Table of Contents

I. Introduction *2*

II. Présentation générale des choix de modélisation *3*

III. Description des différents modèles de données et exemples de commandes *4*

A. MongoDB (Info-API) **4**

1. Description des modèles – référence, emballage, trigger 4
2. Exemples de commandes 6

B. PostgreSQL (trip-API) **7**

1. Description des modèles 7
2. Exemples de commandes 8

C. Neo4j (route-API) **8**

1. Description des modèles 8
2. Exemples des commandes 8

IV. Requêtes et tests *9*

A. MongoDB **9**

B. PostgreSQL **12**

C. Neo4j **12**

V. Conclusion *14*

VI. Annexes – Examples des scripts *15*

Validators (MongoDB) **15**

Trigger (MongoDB) **16**

Insertion (MongoDB) **16**

Création et trigger (PostgreSQL) **17**

Insertion (PostgreSQL) **18**

Contraintes et index (Neo4j) **19**

Création des nœuds City (Neo4j) **19**

Création dynamique des relations LOCATED_AT (Neo4j) **20**

I. Introduction

Ce rapport présente la **conception** et la **mise en œuvre** des bases de données supportant l'architecture de micro-services de l'application intégrant trois systèmes de gestion de bases de données — **MongoDB**, **PostgreSQL** et **Neo4j** — afin de répondre aux exigences spécifiques de chaque service :

- **info-api (MongoDB)** : gestion des données de contenu (villes, points d'intérêt, activités, hébergements). L'utilisation de **triggers**, **contraintes** et **validators** garantit la cohérence, la validité des données et l'adhésion aux exigences de l'énoncé NoSQL, tout en simplifiant l'interaction avec les autres services.
- **trip-api (PostgreSQL)** : orchestration des itinéraires journaliers via les tables `trip`, `trip_activity` et `trip_accommodation`, assurant ainsi la fiabilité transactionnelle.
- **route-api (Neo4j)** : modélisation d'un graphe de villes, avec calcul automatique des distances et temps de trajet grâce à la fonction native `point.distance()`, optimisant ainsi les requêtes de chemin.

Pour faciliter nos développements, nous avons utilisé les interfaces graphiques **MongoDB Compass**, **PgAdmin** et **Neo4j Desktop**. Les bases PostgreSQL et Neo4j sont déployées localement, tandis que MongoDB est hébergée sur **MongoDB Atlas**, ce qui nous a permis d'utiliser les triggers dans un environnement cloud.

Le rapport est structuré de la manière suivante :

II. Présentation générale des choix de modélisation

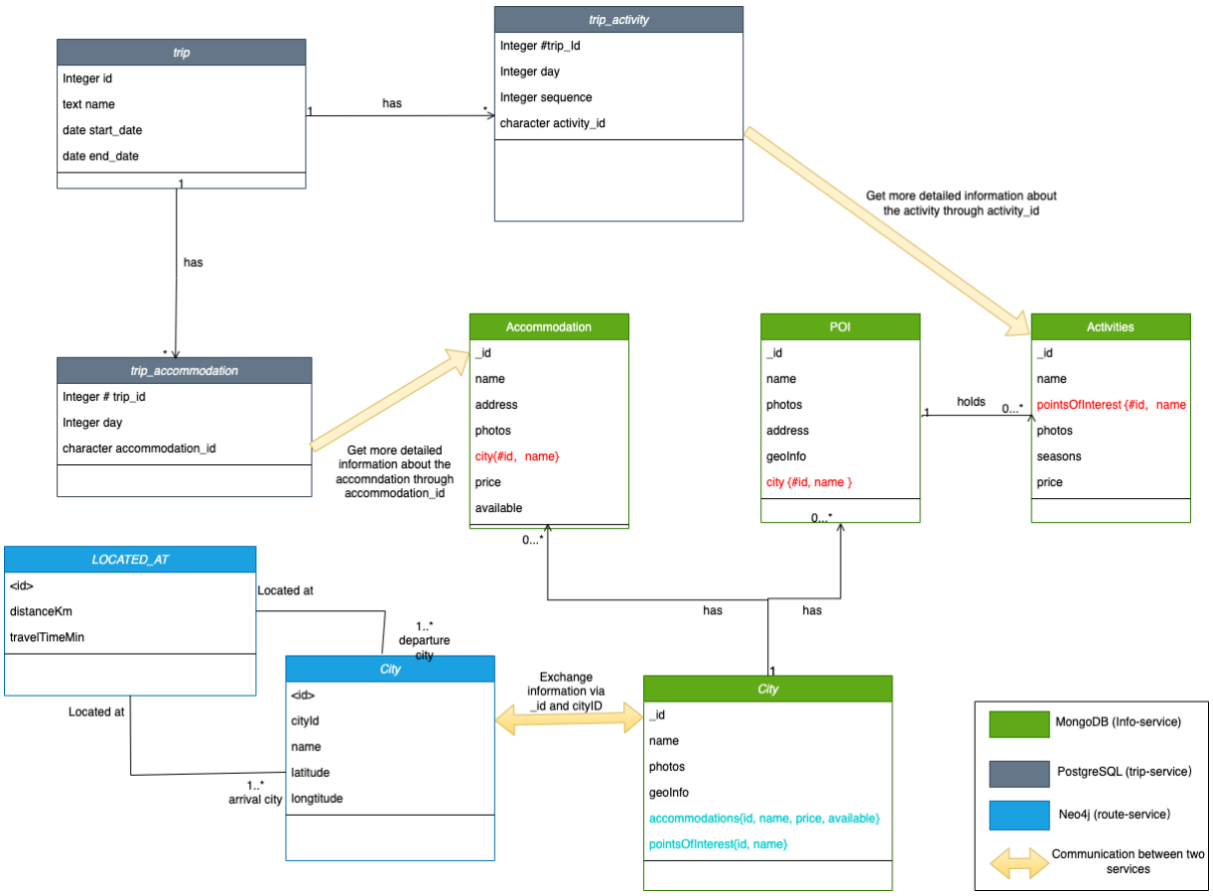
III. Description détaillée des modèles de données, accompagnée d'exemples de commandes pour chaque technologie

IV. Requêtes et tests — se focalisant sur les problématiques que chaque base de données est censée résoudre

V. Conclusion — bilan des tâches accomplies, répartition des responsabilités et principales difficultés rencontrées

VI. Annexes — compilation des scripts complets mentionnés dans le chapitre III

II. Présentation générale des choix de modélisation



Basé sur l'architecture micro-services, notre back-end utilise trois types de bases de données, chacun étant dédié à un service spécifique :

Micro-service	Persistance	Rôle principal
info-api	MongoDB	Opérations CRUD + recherche sur les ressources de contenu : villes, points d'intérêt, activités, hébergements
trip-api	PostgreSQL	Création et gestion d'itinéraires de voyage jour par jour
route-api	Neo4j	Gestion du graphe des villes : calcul des distances, durées de trajet, recherche d'itinéraires

Raisons du choix des bases de données

1. **info-api → MongoDB** *(bloc vert sur l'image)*
   - Le modèle documentaire correspond naturellement à des données hiérarchiques ou semi-structurées comme « ville / point d'intérêt / activité ».
2. **trip-api → PostgreSQL** *(bloc gris sur l'image)*

- Les itinéraires nécessitent de fortes contraintes relationnelles et transactionnelles (utilisateur, itinéraire → jour → étape) et donc une garantie ACID.

3. **route-api → Neo4j** *(bloc bleu sur l'image)*
   - Le réseau routier est intrinsèquement une structure de graphe : nœuds (POI/villes) + arêtes (routes / vols).
   - Neo4j offre des bibliothèques d'algorithmes dédiées aux plus courts chemins, chemins optimisés et pondérations personnalisées (distance, temps, coût).

Communication entre services

- Chaque service stocke uniquement les informations nécessaires.
- Les identifiants (IDs) servent de liens : un service peut récupérer les détails d'un objet stocké par un autre service grâce à un ID unique (voir flèches jaunes sur l'image)

Par exemple, PostgreSQL stocke seulement l'ID d'un **accommodation** et d'une **activity**. Lors de l'exécution de trip-api, notre application Java Web envoie un URL contenant ces IDs vers info-api afin d'obtenir les détails complets pour l'utilisateur.

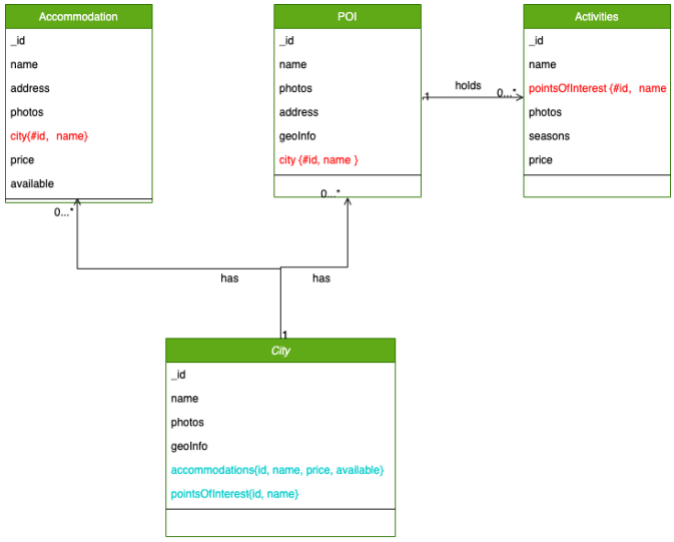
Limites

- Quand une opération implique plusieurs services (trip-api et info-api), maintenir la cohérence requiert une gestion de compensation, ce qui alourdit l'architecture.
- Les appels inter-services (trip-api vers info-api) ajoutent de la latence, pouvant impacter les performances en cas de forte charge.

III. Description des différents modèles de données et exemples de commandes

A. MongoDB (Info-API)

1. Description des modèles – référence, emballage, trigger



Conception initiale de MongoDB

- **Modèle normalisé + références unidirectionnelles**
  Chaque document ne contient qu'un seul champ de référence ("clé étrangère") :
  - PointOfInterest.cityId
  - Activity.pointOfInterestId
  - Accommodation.cityId

Avantage : facile à maintenir, pas de duplication, cohérence simplifiée.
Inconvénient : les requêtes nécessitent $lookup ou plusieurs requêtes, ce qui est coûteux en lecture.

Optimisation par embedding

Pour les relations **frequemment lues mais rarement mises à jour**, nous avons ajouté un embedding partiel :

- Dans City : inclusion de { id, name } pour ses PointOfInterest et { id, name,price,available} pour ces Accommodations.
- Dans Activity : inclusion du pointOfInterest.name.
- Dans POI et Accommodation : inclusion du city.name.

Cela permet d'afficher directement sur l'interface : city + POI/Accommodation, sans faire $lookup.

Limites de l'embedding

Chaque modification d'un champ partagé exige une mise à jour dans tous les documents concernés, par exemple :

- Si poi.name change → on doit mettre à jour city.pois et activity.poi.
- Si city.name change → on doit mettre à jour toutes les références embedées (poi, accommodation).

Cette duplication complique la cohérence des données et introduit des coûts de mise à jour plus élevés.

Solutions pour garantir la cohérence des données

a. Côté application

- Implémentation de la synchronisation dans Spring Boot lors des opérations de CRUD.
- Inconvénients : duplication de logique, risque d'oublis.

b. Transactions multi-documents (Replica Set + transactions)

- Nécessite un déploiement en Replica Set sur MongoDB.
- Avantages : atomicité (toutes les modifications réussissent ou sont annulées).
- Inconvénients : configuration plus complexe.

c. Triggers sur la base (MongoDB Atlas App Service)

- Déclenchés automatiquement sur chaque opération CRUD.
- Permettent une synchronisation centralisée dans la base, sans code applicatif supplémentaire.

Nous avons implémenté six triggers :

1. **SyncAccommodationToCity** : synchronisation entre accommodation et city.accommodations.
2. **SyncPOIToCity** : synchronisation entre pointOfInterest et city.pointsOfInterest.
3. **SyncCityNameToChildren** : propagation de city.name vers les documents poi et accommodation.
4. **SyncPOIToActivity** : synchronisation de pointOfInterest.name et suppression éventuelle dans activity.pointOfInterest.
5. **cascadeDeleteCityChildren** : suppression en cascade des poi et accommodation lors de la suppression d'une ville.
6. **cascadeDeletePOIChildren** : suppression en cascade des activity lors de la suppression d'un POI.

Avec ces triggers, la cohérence des données est entièrement gérée côté base de données via MongoDB Atlas. Il n'est donc pas nécessaire de maintenir un code de synchronisation dans l'application ni de recourir à des transactions multi-documents.

2. Exemples de commandes

Le code complet se trouve dans le dossier **BDD_scripts_NGUYEN_YANG/MongoDB**. Ci-dessous, quelques exemples simplifiés.

Nous avons défini la séquence suivante pour créer la base de données et insérer des données :

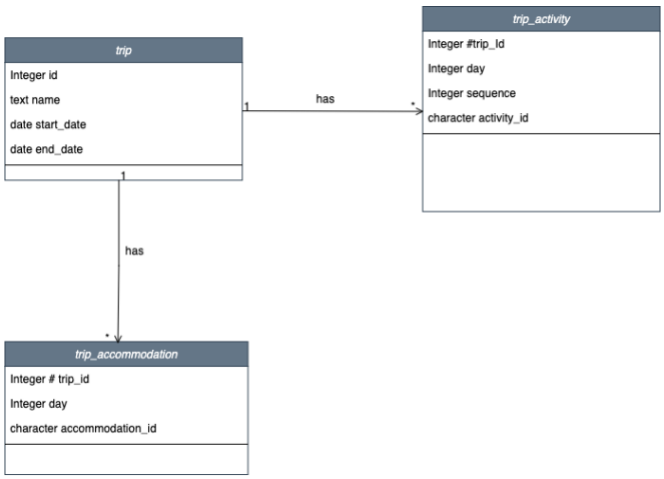
Validator (création des collections) -> Trigger (garantir la cohérence des données entre les collections)-> Insertion (ajout des données)

Voir les exemples suivants dans l'annexe :

- **Validators (MongoDB)**
- **Trigger (MongoDB)**
- **Insertion (MongoDB)**

B. PostgreSQL (trip-API)

1. Description des modèles



Ce schéma de base de données permet à l'utilisateur de gérer un **trip** en consignant chaque jour les lieux de séjour et les activités prévues, afin de concevoir et suivre l'itinéraire au jour le jour.

Tables / Collections

trip

- **Champs** : `id`, `name`, `start_date`, `end_date`
- **Description** : contient les informations générales d'un voyage complet.

trip_activity

- **Champs** : `trip_id`, `day`, `sequence`, `activity_id`
- **Description** : enregistre les activités prévues pour chaque jour du trip, dans un ordre défini (`sequence`).

trip_accommodation

- **Champs** : `trip_id`, `day`, `accommodation_id`
- **Description** : indique le lieu de logement réservé pour chaque nuit du trip.

Relations

- Un **trip** peut être lié à plusieurs **trip_activity** (plusieurs activités sur plusieurs jours).
- Un **trip** peut aussi être lié à plusieurs **trip_accommodation** (une nuitée par jour de voyage).

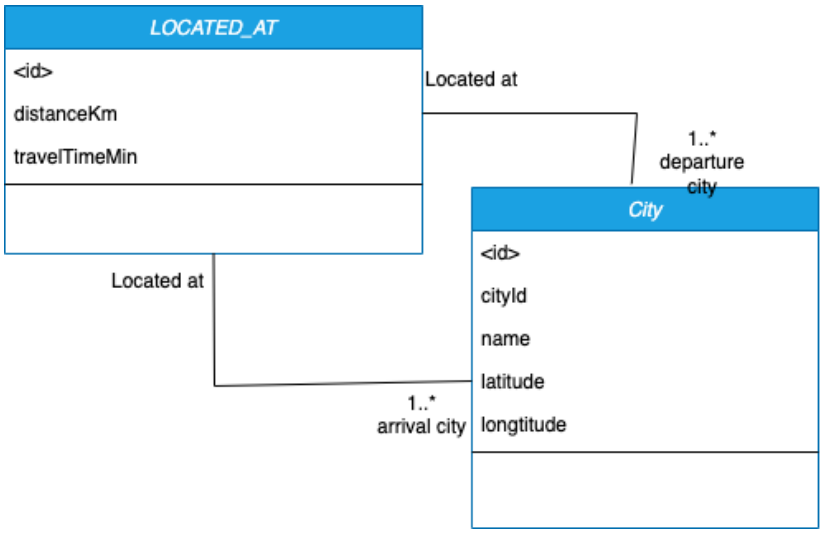
2. Exemples de commandes

Le code complet est disponible dans le répertoire :
`BDD_scripts_NGUYEN_YANG/PostgreSQL.`
Voir les exemples suivants dans l'annexe :

- **Création et trigger (PostgreSQL)**
- **Insertion (PostgreSQL)**

C. Neo4j (route-API)

1. Description des modèles



Dans ce modèle, chaque **ville** est un nœud City identifié par un **cityId** unique et géolocalisé par **latitude** et **longitude**. Chaque liaison routière entre deux villes est représentée par une relation **LOCATED_AT**, avec les attributs distanceKm et travelTimeMin.

Ces deux valeurs sont automatiquement calculées par le système. Concrètement, dès qu'une relation est ajoutée ou mise à jour entre deux villes, Neo4j utilise la fonction point.distance() pour calculer la **distance** et estimer le **temps de trajet**, à partir des coordonnées géographiques des villes.

De plus :

- cityId bénéficie d'une **contrainte d'unicité** (UNIQUE).
- cityName est **indexé** afin d'améliorer les performances des requêtes par nom de ville.

2. Exemples des commandes

Le code complet est disponible dans le répertoire :
BDD_scripts_NGUYEN_YANG /Neo4j.
Voir également les exemples fournis en annexe :

- **Contraintes et index (Neo4j)**
- **Création des nœuds City (Neo4j)**
- **Création dynamique des relations LOCATED_AT (Neo4j)**

IV. Requêtes et tests

A. MongoDB

Nous avons répondu aux trois questions de l'énoncé dans MongoDB : :

1.**Quelles sont les activités associées à un point d'intérêt donné ? Prenons poiId = "507f191e810c19729de860ea" comme exemple.**

Méthode :

```
db.activity.find(
  { "pointOfInterest.id": ObjectId("507f191e810c19729de860ea") },
  {
    name:    1,
    photos:  1,
    seasons: 1,
    price:   1,
    _id:     0
  }
);
```

Réponse :

```
{
  activities: [
    {
      name: 'Biodôme Tour',
      photos: [
        'https://example.com/biodomeTour.jpg'
      ],
      seasons: [
        'April',
        'May',
        'June'
      ],
      price: {
        adult: 24,
        child: 15
      }
    },
    {
      name: 'Photography Workshop',
      photos: [
        'https://example.com/photoWorkshop.jpg'
      ],
      seasons: [
        'May',
        'June'
      ],
      price: {
        adult: 50,
        child: 30
      }
    }
  ],
  poiName: 'Biodôme',
  poiAddress: '4777 Pierre-De Coubertin Ave, Montréal, QC'
}
```

2. Quels sont les hébergements d'une ville ?

Méthode :

```
const cityId = ObjectId("507f191e810c19729de860a1");
db.accommodation.find(
  { "city.id": cityId },
  {
    name:      1,
    address:   1,
    photos:    1,
    price:     1,
    available: 1,
    _id:       0
  }
);
```

Réponse :

```
{
  name: 'Hotel X',
  address: '123 Rue St-Pierre, Montréal, QC',
  photos: [
    'https://example.com/hotelx1.jpg'
  ],
  price: 120,
  available: true
}
{
  name: 'Auberge Y',
  address: '456 Boulevard St-Laurent, Montréal, QC',
  photos: [
    'https://example.com/aubergey1.jpg'
  ],
  price: 90,
  available: false
}
```

3. Quelles activités se font entre avril et juin ?

Méthode :

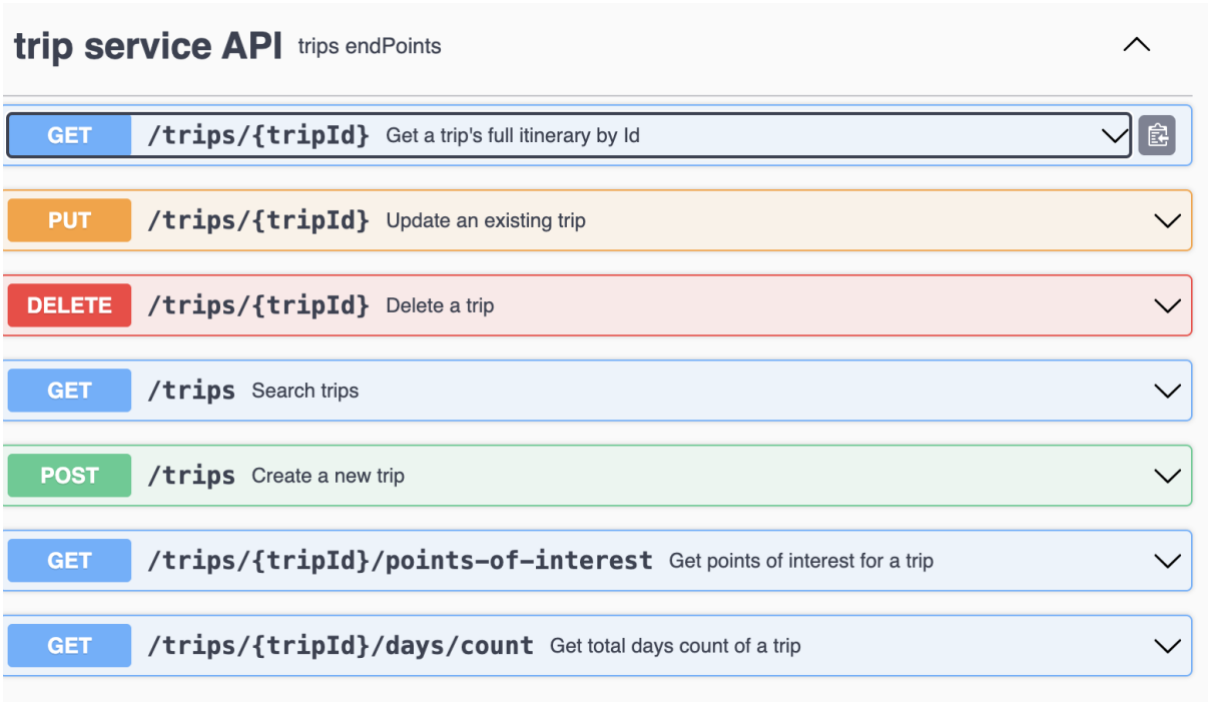
```
db.activity.find(
  {
    seasons: {
      $in: ["April", "May", "June"]
    }
  },
  {
    _id:     0,
    name:    1,
    seasons: 1
  }
);
```

Réponse :

```
  name: 'Biodôme Tour',
  seasons: [
    'April',
    'May',
    'June'
  ]
}
{
  name: 'Photography Workshop',
  seasons: [
    'May',
    'June'
  ]
}
{
  name: 'Architecture Tour',
  seasons: [
    'June',
    'July'
  ]
}
{
  name: 'Skypod Visit',
  seasons: [
    'June'
  ]
}
{
  name: 'Edge Walk',
  seasons: [
    'April',
    'May',
    'June'
  ]
}
{
  name: 'Exhibition Tour',
  seasons: [
    'April',
    'May',
    'June'
  ]
}
{
  name: 'Workshop',
  seasons: [
    'May',
    'June'
  ]
}
```

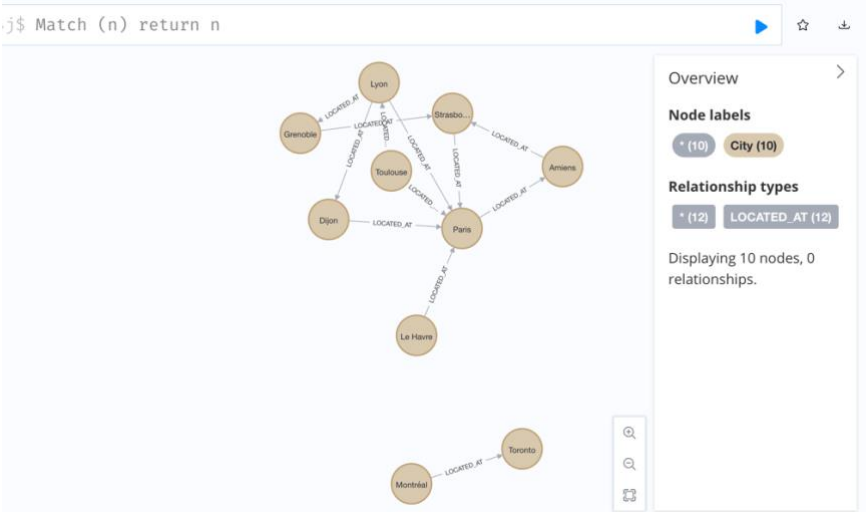
B. PostgreSQL

Ce base de données est principalement chargé de la **gestion des trips**. Voici quelques endpoints associés :



C. Neo4j

Voici un aperçu clair des **nœuds (nodes)** et **relations (edges)** utilisés dans votre modèle Neo4j:



Nous avons répondu aux trois questions de l'énoncé :

1. Quels sont les villes situées à moins de 300km d'une ville donnée ?Example : Paris

```
MATCH (c1:City {name:"Paris"}), (c2:City)
WHERE c1<>c2
  AND point.distance(
        point({latitude:c1.latitude, longitude:c1.longitude}),
        point({latitude:c2.latitude, longitude:c2.longitude})
      ) <= 300000
RETURN c2.name    AS nearbyCity,
       point.distance(
         point({latitude:c1.latitude, longitude:c1.longitude}),
         point({latitude:c2.latitude, longitude:c2.longitude})
       )/1000        AS distKm;
```

	nearbyCity	distKm
1	"Amiens"	115.5663382576141
2	"Dijon"	262.9763479210604
3	"Le Havre"	178.08309599089986

2. Quelle est le temps de trajet et la distance entre 2 villes données ? (Example : Paris et Lyon))

```
MATCH (c1:City {name:"Paris"})-[r:LOCATED_AT]-(c2:City {name:"Lyon"})
RETURN r.distanceKm   AS distanceKm,
       r.travelTimeMin AS travelTimeMin;
```

	distanceKm	travelTimeMin
1	391.9	294

3. Étant données une ville de départ et une ville d'arrivée, quels sont les différentes villes possibles à visiter entre les 2 ? (exemple: Paris et Grenoble)

```
1 MATCH path = shortestPath(
2         (start:City {name:"Paris"})-[:LOCATED_AT*]-(end:City {name:"Grenoble"})
3       )
4 UNWIND nodes(path)[1..-1] AS intermediate
5 RETURN DISTINCT intermediate.name AS cityOnRoute;
```

Table | Text

	cityOnRoute
1	"Strasbourg"

V. Conclusion

Dans ce projet, nous avons consacré un peu plus d'une semaine à la conception et à l'optimisation des structures de base de données. Avec l'évolution continue des applications microservices, nous avons également ajusté nos modèles de données de manière itérative. Yang et Minh ont travaillé ensemble sur la conception globale, chacun en charge d'un sous-système correspondant à des microservices spécifiques.

Phase initiale- Après avoir clarifié les besoins fonctionnels de chaque microservice, nous avons défini l'architecture du système et réparti les responsabilités de modélisation :**Yang** a pris en charge **MongoDB**,**Minh** s'est occupé de **Neo4j** et **PostgreSQL**.

Pour MongoDB, le principal enjeu était d'identifier les cas où certaines données pouvaient être **intégrées (*embedded*)** dans une collection pour optimiser les performances de lecture. Yang a conçu des intégrations (*embedding*) et mis en place des **triggers** dans **MongoDB Atlas** pour garantir la cohérence des données intégrées. Ce travail a inclus :la conception des triggers, leur déploiement sur la plateforme MongoDB Atlas, la validation de leur bon fonctionnement, ainsi que l'ajustement de la logique en interaction avec les microservices.

Dans **PostgreSQL**, Minh a conçu trois tables : trip, trip_activity et trip_accommodation, en définissant les relations et les contraintes nécessaires à la gestion des itinéraires. Dans **Neo4j**, il a modélisé les villes sous forme de nœuds (City) reliés par des relations LOCATED_AT, et a utilisé la fonction native point.distance() pour **calculer dynamiquement les distances et les temps de trajet** entre villes, simplifiant ainsi le traitement côté application.

Principales difficultés rencontrées

1. **Limites de la conception initiale**
   Malgré une première modélisation aussi complète que possible, de nouvelles contraintes métiers sont apparues lors du développement des microservices, rendant nécessaire une **révision fréquente** des schémas de données.
2. **Problèmes de cohérence dans MongoDB**
   Le passage d'un modèle normalisé à un modèle partiellement intégré a soulevé des **problèmes de synchronisation**. Grâce à l'utilisation de **MongoDB Atlas**, un service cloud entièrement managé, nous avons pu automatiser la synchronisation via des **triggers**, assurant une mise à jour efficace et cohérente des documents intégrés.

Ce travail nous a permis d'approfondir notre compréhension des systèmes de gestion de bases de données hétérogènes et de construire une architecture évolutive, prête à intégrer de nouvelles fonctionnalités dans le futur.

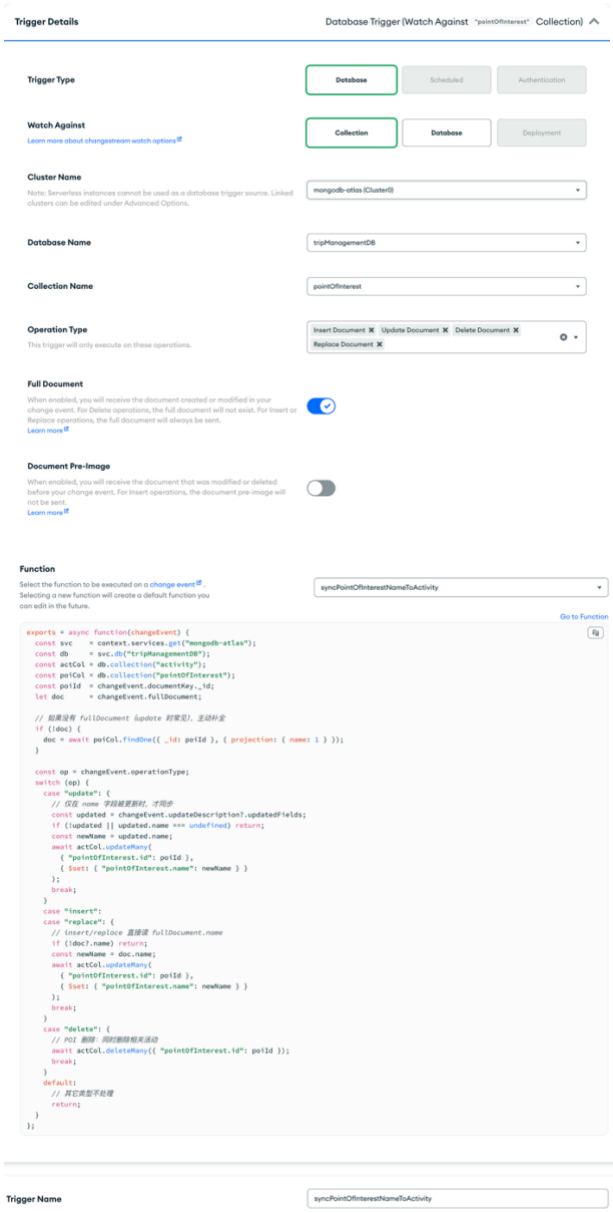
VI. Annexes – Examples des scripts

Seuls quelques scripts sont fournis ici à titre d'exemple. Pour le contenu complet, veuillez consulter le fichier BDD_scripts_NGUYEN_YANG.

Validators (MongoDB)

```
db.createCollection("city", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      description: "A city with denormalized POIs and accommodations",
      required: ["_id", "name", "photos", "geoInfo", "accommodations", "pointsOfInterest"],
      properties: {
        _id: {
          bsonType: "objectId",
          description: "Unique identifier for the city"
        },
        name: {
          bsonType: "string",
          description: "Name of the city"
        },
        photos: {
          bsonType: "array",
          description: "Array of URLs of city photos",
          items: {
            bsonType: "string",
            pattern: "^https?://",
            description: "A valid HTTP or HTTPS URL"
          }
        },
        geoInfo: {
          bsonType: "object",
          description: "Geographical coordinates",
          required: ["lat", "lon"],
          properties: {
            lat: {
              bsonType: ["double", "decimal"],
              minimum: -90,
              maximum: 90,
              description: "Latitude in degrees"
            },
            lon: {
              bsonType: ["double", "decimal"],
              minimum: -180,
              maximum: 180,
              description: "Longitude in degrees"
            }
          }
        },
        accommodations: {
          bsonType: "array",
          description: "Denormalized list of accommodation summaries",
          items: {
            bsonType: "object",
            required: ["id", "name", "price", "available"],
            properties: {
              id: {
                bsonType: "objectId",
                description: "Reference to an accommodation document"
              },
              name: {
                bsonType: "string",
                description: "Accommodation name"
              },
              price: {
                bsonType: ["int", "double", "decimal"],
                minimum: 0,
                description: "Nightly price in USD (or smallest currency unit)"
              },
              available: {
                bsonType: "bool",
                description: "Availability flag"
              }
            }
          }
        },
        pointsOfInterest: {
          bsonType: "array",
          description: "Denormalized list of POI summaries",
          items: {
            bsonType: "object",
            required: ["id", "name"],
            properties: {
              id: {
                bsonType: "objectId",
                description: "Reference to a pointOfInterest document"
              },
              name: {
                bsonType: "string",
                description: "POI name"
              }
            }
          }
        }
      }
    }
  }
});
```

Trigger (MongoDB)



Insertion (MongoDB)

```
const city1Id = ObjectId("507f191e810c19729de860a1");

const city2Id = ObjectId("507f191e810c19729de860a2");

db.city.insertMany([

 {

  _id: city1Id,

  name: "Montréal",

  photos: [

   "https://example.com/montreal1.jpg",

   "https://example.com/montreal2.jpg"

  ],

  geoInfo: { lat: 45.50, lon: -73.57 },
```

```
  accommodations: [
   { id: acc1Id, name: "Hotel X", price: 120, available: true },
   { id: acc2Id, name: "Auberge Y", price:  90, available: false }
  ],
  pointsOfInterest: [
   { id: poi1Id, name: "Biodôme" },
   { id: poi2Id, name: "Notre-Dame Basilica" }
  ]
 },
 {
  _id: city2Id,
  name: "Toronto",
  photos: [
   "https://example.com/toronto1.jpg",
   "https://example.com/toronto2.jpg"
  ],
  geoInfo: { lat: 43.65, lon: -79.38 },
  accommodations: [
   { id: acc3Id, name: "Fairmont Royal York", price: 250, available: true },
   { id: acc4Id, name: "The Drake Hotel", price: 180, available: true }
  ],
  pointsOfInterest: [
   { id: poi3Id, name: "CN Tower" },
   { id: poi4Id, name: "Royal Ontario Museum" }
  ]
 }
]);
```

Création et trigger (PostgreSQL)

```
-- 1) Main table: trip
CREATE TABLE trip (
  id         SERIAL      PRIMARY KEY,
  name       TEXT        NOT NULL,
  start_date DATE        NOT NULL,
  end_date   DATE        NOT NULL,
  -- Ensure that the end date is no earlier than the start date
```

```
  CONSTRAINT chk_trip_dates CHECK (end_date >= start_date)

);



-- 2) Trigger function: Ensure day <= total trip days

CREATE OR REPLACE FUNCTION check_trip_day()

RETURNS TRIGGER AS $$

DECLARE

  max_days INT;

BEGIN

  SELECT (t.end_date - t.start_date + 1)

    INTO max_days

    FROM trip t

   WHERE t.id = NEW.trip_id;



  IF NEW.day > max_days THEN

    RAISE EXCEPTION

      'trip_id=% day=% Exceeding the total number of days for the trip %', NEW.trip_id, NEW.day, max_days;

  END IF;

  RETURN NEW;

END;

$$ LANGUAGE plpgsql;
```

Insertion (PostgreSQL)

```
            INSERT INTO trip(id, name, start_date, end_date)

VALUES

  (1, 'Montreal & Toronto Tour', '2025-07-01', '2025-07-04');



INSERT INTO trip_activity(trip_id, day, sequence, activity_id) VALUES
```

```
(1, 1, 1, '507f191e810c19729de860f1'),

(1, 1, 2, '507f191e810c19729de860f2'),

(1, 2, 1, '507f191e810c19729de860f3'),

(1, 2, 2, '507f191e810c19729de860f4'),

(1, 3, 1, '507f191e810c19729de860f5'),

(1, 3, 2, '507f191e810c19729de860f6'),

(1, 4, 1, '507f191e810c19729de860f7'),

(1, 4, 2, '507f191e810c19729de860f8');



INSERT INTO trip_accommodation(trip_id, day, accommodation_id) VALUES

(1, 1, '507f191e810c19729de860d1'),

(1, 2, '507f191e810c19729de860d2'),

(1, 3, '507f191e810c19729de860d3'),

(1, 4, '507f191e810c19729de860d4');
```

Contraintes et index (Neo4j)

```
CREATE CONSTRAINT cityIdUnique IF NOT EXISTS
  FOR (c:City) REQUIRE c.cityId IS UNIQUE;

CREATE INDEX cityNameIndex IF NOT EXISTS
  FOR (c:City) ON (c.name);
```

Création des nœuds City (Neo4j)

```
CREATE
  // Canada
  (m:City {cityId:'507f191e810c19729de860a1', name:'Montréal', latitude:45.50, longitude:-73.57}),
  (t:City {cityId:'507f191e810c19729de860a2', name:'Toronto',  latitude:43.65, longitude:-79.38}),
  // France
  (p:City {cityId:'507f191e810c19729de860b1', name:'Paris',      latitude:48.8566, longitude:2.3522}),
  (a:City {cityId:'507f191e810c19729de860b2', name:'Amiens',     latitude:49.8941, longitude:2.2958}),
  (s:City {cityId:'507f191e810c19729de860b3', name:'Strasbourg', latitude:48.5734, longitude:7.7521}),
  (ly:City{cityId:'507f191e810c19729de860b4', name:'Lyon',       latitude:45.7640, longitude:4.8357}),
  (to:City{cityId:'507f191e810c19729de860b5', name:'Toulouse',   latitude:43.6047, longitude:1.4442}),
  (g:City {cityId:'507f191e810c19729de860b6', name:'Grenoble',   latitude:45.1885, longitude:5.7245}),
  (d:City {cityId:'507f191e810c19729de860b7', name:'Dijon',      latitude:47.3220, longitude:5.0415}),
  (lh:City{cityId:'507f191e810c19729de860b8', name:'Le Havre',   latitude:49.4944, longitude:0.1079});
```

Création dynamique des relations LOCATED_AT (Neo4j)

```
WITH [
  ['Montréal','Toronto'],
  ['Paris','Amiens'],
  ['Amiens','Strasbourg'],
  ['Strasbourg','Paris'],
  ['Grenoble','Strasbourg'],
  ['Lyon','Paris'],
  ['Lyon','Grenoble'],
  ['Lyon','Dijon'],
  ['Dijon','Paris'],
  ['Le Havre','Paris'],
  ['Toulouse','Paris'],
  ['Toulouse','Lyon']
] AS edges
UNWIND edges AS pair
MATCH (c1:City {name: pair[0]}), (c2:City {name: pair[1]})
WITH
  c1, c2,
  point.distance(
    point({latitude:c1.latitude, longitude:c1.longitude}),
    point({latitude:c2.latitude, longitude:c2.longitude})
  ) AS distMeters
MERGE (c1)-[r:LOCATED_AT]->(c2)
SET
  r.distanceKm    = round(distMeters / 1000.0, 1),
  r.travelTimeMin = toInteger(round((distMeters / 1000.0) / 80 * 60, 0));
```