | total + rel.distance) AS distance_totale
ORDER BY distance_totale ASC
LIMIT 1
```

## 4 Choix de modélisation : avantages et limites

Dans le cadre de notre projet de base de données polyglotte, nous avons effectué des choix de modélisation adaptés à la nature des données à manipuler et aux types de requêtes envisagées. Voici un retour détaillé sur ces choix, ainsi que les avantages et les limites associés à chaque technologie.

### 4.1 PostgreSQL – Données relationnelles structurées

**Choix** Nous avons utilisé PostgreSQL pour modéliser les entités structurées telles que les **Villes**, **Hébergements**, **Points d'intérêt**, **Activités** et **Avis**. Ces entités présentent des relations bien définies (clés étrangères) et sont soumises à des contraintes d'intégrité fortes.

#### Avantages

- Forte cohérence grâce aux contraintes d'intégrité référentielle.
- Requêtes SQL expressives et optimisées pour les jointures.
- Transactions ACID garantissant la fiabilité des opérations.

### Limites

- Peu adapté aux structures de données flexibles ou imbriquées (ex. : commentaires/réponses).
- Les relations complexes (ex. graphe) sont coûteuses à exprimer en SQL.

## 4.2 MongoDB – Données semi-structurées

**Choix** Nous avons utilisé MongoDB pour stocker les **commentaires** et leurs **réponses imbriquées**, car ces données varient d’un avis à l’autre en structure, profondeur ou nombre de réponses.

### Avantages

- Grande flexibilité du modèle document (pas besoin de schéma rigide).
- Représentation naturelle des structures hiérarchiques.
- Évolutivité horizontale et insertion rapide des documents.

### Limites

- Pas de contraintes relationnelles fortes (ex. : références aux identifiants d’avis non vérifiées).
- Requêtes complexes (aggrégations, recherches profondes) parfois moins intuitives.
- Risque de duplication d’information dans les réponses imbriquées.

## 4.3 Neo4j – Données orientées graphe

**Choix** Neo4j a été utilisé pour modéliser les **villes** et leurs **relations de connectivité** (trajets, distances, durées), permettant des calculs d’itinéraires et de voisinages difficiles à exprimer en SQL.

### Avantages

- Modélisation naturelle des réseaux et des relations orientées.
- Performances excellentes sur les requêtes de parcours de graphe (traversée, chemin le plus court, etc.).
- Syntaxe Cypher claire pour décrire des patterns complexes.

### Limites

- Moins adapté pour la gestion de grandes quantités de données fortement structurées (par exemple : factures ou états financiers).
- Pas de support natif pour les transactions multi-sources (cross-DB).
- Nécessite une duplication partielle des données (les villes sont aussi présentes dans PostgreSQL).

## 4.4 Synthèse

Ces choix de modélisation reflètent une approche pragmatique : chaque technologie a été utilisée selon ses forces. La séparation par périmètre fonctionnel (relationnel, document, graphe) permet d'obtenir un système modulaire, performant et adapté à des usages réels.

Cependant, cette architecture polyglotte implique également une complexité accrue dans l'intégration des données et la cohérence entre bases, ce qui nécessiterait dans un système en production de mettre en place une couche applicative robuste pour orchestrer les synchronisations entre composants.

## 5 Requêtes et résultats

Notre système polyglotte permet de répondre efficacement à une diversité de requêtes, chacune exploitant les forces de la technologie sous-jacente (PostgreSQL, MongoDB, Neo4j). Ci-dessous, nous illustrons six requêtes clés représentatives des cas d'usage du projet.

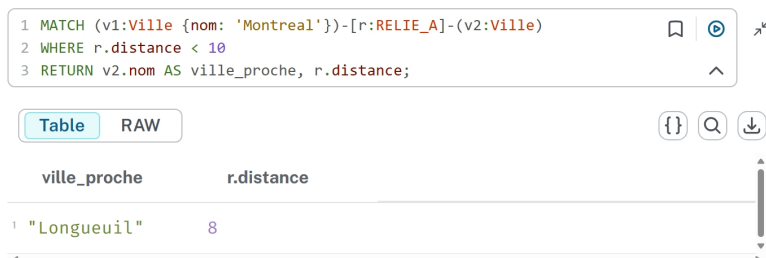
### 5.1 Villes situées à moins de 10 km d'une ville donnée (Neo4j)

Cette requête exploite la base de données graphe pour identifier les connexions de proximité :

**Requête :**

```
MATCH (v1:Ville {nom: 'Montreal'})-[r:RELIE_A]-(v2:Ville)
WHERE r.distance < 10
RETURN v2.nom AS ville_proche, r.distance;
```

**Résultat :**



The screenshot shows a Neo4j query editor with a Cypher query and its result in table format. The query is:

```
1 MATCH (v1:Ville {nom: 'Montreal'})-[r:RELIE_A]-(v2:Ville)
2 WHERE r.distance < 10
3 RETURN v2.nom AS ville_proche, r.distance;
```

The result is displayed in a table with two columns: `ville_proche` and `r.distance`. The first row shows the value `"Longueuil"` for `ville_proche` and `8` for `r.distance`.

ville_proche	r.distance
"Longueuil"	8

### 5.2 Activités associées à un point d'intérêt donné (PostgreSQL)

Cette requête relationnelle permet de lister les activités liées à un point d'intérêt spécifique :

**Requête :**

```
SELECT
    a.*
FROM
    Activite a
    JOIN PointInteret p ON a.poi_id = p.id
WHERE
    p.nom = 'Tour CN';
```

Résultat :

id	nom	description	prix	duree
4	Ascension Tour CN	Vue panoramique	35.00	45
date_debut		date_fin	poi_id	
2025-02-01		2025-11-30	4	

(1 row)

### 5.3 Hébergements d'une ville donnée (PostgreSQL)

Cette requête récupère les hébergements liés à une ville spécifique :

Requête :

```
SELECT
  h.*
FROM
  Hebergement h
  JOIN Ville v ON h.ville_id = v.id
WHERE
  v.nom = 'Montreal';
```

Résultat :

id	nom	type	prix	latitude	longitude	note	ville_id
1	Hotel Belle Vue	Hotel	120.00	45.502	-73.5675	4.50	1
2	Auberge du Vieux-Port	Auberge	85.00	45.5041	-73.554	4.00	1

(2 rows)

### 5.4 Distance et temps de trajet entre deux villes (Neo4j)

Cette requête retourne les informations de déplacement direct entre deux villes :

Requête :

```
MATCH (v1:Ville {nom: 'Toronto'})-[r:RELIE_A]-(v2:Ville {nom: 'Vancouver'})
RETURN v1.nom AS ville_depart, v2.nom AS ville_arrivee, r.distance, r.temps;
```

Résultat :



The screenshot shows a Neo4j Cypher query editor with the following query:

```
1 MATCH (v1:Ville {nom: 'Toronto'})-[r:RELIE_A]-(v2:Ville {nom: 'Vancouver'})
2
3 RETURN v1.nom AS ville_depart, v2.nom AS ville_arrivee, r.distance,
4      r.temps;
```

Below the query editor, there is a table view of the results:

ville_depart	ville_arrivee	r.distance	r.temps
"Toronto"	"Vancouver"	3370	3000



## 5.5 Activités ayant lieu entre avril et juin (PostgreSQL)

Cette requête permet de filtrer les activités selon leurs dates :

Requête :

```
SELECT *
FROM Activite
WHERE
    EXTRACT(MONTH FROM date_debut) >= 4
    AND EXTRACT(MONTH FROM date_fin) <= 6;
```

Résultat :

id	nom	description	prix	duree
1	Visite guidée Vieux-Montreal	Découverte historique à pied	20.00	90

date_debut	date_fin	poi_id
2025-04-01	2025-05-27	1

(1 row)

## 5.6 Itinéraires possibles entre deux villes (Neo4j)

Nous listons les différentes étapes d'un itinéraire entre deux villes, en tenant compte de tous les chemins possibles :

Requête :

```
MATCH path = (start:Ville {nom: 'Montreal'})-[:RELIE_A*]-(end:Ville {nom: 'Vancouver'})
WHERE start <> end
RETURN
    [v IN nodes(path) | v.nom] AS villes_etapes,
    reduce(total = 0, rel IN relationships(path) | total + rel.distance) AS distance_totale,
    reduce(total = 0, rel IN relationships(path) | total + rel.temps) AS duree_totale
ORDER BY size(nodes(path)) ASC
```

Résultat :



The screenshot shows a Neo4j query editor with a Cypher query and its results in a table view. The query finds paths between Montreal and Vancouver, calculating total distance and duration. The results table shows two paths: one via Halifax (distance 7270, duration 5100) and one via Ottawa and Toronto (distance 4000, duration 3380).

villes_etapes	distance_totale	duree_totale
1 ["Montreal", "Halifax", "Vancouver"]	7270	5100
2 ["Montreal", "Ottawa", "Toronto", "Vancouver"]	4000	3380

Chaque requête démontre comment l'usage combiné de plusieurs bases permet d'exprimer efficacement des besoins variés dans un contexte touristique. Ces résultats pourraient, dans une version applicative, être intégrés dans une interface de consultation ou une API.

## 6 Conclusion

Ce projet de base de données polyglotte nous a permis d'aborder concrètement les enjeux liés à l'intégration de technologies hétérogènes au sein d'un même système d'information. Grâce à l'utilisation conjointe de PostgreSQL, MongoDB et Neo4j, nous avons pu construire un modèle de données cohérent, capable de répondre à des besoins variés de manière optimisée.

**Difficultés rencontrées :** Plusieurs défis ont jalonné notre progression :

- La conception d'un modèle global cohérent, tout en répartissant correctement les entités entre les trois technologies.
- L'adaptation des requêtes à chaque langage spécifique (SQL, Cypher, MongoDB queries), nécessitant une prise en main fine des syntaxes.
- L'absence d'un connecteur natif entre les bases, ce qui a nécessité l'usage de scripts Python intermédiaires pour simuler une cohérence entre les composants.
- Le traitement des relations orientées et pondérées dans Neo4j, notamment pour modéliser les villes et les trajets.

**Répartition du travail entre les membres du binôme :** Le projet a été réalisé en binôme, avec une répartition équilibrée des tâches :

- **Elias KESRAOUI** s'est principalement occupé de la modélisation SQL, des requêtes PostgreSQL, de l'écriture des scripts d'insertion, et de la rédaction du rapport.
- **Hakim IGUENI** a pris en charge la modélisation des graphes avec Neo4j, la conception des requêtes Cypher, la manipulation de MongoDB et la coordination des scripts d'interaction entre les bases.

Toutes les décisions ont été prises en concertation, avec un effort constant de validation croisée et de tests partagés.

**Temps consacré au projet :** Nous avons estimé avoir consacré environ **35 à 40 heures chacun**, réparties entre :

- Analyse du sujet et modélisation : 8h
- Implémentation des scripts SQL, Cypher, MongoDB : 15h
- Tests, corrections, et requêtes : 10h
- Rédaction du rapport et documentation : 7h

Ce projet fut riche d'enseignements et a renforcé nos compétences tant techniques que méthodologiques, notamment en matière d'architecture distribuée, de modélisation de données avancée, et de structuration documentaire.