SAÉ Migration de données vers un environnement NoSQL

Contexte futuriste

À la suite de la tempête de 2050 qui a bouleversé la ville, Paris a dû réinventer ses transports urbains. Désormais en 2055, la métropole ambitionne de faire évoluer son système de suivi et d'analyser plus finement l'empreinte écologique de ses déplacements.

Les données actuelles sont stockées dans une base relationnelle `SQLite` contenant :

- les lignes de bus, arrêts, véhicules, chauffeurs,
- les données de trafic en temps réel (incidents : retards, pannes),
- les données de géolocalisation (coordonnées GPS des arrêts et positions des véhicules),
- et les indicateurs environnementaux (émissions de CO_2, consommation).

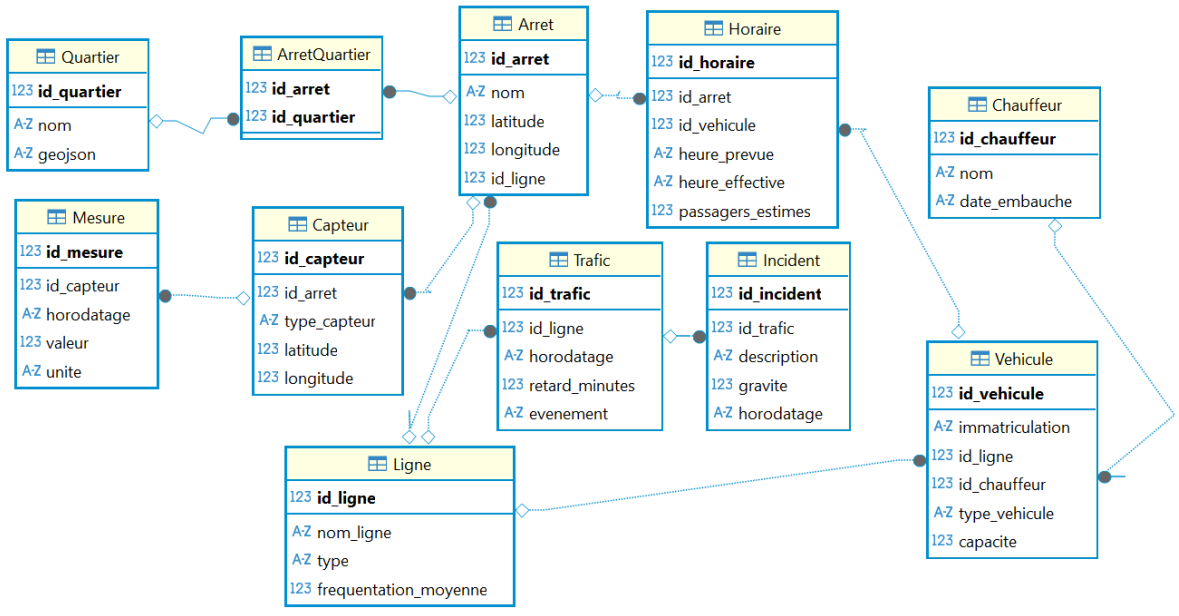
La collectivité veut migrer vers un environnement NoSQL (MongoDB) afin de mieux gérer des flux temps réel, des données semi-structurées (capteurs) et une visualisation géographique dynamique.

Pseudo schéma relationnel

Table	Description	Clés primaires / étrangères	Nb tuples
Ligne	Ligne de transport (bus, tram, métro)	id_ligne (PK)	100
Arret	Point d'arrêt avec coordonnées GPS	id_arret (PK), id_ligne (FK)	5 000
Chauffeur	Personnel roulant	id_chauffeur (PK)	3 000
Vehicule	Bus ou tram affecté à une ligne	id_vehicule (PK), id_ligne (FK), id_chauffeur (FK)	2 000
Horaire	Passage prévu d'un véhicule à un arrêt	id_horaire (PK), id_arret (FK), id_vehicule (FK)	200 000
Trafic	Informations de trafic (retards, incidents)	id_trafic (PK), id_ligne (FK)	50 000
Incident	Détail des incidents signalés	id_incident (PK), id_trafic (FK)	20 000
Capteur	Données environnementales (CO_2, bruit, température)	id_capteur (PK), id_arret (FK)	4 658
Mesure	Relevés des capteurs	id_mesure (PK), id_capteur (FK), horodatage	208 706
Quartier	Découpage géographique de la ville	id_quartier (PK)	200
ArretQuartier	Relation n–n entre Arrêt et Quartier	id_arret (FK), id_quartier (FK)	5 257

Vous disposez de la base de données `Paris2055.sqlite`. Pour visualiser le contenu de ces tables :

- Outil en ligne sut Github : https://inloop.github.io/sqlite-viewer/
- Installer DBBrowser for SQLite : https://sqlitebrowser.org ou DBeaver https://dbeaver.io/



SQLite *est un système de* ***bases de données relationnelles*** *qui a la particularité de fonctionner sans serveur, on dit aussi "standalone" ou "base de données embarquée". Ecrit en langage C et grâce à son extrême légèreté, SQLite est l'un des moteurs de base de données les plus utilisés au monde.*

L'objectif de saé est de migrer les données d'une base de données relationnelles à une base de données NoSQL et d'exploiter les données de cette dernière. Le choix a été fait de passer du format *SQLite* à un format *MongoDB*. Il va donc falloir préparer la migration et l'exploitation de la BD à l'aide du langage Python. Ce travail va se faire en 4 étapes qui constitueront les 4 parties de votre travail à réaliser pour cette saé :

1. Création de requêtes de tests en SQL sur la BD *SQLite* ;
2. Réflexion sur le format des données à obtenir en NoSQL (nombre de collections à créer) et écriture du script Python permettant le passage de *SQLite* à *MongoDB* ;
3. Création des requêtes de tests en SQL au nouveau format *MongoDB* pour s'assurer du bon déroulement de la migration.
4. Un tableau de bord avec cartographie et carte choroplèthe.

Partie 1 — Requêtes SQL sur la base relationnelle `SQLite`

Accès à SQLite dans Python : nous allons utiliser la librairie `sqlite3` pour créer la connexion à la base de données. La librairie `pandas` permettra d'exécuter une requête `SELECT` sur une BD et de récupérer le résultat dans un `DataFrame`. Voici le code pour créer la connexion et récupérer le contenu de la table `Pays` par exemple :

```
import sqlite3
import pandas

# Création de la connexion
conn = sqlite3.connect("covid.sqlite")

# Récupération du contenu de Customers avec une requête SQL
df = pandas.read_sql_query("SELECT * FROM pays;", conn)

# Fermeture de la connexion
conn.close()
```

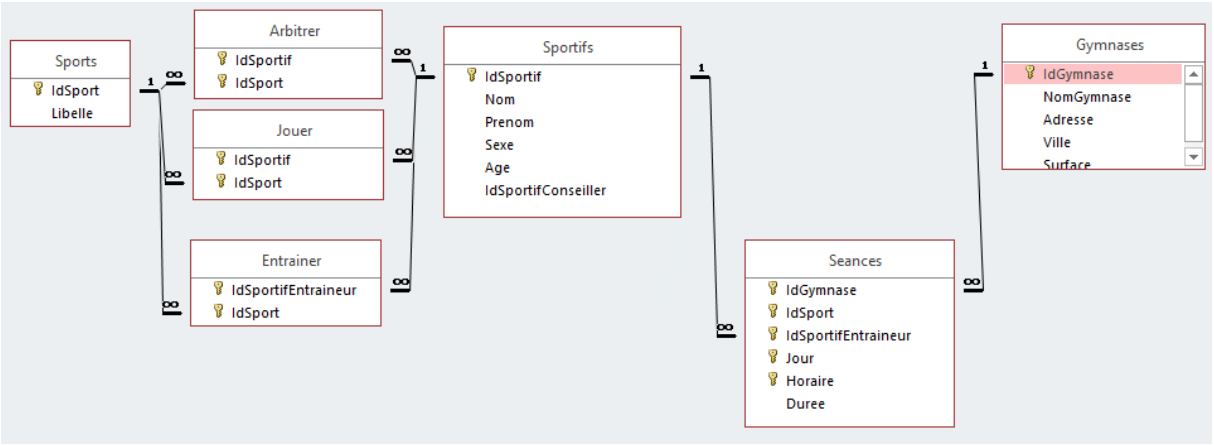
	TRAVAIL A FAIRE
1	Ecrire les requêtes suivantes en SQL et les exécuter en Python en faisant un choix d'ordonnancement du résultat (tri), s'il n'est pas précisé ou naturel. Le résultat devra être stocké dans un fichier csv, cela permettra de mieux comparer les résultats après migration (partie 3) : a. Moyenne des retards par ligne de transport b. Nombre moyen de passagers transportés par jour et par ligne c. Taux d'incident sur chaque ligne d. Emissions moyennes de CO_2 par véhicule e. Top 5 des quartiers avec le plus de nuisances sonores (moyenne capteur bruit) f. Liste des lignes sans incident mais avec retards > 10 min g. Taux de ponctualité global (trajets sans retard / total) h. Nombre d'arrêts par quartier i. Corrélation entre trafic et pollution (CO_2 ↑ quand retard ↑) par ligne j. Moyenne de température par ligne (via capteurs proches) k. Performance chauffeur (retard moyen par chauffeur) l. % de véhicules électriques par ligne de bus m. Requête CASE WHEN : classification du niveau de pollution (faible/moyen/élevé) par capteur avec sa localisation n. Trouver une autre requête utilisant un *case when* et qui ait du sens dans ce contexte.

Partie 2 — Migration vers MongoDB : Passage modèle relationnel - modèle à base de documents

Objectif : Transformer le modèle relationnel en un certain nombre de collections dans MongoDB.

Rappel de cours : *Passage MCD -> JSON*

Pour vous aider dans le processus de migration, commencez par travailler sur cette nouvelle base de données exemple nommée `Gymnase`, disponible au format SQLite sur Updago. Voici son schéma relationnel graphique sous ACCESS :



Par souci de clarté, il manque ici visuellement quelques lisaisons : de `Seances` vers la table `Sports`, de `idSportifConseiller` vers `idSportif`, `Jour` et `Horaire`...

Dans ce schéma et après analyse, on peut décider de créer **2 collections** :

- **Sportifs** : chaque document concernera un seul *sportif*, dans lequel on notera en plus les sports qu'il *joue*, qu'il *entraîne* et qu'il *arbitre* ;
- **Gymnases** : ici, chaque document concernant un seul *gymnase*, dans lequel on ajoutera les informations de toutes les *séances* prévues, dans un tableau.

Avec la base de données `gymnase` disponible sur Updago :

```
import sqlite3
import pandas

conn = sqlite3.connect("gymnase.sqlite")
```

1. Récupération des informations dans la BD

On doit tout d'abord récupérer les informations des deux tables `Gymnases` et `Seances`, que l'on va stocker dans des DataFrames.

Attention : on va tout de suite récupérer le nom du sport en chaîne de caractères. On supprimera l'identifiant du sport ultérieurement.

```
gymnases = pandas.read_sql_query("SELECT * FROM Gymnases;",conn)
seances = pandas.read_sql_query("SELECT * FROM Seances INNER JOIN Sports USING (IdSport);",
conn)
```

2. Intégration des séances dans le DataFrame `gymnases`

Maintenant, nous devons ajouter la liste des séances comme colonne du DataFrame `gymnases`. Pour cela, si nous considérons le gymnase avec l'identifiant 1, nous pouvons déjà récupérer l'ensemble de ces séances :

```
id = 1
df = seances.query('IdGymnase == @id')
```

Ensuite, pour supprimer les colonnes `IdGymnase` et `IdSport` :

```
df = seances.query('IdGymnase == @id').drop(columns = [["IdGymnase", "IdSport"]])
```

Enfin, pour transformer le résultat (qui est un DataFrame) en un dictionnaire python, nous utilisons la fonction `to_dict()`, avec en paramètre `orient` égal à `records`, ceci permet de faire un tableau de dictionnaires :

```
df = seances.query('IdGymnase == @id').drop(columns =
                                ["IdGymnase","IdSport"]).to_dict(orient = "records")
```

Ce code peut maintenant s'intégrer dans une *list comprehension* pour créer une liste de dictionnaires pour chaque gymnase.

```
liste = [seances.query('IdGymnase == @id').drop(columns=["IdGymnase",
              "IdSport"]).to_dict(orient = "records") for id in gymnases.IdGymnase]
```

On peut maintenant ajouter cette liste des séances au DataFrame `gymnases` :

```
gymnases = gymnases.assign(Seances = liste)
print(gymnases.head())
```

3. Intégration du DataFrame `gymnases` à MongoDB

Créer ou non au préalable la BD « `SAE` » dans Compass ainsi que la collection `Gymnases`, y placer toutes les données grâce aux fonctions `to_dict()` et `insert_many()` applicables sur la collection :

```
connect = pymongo.MongoClient("mongodb://127.0.0.1:27017/")
db = connect.SAE   # Créé la base SAE si elle n'existe pas dans MONGODB
db.Gymnases.insert_many(gymnases.to_dict(orient = "records"))
```

Vous pouvez aller voir le résultat dans Compass (**faire un reload data du server**). Vous pouvez aussi tester directement en interrogeant MongoDB pour le nombre de documents de la collection :

```
print(db.Gymnases.count_documents({}))
```

Et aussi sur le contenu de la collection :

```
print(list(db.Gymnases.find()))
```

4. Suppression d'une collection

Si vous avez fait une erreur, vous pouvez supprimer une collection (voire une BD) soit directement dans Compass, soit en passant par du code python. Le code ci-dessous permet de supprimer la collection `Gymnases` par exemple.

```
db.Gymnases.drop()
```

5. Création de la collection `Sportifs`

⇨ **A FAIRE !!!**

6. Jointure entre deux collections

Supposons que vous avez créé la collection `Sportifs`, avec comme champ `IdSportif`. Si nous souhaitons récupérer les informations l'entraineur pour chaque séance, pour chaque gymnase, nous pouvons exécuter le code suivant :

```
df = pandas.DataFrame(list(db.Gymnases.aggregate([
    { "$limit": 1 },
    { "$unwind": "$Seances" },
    { "$lookup": {
       "from": "Sportifs",
       "localField": "Seances.IdSportifEntraineur",
       "foreignField": "IdSportif",
       "as": "Entraineur"
     }}
])))
```

Remarque : le résultat d'un `lookup` est forcément un tableau, même s'il n'y a qu'une seule valeur. A vous de faire le travail pour l'extraire dans un littéral simple (une chaîne de caractères) si vous le souhaitez (par exemple avec `$first`).

⇨ **A FAIRE si vous le souhaitez** : récupérer les informations du conseiller pour chaque sportif.

A partir du schéma de la base de données `Paris2055.sqlite` page 1, vous devez désormais réfléchir à la ré-organisation des données pour les placer dans des **documents** au format JSON, réunis dans une ou plusieurs collections.

	TRAVAIL A FAIRE
2	Vous devrez donc établir pour cette Saé : . le nombre de collections (chacune contenant idéalement un seul type de documents) . le schéma type des documents à l'intérieur de ces collections. . **Ecrire le script Python qui effectue la migration de la BD de la page 1 vers une BD MongoDB**

Partie 3 : Requête tests en MongoDB

	TRAVAIL A FAIRE
3	Ecrire les requêtes tests de la question 1 en MongoDB avec Python. Vérifier les résultats obtenus.

Partie 4 : Tableau de bord et cartographie à partir de la base MONGODB

	TRAVAIL A FAIRE
4	**Objectif** : Créer un tableau de bord interactif (Streamlit ou plotly ou Flask + Folium) comprenant : 1. Graphiques pandas/matplotlib : o histogramme des retards moyens par ligne, o courbe de tendance des émissions de CO_2, o pie chart de la répartition des véhicules par type. o Autres graphiques donnant du sens aux décideurs 2. Cartographie Folium : o Carte choroplèthe : niveau moyen de CO_2 par quartier. o Carte à marqueurs avec filtre(s) obligatoire(s) : ▪ chaque arrêt est un marqueur coloré selon le niveau de pollution, ▪ popup affichant : nom de l'arrêt, nombre de lignes, bruit moyen, température moyenne. o Sélection d'une ligne et visualisation des arrêts dans un tableau et sur une carte : Rajouter des indicateurs de votre choix et qui soient pertinents sur cette carte. **Remarque** : Commande pour lancer une application streamlit dans spyder : `!streamlit run "partie4.py"`

Vous devez déposer au plus tard le dimanche 11 janvier 23h59 sur Updago, l'ensemble des scripts Python de cette Saé :

- **Script Python partie 1 avec requêtes de tests en SQL**
- **Script Python partie 2 migration vers NoSQL**
- **Script Python partie 3 requêtes tests en MongoDB**
- **Scripts Python partie 4 (+ scripts web éventuels) pour le tableau de bord et la cartographie.**