Rapport Projet TER

Étudiants:

OUMEZZAOUCHE Mohamed

DURBASS (RIVIERE) Tasneem

NZAMBOUE TISSEU Hilary

Tuteurs :

Stéphane LOPES
Zoubida KEDAD

Introduction :

Dans le cadre de ce projet, nous avons reçu un scénario initial dans lequel une application web existante interagit avec 2 modèles de base de données différents - un modèle relationnel et un modèle non relationnel. Ainsi, ce projet a pour but d'étudier ces deux modèles de base de données afin de trouver une meilleure façon d'en extraire les données, et donc de mettre en place des techniques d'intégration de données.

Présentation des bases de données

	Judilibre	Piratage
Description	Projet concernant les décisions de justice.Données récupérées depuis le site de la 'Cour de cassation'	projet concernant le piratage. Base de donnée contenant toutes les informations regroupés dans le cadre de l'étude sur l'évolution du cyber-crime
Modèle de BD	Mongo DB	PostgreSql
Données	10 Collections	19 tables (relationnels)

PostgreSQL et MongoDB sont deux systèmes de gestion de bases de données (SGBD) très populaires qui ont des approches différentes en matière de stockage et de récupération des données. Dans ce rapport, nous allons examiner les différences et les similitudes entre ces deux SGBD.

Différences entre les 2 modèles :

PostgreSQL	MongoDB
Est un système de gestion de bases de données relationnelles (SGBDR): les données sont transférées dans des tables avec des relations entre elles.	est un SGBD NoSQL, qui stocke des données sous forme de documents JSON.
utilise SQL (Structured Query Language) pour les requêtes	utilise son propre langage de requête, basé sur les opérations de CRUD (Create, Read, Update, Delete).
utilise l'index B-tree pour accélérer les requêtes	utilise un système d'indexation de documents
offre une grande extensibilité grâce à sa prise en charge de langages de programmation tels que PL/SQL, PL/Python et PL/Perl	dispose également d'une grande extensibilité grâce à son architecture basée sur des plugins

prend en charge les transactions ACID (Atomicité, Cohérence, Isolation, Durabilité)	offre des transactions distribuées mais ne prend pas en charge les transactions ACID

Similitudes entre les 2 modèles:

1. **Open source** : Les deux SGBD sont disponibles sous licence open source.
2. **Scalabilité** : Les deux SGBD sont hautement évolutifs et peuvent être utilisés pour de grandes applications.
3. **Support communautaire** : Les deux SGBD disposent d'une grande communauté de développeurs et d'utilisateurs qui fournissent des ressources et des forums d'assistance.
4. **Gestion de l'authentification et des autorisations** : Les deux SGBD permettent une gestion de l'authentification et des autorisations basée sur des rôles.
5. **Réplication** : Les deux SGBD offrent des fonctionnalités de réplication pour la haute disponibilité.

Choix d'une solution

Lorsque nous avons essayé de décider quelle solution de base de données convenait le mieux à notre application, il était important de comprendre comment l'application utilise les bases de données. Une façon de procéder consiste à analyser les performances de chaque base de données séparément pour déterminer leurs forces et leurs faiblesses.

Solutions possibles au problème :

1. **Fusion**

La fusion de données à partir de deux bases de données différentes telles que MongoDB et PostgreSQL peut être un processus complexe en raison des différences entre les systèmes de gestion de bases de données.

La première étape à la réalisation de cette solution serait de choisir un seul modèle de base de données qui pourrait intégrer les données, ou les structures de données présentes dans les deux modèles précédemment utilisés.

Le choix d'une base de données est une décision importante pour tout projet de développement de logiciels ou d'applications. Il existe plusieurs critères à prendre en compte lors de la sélection d'une base de données. Nous nous sommes basé principalement sur les suivantes:

1. Performance : Les performances de la base de données sont un critère important à considérer. La mémoire rapide de traitement des requêtes, la disponibilité des

données, la capacité de traitement des données en temps réel et la gestion de la sont tous des aspects qui influencent les performances.
2. Scalabilité : La scalabilité est un autre critère important. Une base de données qui peut facilement s'adapter pour répondre aux besoins futurs est essentielle pour une croissance durable de l'application.
3. Modèle de données : Le modèle de données est un critère crucial. Les bases de données relationnelles sont adaptées aux applications qui manipulent des données intégrées, alors que les bases de données NoSQL sont conçues pour les données semi-structurées et non intégrées. Le choix du modèle de données doit être en fonction des besoins spécifiques de l'application.
4. Support de la communauté : La base de données doit être soutenue par une communauté active et dynamique de développeurs, de contributeurs et d'utilisateurs. La documentation, les forums, les ressources de formation et de support sont tous des éléments clés d'une communauté de support active.
5. Facilité d'utilisation : La facilité d'utilisation de la base de données est un aspect à considérer. La base de données doit être facile à installer, à configurer et à utiliser, avec une interface utilisateur claire et concise.
6. Intégration : La base de données doit pouvoir s'intégrer facilement avec les autres composants de l'application, tels que le serveur d'applications, le système de fichiers, les services de messagerie, etc.

De plus, les besoins spécifiques de l'application doivent être pris en compte pour trouver la base de données qui convient le mieux.

PostgreSQL est une base de données relationnelle qui est bien adaptée pour les applications qui manipulent des données intégrées/structurées. PostgreSQL est connu pour sa fiabilité, ses performances et sa robustesse.

MongoDB, en revanche, est une base de données NoSQL qui est adaptée aux applications qui manipulent des données semi-structurées et non structurées (*des données n'ayant pas un schéma fixe et qui peuvent varier en termes de structure et de format*). Elle offre des fonctionnalités avancées telles que la *réplication*, la *gestion du sharding* et la *gestion de l'agrégation*. Ces fonctionnalités avancées de MongoDB offrent des avantages significatifs en termes de disponibilité, de performance et de capacité à gérer des charges de travail de données importantes. Elles permettent aux applications de s'adapter à l'évolution des besoins et de garantir une expérience utilisateur fluide.

Il est important de noter que la fusion de données entre MongoDB et PostgreSQL peut être complexe en raison des différences de structure de données entre les deux bases de données. Il est recommandé de réaliser des tests de fusion de données dans un environnement de test avant de le faire sur un environnement de production.

2. Approche hybride

Pour une approche hybride, il existe plusieurs façons d'utiliser Postgresql et MongoDB ensemble dans notre application. Voici quelques possibilités :

1. **Utilisez Postgresql comme base de données principale et MongoDB comme base de données secondaire** pour stocker des données non critiques ou pour effectuer des analyses. Cette approche pourrait aider à décharger certaines des charges de travail lourdes en lecture de Postgresql et à améliorer les performances globales.
2. **Utilisez MongoDB comme base de données principale et Postgresql comme base de données secondaire** pour stocker des données plus structurées ou pour effectuer des transactions. Cette approche pourrait tirer parti de la flexibilité et de l'évolutivité de MongoDB tout en garantissant la fiabilité et la cohérence des données transactionnelles.
3. **Utilisez les deux bases de données de manière complémentaire**, chaque base de données étant responsable de différents types de données ou de différentes parties de l'application. Par exemple, vous pouvez utiliser Postgresql pour gérer les comptes d'utilisateurs et l'authentification, tout en utilisant MongoDB pour stocker et analyser de grandes quantités de contenu généré par les utilisateurs. On pourrait utiliser *Apache NiFi* comme un intermédiaire entre les deux bases de données.

1. Présentation de la solution Fusion

En général, comme l'application manipule des données intégrées(dans la base piratage) ainsi que des données semi-structurées(dans la base judilibre), alors nous avons conclu que **PostgreSQL** pouvait être le choix le plus adapté comme modèle de base de donnée cible en optant pour la fusion, en raison de sa richesse en fonctionnalités relationnelles.

En fait, nous avions trouvé que si on avait opter pour la ***migration de la BD postgreSQL vers la BD MongoDB***, pour migrer correctement, nous devrions probablement concevoir un nouveau modèle de données qui tire parti des qualités de la base de données NoSQL vers laquelle nous nous dirigeons et qui résout le problème qui nous a poussés à migrer. Cela veut dire qu'il faudrait probablement dénormaliser dans une certaine mesure (*il nous faudrait donc décider comment*).

Il nous faudrait donc nous préoccuper des choses que la base de données relationnelle prenait en charge et que la nouvelle pourrait ne pas prendre en charge : les transactions ACID ou les contrôles d'intégrité relationnelle, par exemple.

Même en ayant une BD simple, en faisant la transformation des données de PostgreSQL vers mongoDB, nous devions effectuer le changement correctement et concevoir le modèle non relationnel comme il se doit. C'est le point le plus délicat.

Enfin, nous devrons tenir compte des clients qui utilisent déjà la base de données - comme l'ancienne base de données SQL ne fonctionnera pas sur la nouvelle base de données NoSQL, concernant la nouvelle API aux logiciels qui accèdent aux données, serait-elle plus complexe à mettre en place?

Alors, la fusion a plutôt été faite en effectuant une migration des données de MongoDB vers PostgreSQL car nous avons pensé que ca serait moins compliqué de définir une structure

pour des données non-structurées plutôt que de casser les structures de données déjà existantes en structures de données semi-structurées.

Pour mettre en œuvre cette solution, nous avons implémenté un code/scripte en **python** personnalisé pour extraire les données des deux bases et les fusionner en une seule base de données. L'implémentation du scripte suit les étapes suivantes:

1. **Choix d'un modèle de BD adapté** : PostgreSQL
2. **Analyse des schémas de données** : La première étape consiste à analyser les schémas de données des deux bases de données et à identifier les données à fusionner. Ici on migre les données de la BD mongoDB vers la BD relationnelle, PostgreSQL.
3. Installation de Python et des librairies à utiliser
4. Création d'une base de donnée en local s'appelant "**piratage_judilibre**" & Création d'un utilisateur en local (user_ter) ayant accès à cette base (la solution a été créé pour faire l'intégration en local)
5. **Exportation des données** : Les données à fusionner doivent être exportées depuis les deux bases de données. Il existe plusieurs outils d'exportation de données disponibles pour MongoDB et PostgreSQL, tels que *mongodump* pour MongoDB et *pgdump* pour PostgreSQL. Nous avons utilisé pgdump pour avoir les données de la base Piratage et nous n'avions qu'à exécuter le fichier .dump généré, sur la base de données cible comme ils étaient tous les deux du même modèle(avec la commande psql).

```
# A executer les commandes suivante sur le terminal
# Créer une sauvegarde de la base de données source
# "pg_dump -U piratage -h pg.adam.uvsq.fr -d piratage -f piratage.dump"

# Restaurer le fichier dump dans la base de données de destination
# "psql -U user_ter -h localhost -d piratage_judilibre -f piratage.dump"
```

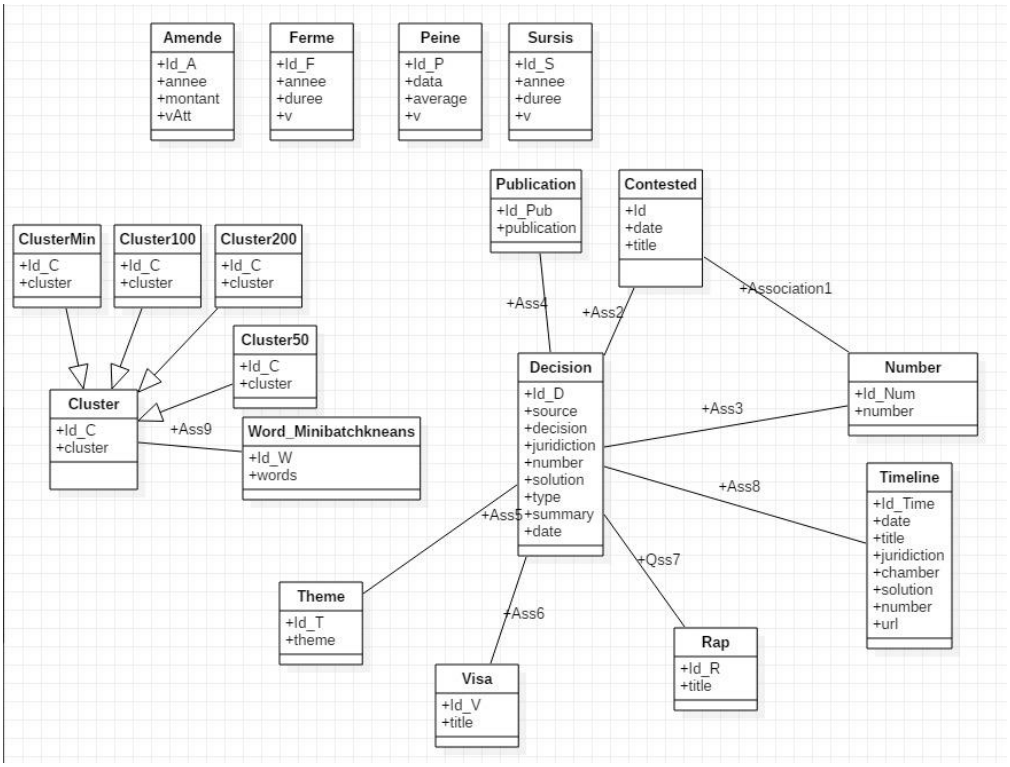
Puis nous avons utilisé la bibliothèque 'pymongo' pour pouvoir collecter les données des différentes collections de Judilibre en format 'clé-valeur'(illustré ci-dessous).

```
amende_collection = mongo_db["amende"].find()
for x in amende_collection:
  print(x)
```

```
pg_data = []
for doc in amende_collection:
    pg_doc = {
        "annee": doc["annee"],
        "montant": doc["montant"],
        "__v": doc["__v"]
    }
    pg_data.append(pg_doc)
```

6. **Conversion/Transformation des données** : Comme les données exportées depuis MongoDB sont dans un format semi-structuré, il a donc été nécessaire de les convertir dans un format plus compatible au modèle postgreSQL. Bien qu'il soit possible d'utiliser des outils de conversion tels que Talend, ou Pentaho, nous avons choisi de s'occuper de la transformation à la main, en utilisant nos connaissances acquises dans le cours de conception de BD.

   Nous avons un peu procédé à un 'reverse engineering' à partir du modèle NoSQL de mongoDB vers un modèle entités associations puis vers un modèle relationnelle de table postgreSQL. A noter que l'analyse des structures de données présentes dans les collections a été faite avec l'outil MongoDB Compass qui offre la possibilité d'analyser les schémas des collections.



Dans le modèle E/A ci-dessus, nous représentons toutes les relations entre les collections retrouvées dans la base de données Judilibre.

Après avoir effectuer la transformation du modèle E/A en modèle relationnelle, les tables obtenues (avec un préfixe ‘jl_’) sont les suivantes :

- amende(id, Année, montant, __v),
- cluster_minibatchkmeans(id, cluster),
- cluster100(id, cluster),
- cluster200(id, cluster),,
- cluster50(id, cluster),
- decision(id, #contested, source, text, chamber, decision_date, jurisdiction, number, solution, type, summary, update_date, forward, __v),
- ferme(id, année, durée, __v),
- peine(id, data, average, __v),
- sursis(id, année, durée, __v),
- word_minibatchkmeans(id, cluster, words).

d’autres tables qui pourraient être utilisées pour les associations:

- contested(id, date, title, #number),
- number(id, number),
- numberDecisions(idDecision, idNumber),
- publication(id, publication),
- pubDecision(idPublication, idDecision),
- theme(id, theme),
- themeDecision(idTheme, idDecision),
- visa(id, title),
- visaDecision(idVisa, idDecision),
- rapprochement(id, title),
- rapDecision(idRapprochement, idDecision),
- timeline(id, date, title, jurisdiction, chamber, solution, number, url)
- timelineDecision(idTimeline, idDecision)

Comme les données récupérées étaient semi-structurées, nous avons vu que parfois il y avait soit des valeurs de champs manquants ou des champs manquants dans les collections. Par exemple, dans la collection décision, dans MongoDB Compass on peut voir que le champ timeline est souvent null. Pour stocker ces champs là, il faudrait alors faire des conditions dans le script pour voir si le champ ou la valeur du champ existe ou pas.

7. Fusion **des données** : Une fois les données converties (les tables créées dans la nouvelle base cible), il a été temps de les fusionner en utilisant les facilités offertes par les librairies utilisées sur python comme: *pymongo, pandas ou psycopg2*
8. **Importation des données fusionnées** : Enfin, les données fusionnées peuvent être importées dans une nouvelle base de données ou dans une des bases de données existantes.
9. **Vérification des données** : Une fois les données fusionnées et importées, il est important de vérifier leur qualité et leur intégrité. La vérification des données a été effectuée à l'aide de l’outil pgAdmin sur nos machines en locale

Néanmoins, la fusion a quand même un certain nombre d'implications et de complications potentielles, qui demande du temps et de la recherche supplémentaire. Certaines de ces implications et complications sont:

- Différences dans la conception des schémas: la migration des données a nécessité des changements significatifs au niveau du schéma. Cette solution peut exiger une planification minutieuse pour s'assurer que les données sont correctement traduites entre les deux systèmes.

  Par exemple : le type de transformation choisi pourrait être problématique. Nous sommes conscients que l'inconvénient de notre type de transformation choisi serait le fait que cela implique de créer plusieurs tables et donc plusieurs jointures, qui pourraient complexifier la récupération des données et être coûteux en termes de performances.

  Néanmoins, Il existe quand même d'autre types de transformations qui auraient pu être utilisés: par exemple, les structures de champs 'String array' en Mongodb pourraient être transformées en 'TEXT[]' en PostgreSQL. De plus, les 'Object Array' en Mongodb pourraient être transformées en 'JSONB[]'. Pour la solution, nous avons opté pour la simplicité et n'avons pas risqué d'utiliser des notions que l'on ne maîtrise pas vraiment.

- Différences dans le langage d'interrogation : MongoDB utilise un langage de requête basé sur les documents, tandis que PostgreSQL utilise un langage de requête basé sur SQL. Cela signifie que les requêtes écrites pour MongoDB peuvent devoir être réécrites pour fonctionner avec PostgreSQL. Cela peut prendre du temps et nécessiter une expertise importante dans les deux systèmes.
- Différences de performances : MongoDB et PostgreSQL ont des caractéristiques de performance différentes, ce qui peut entraîner des différences dans le temps d'exécution des requêtes et l'utilisation des ressources. Cela signifie que les requêtes qui fonctionnent bien sur MongoDB peuvent ne pas fonctionner aussi bien sur PostgreSQL.
- Considérations de sécurité : MongoDB et PostgreSQL ont des modèles de sécurité et des mécanismes de contrôle d'accès différents. Cela signifie qu'il peut être nécessaire d'évaluer soigneusement les considérations de sécurité lors de la migration des données entre les deux systèmes.

La migration des données de MongoDB vers PostgreSQL peut avoir un certain nombre d'implications et de complications potentielles mais une planification minutieuse et la prise en compte de ces facteurs peuvent contribuer à la réussite du processus de migration.

D'autres approches a la solution de Fusion:

Utiliser des outils tiers **ETL**(Extract, Transform, Load) tels que **Pentaho Data Integration** ou Talend pour extraire les données des deux bases de données, les transformer dans un format commun, puis les charger dans une nouvelle base de données. Cette approche peut nous aider à intégrer des données provenant de plusieurs bases de données. Ces outils offrent une interface "drag and drop" pour mapper les données de différentes sources et simplifier le processus d'intégration.

2. Présentation d’une solution médiateur en CustomCode codée en java

L’autre solution que nous avons mise en place est l'implémentation d’une application qui sert de lien entre la base MongoDB et PostgreSQL. Les données restent là où elles sont et l’utilisateur peut via une application médiateur choisir la base sur laquelle il veut travailler. Cette solution nous a semblé évidente pour plusieurs raisons. Premièrement le fait que les deux bases n’ont strictement rien à voir et auront surement des utilisations et des traitements différents, donc nous doutons fort qu’elles seront utilisées en parallèle dans un usage commun et tourné vers le même objectif. On peut très bien imaginer deux profils d’utilisateurs différents qui auront des objectifs différents vis à vis de la base de données souhaitée et cette approche considère que fusionner les données au même endroit n’a pas de sens et amène juste plus de complexité dans le traitement de l’information que l’utilisateur voudra faire, étant donné que les données des deux bases sont drastiquement différentes. De nombreux autres facteurs peuvent nous faire pencher faire cette solution comme :

1. Flexibilité : En utilisant une solution médiateur, on peut laisser les données dans leur base respective (PostgreSQL et MongoDB) sans avoir à les transférer ou les convertir d'une base à l'autre. Cela permet de préserver la structure et les fonctionnalités spécifiques de chaque base de données, tout en permettant d'accéder aux données de manière transparente.

2. Hétérogénéité des données : PostgreSQL et MongoDB sont deux bases de données différentes, avec des modèles de données différents. PostgreSQL est une base de données relationnelle, tandis que MongoDB est une base de données orientée document. En utilisant une solution médiateur, on peut exploiter les avantages de chaque modèle de données et accéder à des fonctionnalités spécifiques à chaque base de données.

3. Réutilisabilité du code : En développant une application commune en Java qui utilise des drivers pour communiquer avec les bases de données, on peut réutiliser le code existant et éviter la duplication. Cela simplifie la maintenance et les mises à jour, car les changements ne doivent être effectués qu'à un seul endroit.

4. Facilité d'accès aux données : Avec une solution médiateur, on peut exécuter des requêtes sur les deux bases de données via une application commune. Cela facilite l'accès aux données, car les utilisateurs n'ont pas besoin de connaître les spécificités de chaque base de données. De plus, cela permet d'effectuer des opérations de jointure et de combinaison de données provenant des deux bases de données.

5. Scalabilité : Une solution médiateur peut offrir une évolutivité en permettant l'ajout de nouvelles bases de données à intégrer facilement. Par exemple, si nous souhaitons ajouter une autre base de données NoSQL, nous pouvons étendre la solution médiateur existante sans avoir à modifier l'application commune.

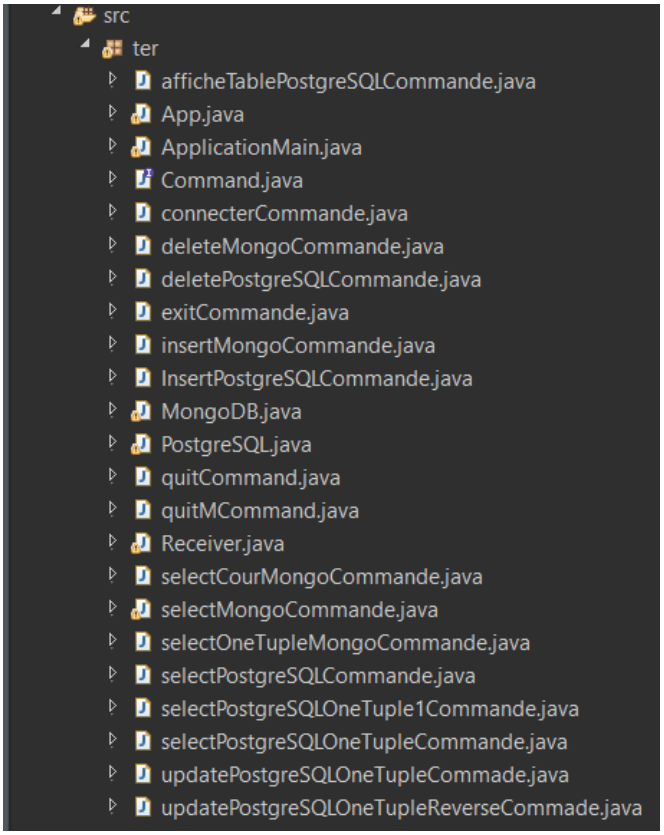
6. Coût : En évitant la nécessité de migrer toutes les données d'une base à l'autre, nous pouvons réduire les coûts associés à la conversion et à la maintenance de deux bases de données distinctes. De plus, en utilisant une application commune, nous pouvons

économiser sur les ressources nécessaires pour développer et maintenir deux applications distinctes pour chaque base de données.

En résumé, l'utilisation d'une solution médiateur avec des drivers Java pour intégrer une base de données PostgreSQL et MongoDB offre une flexibilité, une réutilisabilité du code, une facilité d'accès aux données, une scalabilité et une réduction des coûts. Cela permet de tirer parti des avantages spécifiques de chaque base de données tout en simplifiant le développement et la maintenance de l'application globale.

Passons à la présentation de l’application.

Voilà comment se décrit l’architecture et l’implémentation des classes :



La classe Application est la classe qui lance le tout en instanciant un nouvel objet de type App :

```java
package ter;

import java.sql.SQLException;

public class ApplicationMain {

    public static void main(String args[]) throws SQLException {
        App app = new App();
    }

}
```

Voici la classe App :

```java
public class App {

    private static HashMap<String, Command> commands = new HashMap<>();
    private static Receiver receiver = new Receiver();

    public App() throws SQLException {
        String newLine = System.getProperty("line.separator");
        commands.put("PostgreSQL", new connecterCommande(receiver,"PostgreSQL"));
        commands.put("MongoDB", new connecterCommande(receiver,"MongoDB"));
        commands.put("EXIT", new exitCommande(receiver));
        System.out.println("***Integration Database Application by OUMEZZAOUCHE Mohamed***");
        System.out.println("Veuillez vous connecter a l'une des bases en saissisant PostgreSQL ou MongoDB");
        while (true) {
            System.out.print(">> ");
            Scanner Commande = new Scanner(System.in);
            String nextCommande = Commande.nextLine();
            if(commands.containsKey(nextCommande)) {
                commands.get(nextCommande).execute();
            }else {
                System.out.print("Commande inconnue");
            }
        }

    }

}
```

L'implémentation globale utilisera le design pattern Commande. Via une hashmap, nous stockons les chaînes de caractères et les objets associés qui serviront à lancer la commande que l'utilisateur écrira dans la ligne de commande. L'utilisateur devra alors rentrer "PostgreSQL" ou "MongoDB" pour se connecter à la base qu'il veut. La possibilité de quitter l'application existe via la commande "EXIT". Que l'utilisateur se connecte à MongoDB ou PostgreSQL, une instance de la classe connecterCommande sera créée avec la bonne base associée (ici on a mis une chaine de caractère).

```java
package ter;

import java.sql.SQLException;

public class connecterCommande implements Command {

    private Receiver receiver;
    private String string;

    public connecterCommande(Receiver receiver, String string) {
        this.receiver = receiver;
        this.string = string;
    }

    @Override
    public void execute() throws SQLException {
        receiver.connect(string);

    }

}
```

Toutes les actions que l'utilisateur pourra écrire au clavier auront une classe associée comme celle ci, on aura besoin d'un objet de type receiver qui comportera toutes les implémentations des différentes actions que peut faire l'utilisateur. Pour connecterCommande on a pas aussi un String qui représente la base.

Je ne montrerai pas les autres classes pour les autres actions étant donné qu'elles ont toutes le même schéma, c'est à dire qu'elle ont toutes comme attribut le receiver ainsi que des objets qui seront primordiales pour le traitement des actions dans la classe Receiver qui contient l'implémentation des actions.

Voici la méthode connect dans la classe Receiver.

```java
public void connect(String string) throws SQLException {
    if(string == "PostgreSQL") {
        try {
            //étape 1: charger la classe de driver
            Class.forName("org.postgresql.Driver");
            //étape 2: créer l'objet de connexion
            Connection conn = DriverManager.getConnection(
            "jdbc:postgresql://pg.adam.uvsq.fr/piratage","piratage","f0f89cee55");
            PostgreSQL p = new PostgreSQL(conn);
       }catch(Exception e){
            System.out.println(e);
        }

    }else if(string == "MongoDB"){
        try {
            MongoClient mongoClient = MongoClients.create("mongodb+srv://Zeyphax:zeyphax00@bd-decisions.eok3p.mongodb.net/mshclemi");
            MongoDB m = new MongoDB(mongoClient);
        }catch (MongoException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

On se connecte à l'une ou l'autre base de données en fonction du string passé en argument. Pour la connexion à la base postgreSQL on charge le driver via Class.forName, on établit ensuite la connexion via DriverManager.getConnection en faisant attention à inclure le nom de la base et le mot de passe d'accès. Pour la connexion à la base MongoDB on utilise MongoClients.create à laquelle on passe le lien de connexion à la base et à la bonne collection. Après s'être connecté à l'une des deux bases on crée les instances associées.

```java
public class PostgreSQL {

    private Connection conn;
    private static HashMap<String, Command> commands = new HashMap<>();
    private static Receiver receiver = new Receiver();

    public PostgreSQL(Connection conn) throws SQLException {
        this.conn = conn;
        Statement stmt = conn.createStatement();
        commands.put("SELECT * FROM bsa", new selectPostgreSQLCommande(receiver,stmt));
        commands.put("SELECT * FROM bsa WHERE country = 'Australia' AND year = 1995", new selectPostgreSQLOneTupleCommande(receiver,stmt));
        commands.put("SELECT * FROM bsa WHERE country = 'Japon' AND year = 1995", new selectPostgreSQLOneTuple1Commande(receiver,stmt));
        commands.put("UPDATE bsa SET index = 30 WHERE country = 'Australia' AND year = 1995", new updatePostgreSQLOneTupleCommade(receiver,stmt));
        commands.put("UPDATE bsa SET index = 35 WHERE country = 'Australia' AND year = 1995", new updatePostgreSQLOneTupleReverseCommade(receiver,stmt));
        commands.put("d", new afficheTablePostgreSQLCommande(receiver,stmt));
        commands.put("DELETE FROM bsa WHERE country = 'Japon' AND year = 1995", new deletePostgreSQLCommande(receiver,stmt));
        commands.put("INSERT INTO bsa (region_id,country_id,country,year,value,index) VALUES (1,1,'Japon',1995,90,30)", new InsertPostgreSQLCommande(receiver,stmt));
        commands.put("QUIT", new quitCommand(receiver,conn));
        System.out.println("***Bienvenue sur la base Piratage***");
        System.out.println("Tapez SELECT * FROM [nom_de_table] pour faire un select sur une table");
        System.out.println("Tapez SELECT * FROM [nom_de_table] WHERE [attribut] = valeur AND [attribut] = valeur pour faire un select d'un tuple sur une table");
        System.out.println("Tapez UPDATE [nom_de_table] SET [attribut] = valeur WHERE [attribut] = valeur AND [attribut] = valeur pour faire un update d'un tuple sur une table");
        System.out.println("Tapez INSERT INTO [nom_de_table] (...) VALUES (...) pour insérer dans une table");
        System.out.println("Tapez DELETE FROM [nom_de_table] WHERE [attribut] = valeur AND [attribut] = valeur pour supprimer un tuple d'une table");
        System.out.println("Tapez d pour afficher le nom des tables");
        System.out.println("Tapez QUIT pour fermer la connexion et revenir a l'acceuil");
        while (true) {
            System.out.print(">> ");
            Scanner Commande = new Scanner(System.in);
            String nextCommande = Commande.nextLine();
            if(commands.containsKey(nextCommande)) {
                commands.get(nextCommande).execute();
            }else {
                System.out.print("Commande inconnue");
            }
```

```java
public class MongoDB {

    private MongoClient mongoClient;
    private static HashMap<String, Command> commands = new HashMap<>();
    private static Receiver receiver = new Receiver();

    public MongoDB(MongoClient mongoClient) throws SQLException {
        System.out.println("***Bienvenue sur la base Judilibre***");
        System.out.println("Tapez SELECT * pour faire un select de la base");
        System.out.println("Tapez SELECT id pour selectionner un id particulier");
        System.out.println("Tapez cour");
        System.out.println("Tapez delete id pour supprimer un tuple particulier");
        System.out.println("Tapez insert pour ré insérer le tuple supprimé");
        System.out.println("Tapez QUIT pour fermer la connexion et revenir à l'acceuil");
        this.mongoClient = mongoClient;
        MongoDatabase database = mongoClient.getDatabase("mshclemi");
        MongoCollection<Document> collection = database.getCollection("decision");
        commands.put("SELECT *", new selectMongoCommande(receiver,collection));
        commands.put("SELECT id", new selectOneTupleMongoCommande(receiver,collection));
        commands.put("cour", new selectCourMongoCommande(receiver,collection));
        commands.put("DELETE id", new deleteMongoCommande(receiver,collection));
        commands.put("INSERT id", new insertMongoCommande(receiver,collection));
        commands.put("QUIT", new quitMCommand(this.mongoClient,receiver));
        while (true) {
            System.out.print(">> ");
            Scanner Commande = new Scanner(System.in);
            String nextCommande = Commande.nextLine();
            if(commands.containsKey(nextCommande)) {
                commands.get(nextCommande).execute();
            }else {
                System.out.print("Commande inconnue");
            }
        }
    }
```

Les deux classes possèdent des Hashmaps pour stocker les actions que l'utilisateur pourra faire sur les deux bases. On a essayé d'implémenter toutes les actions CRUD possibles pour qu'un utilisateur ait vraiment l'impression d'utiliser un SGBD.

Sur la base PostgreSQL, un utilisateur peut par exemple select toute la table bsa.

```java
public void selectPostgreSQL(Statement stmt) throws SQLException {
    ResultSet res = stmt.executeQuery("SELECT * FROM bsa");
    while(res.next())
        System.out.println(res.getInt(1)+"  "+res.getString(2)
        +"  "+res.getString(3) + " " +res.getString(4)+ " " +  res.getString(5) + " " + res.getString(6));
}
```

On stocke le résultat de la requête dans un ResultSet que l'on pourra parcourir pour y afficher chaque tuple attribut par attribut.

L'utilisateur peut aussi afficher qu'un seul tuple de cette base :

```java
public void selectPostgreSQLOneTuple(Statement stmt) throws SQLException {
    ResultSet res = stmt.executeQuery("SELECT * FROM bsa WHERE country = 'Australia' AND year = 1995");
    while(res.next()) {
        System.out.println(res.getInt(1)+"  "+res.getString(2)
        +"  "+res.getString(3) + " " +res.getString(4)+ " " +  res.getString(5) + " " + res.getString(6));
    }
```

Il peut aussi faire un update :

```
public void updatePostgreSQLOneTuple(Statement stmt) throws SQLException {
     int numRowsAffected = stmt.executeUpdate("UPDATE bsa SET index = 30 WHERE country = 'Australia' AND year = 1995");
        if (numRowsAffected > 0) {
            System.out.println(numRowsAffected + " lignes ont ete modifiees !");
        } else {
            System.out.println("Aucune ligne n'a ete modifiee.");
        }

}
```

La variable numRowsAffected représente le nombre de tuples affectés par la requête, cela permet de savoir si la requête de l'utilisateur a eu de l'impact ou pas.

L'utilisateur peut aussi afficher le nom de toutes les tables de la base :

```
public void afficheTablePostgreSQL(Statement stmt) throws SQLException {
    ResultSet res = stmt.executeQuery("SELECT tablename FROM pg_tables WHERE schemaname = 'public'");
    System.out.println("tableName");
    System.out.println("..........");
    while(res.next()) {
        String tableName = res.getString(1);
        System.out.println(tableName);
    }

}
```

La requête dans le ExecuteQuery va chercher tous les noms de tables auxquels l'utilisateur peut accéder.

L'utilisateur peut aussi insérer un tuple:

```
public void InsertPostgreSQLCommande(Statement stmt) throws SQLException {
    String sql = "INSERT INTO bsa (region_id,country_id,country,year,value,index) VALUES (1,1,'Japon',1995,90,30)";

    int numRowsAffected = stmt.executeUpdate(sql);

    if (numRowsAffected > 0) {
        System.out.println(numRowsAffected + " ligne(s) ont ete inseree(s) !");
    } else {
        System.out.println("Aucune ligne n'a ete inseree.");
    }

}
```

Il peut savoir si le tuple a été effectivement crée ou non grâce à numRowsAffected.

L'utilisateur peut très bien fermer la connexion pour revenir à l'interface de choix de connexion entre MongoDB et PostgreSQL.

```
public void quit(Connection conn) throws SQLException {
    conn.close();
    App app = new App();

}
```

Passons aux méthodes de la base MongoDB.

L'utilisateur peut faire un select de la collection entière :

```
public void selectMongo(MongoCollection<Document> collection) throws SQLException {
    int pageSize = 20;
    int pageNumber = 1;

    Scanner scanner = new Scanner(System.in);

    while (true) {
        MongoCursor<Document> cursor = collection.find().skip((pageNumber - 1) * pageSize).limit(pageSize).iterator();

        while (cursor.hasNext()) {
            System.out.println(cursor.next().toJson());
        }

        cursor.close();
        System.out.println(newLine);
        System.out.println("**Appuie sur entrée pour afficher d'autres tuples ou sur e pour quitter l'affichage**");

        if (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (line.equalsIgnoreCase("e")) {
                break;
            }
        }

        pageNumber++;
    }
}
```

Etant donné que la collection contient beaucoup de documents, on a choisi d'implémenter ce select en affichant les tuples petit à petit. Si l'utilisateur veut afficher les tuples suivants il appuie sur la touche entrée, s' il veut quitter l'affichage il appuie sur la touche e. On utilise un cursor sur lequel on va appliquer la méthode skip pour ne pas avoir les mêmes tuples à chaque affichage, ainsi la nouveauté des documents est garantie.

L'utilisateur peut afficher un document en fonction de l'id du document demandé :

```
public void selectOneTupleMongo(MongoCollection<Document> collection) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Veuillez entrer un id : ");
    String idToSelect = scanner.nextLine();
    Document filter = new Document("_id", idToSelect);
    MongoCursor<Document> cursor = collection.find(filter).iterator();
    while (cursor.hasNext()) {
        System.out.println(cursor.next().toJson());
    }
}
```

Le scanner attend le string que l'utilisateur rentrera et le stocke dans un filtre. Ce filtre servira pour limiter la méthode find aux documents qui contiennent le même id rentré par l'utilisateur.

La collection contient des décisions de cours d'appels et de cours d'assise. Après étude de la collection, nous avons remarqué que les collections ayant le champ "source" égal à jurica étaient des décisions de cour d'appel tandis que les champs "source" égaux à dila ou jurinet étaient des décisions de cours d'assises. Ainsi en spécifiant l'un des trois champs, l'utilisateur peut aussi afficher les documents associées aux décisions qu'il souhaite.

```java
public void selectCourMongo(MongoCollection<Document> collection) {
    int pageSize = 20;
    int pageNumber = 1;
    Scanner scanner = new Scanner(System.in);
    System.out.print("Veuillez entrer jurica pour cour d'appel ou jurinet/dila pour cour d'assise : ");
    String cour = scanner.nextLine();
    Document query = new Document("source", new Document("$eq",cour));
    while (true) {
        MongoCursor<Document> cursor = collection.find(query).skip((pageNumber - 1) * pageSize).limit(pageSize).iterator();

        while (cursor.hasNext()) {
            System.out.println(cursor.next().toJson());
        }

        cursor.close();
        System.out.println(newLine);
        System.out.println("**Appuie sur entrée pour afficher d'autres tuples ou sur e pour quitter l'affichage**");

        if (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (line.equalsIgnoreCase("e")) {
                break;
            }
        }

        pageNumber++;
    }

}
```

Étant donné que l'affichage contient beaucoup de documents, nous utilisons encore cette notion de skip pour afficher petit à petit les documents et quitter quand l'utilisateur en a assez vu.

L'utilisateur peut aussi supprimer un document.

```java
public void deleteMongo(MongoCollection<Document> collection) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Veuillez entrer un id : ");
    String idToDelete = scanner.nextLine();
    Document filter = new Document("_id", idToDelete);
    collection.deleteOne(filter);
    System.out.println("Tuple supprime avec succes.");

}
```

On utilise encore un filtre sur lequel on va appliquer la méthode deleteOne pour la suppression.

L’utilisateur peut aussi insérer un document.

```java
public void insertMongo(MongoCollection<Document> collection) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Veuillez entrer un id : ");
    String id = scanner.nextLine();
    // Création du document à insérer
    Document document = new Document("_id", id)
            .append("contested", null)
            .append("numbers", new Document())
            .append("publication", new Document())
            .append("themes", new Document())
            .append("timeline", null)
            .append("visa", new Document())
            .append("rapprochements", new Document())
            .append("source", "jurinet")
            .append("text", "N° S 16-86.881 F-P+B\n\nN° 113\n\nFAR\n27 FÉVRIER 2018\n\n\nCASSATION PARTIELL…")
            .append("chamber", "Chambre criminelle")
            .append("decision_date", "2018-02-27T00:00:00.000+00:00")
            .append("ecli", "ECLI:FR:CCASS:2018:CR00113")
            .append("jurisdiction", "Cour de cassation")
            .append("number", "16-86.881")
            .append("solution", "Cassation")
            .append("type", "Autre")
            .append("formation", "Formation restreinte hors RNSM/NA")
            .append("summary", "La cour d'appel saisie d'une demande indemnitaire pour des faits de co…")
            .append("update_date", "2019-03-07T00:00:00.000+00:00")
            .append("forward", null)
            .append("__v", 0);
 // Insertion du document dans la collection
    collection.insertOne(document);
    System.out.println("Tuple insere avec succes.");

}
```

On peut alors entrer l’id que l’on souhaite et le document associé va alors être crée et inséré.

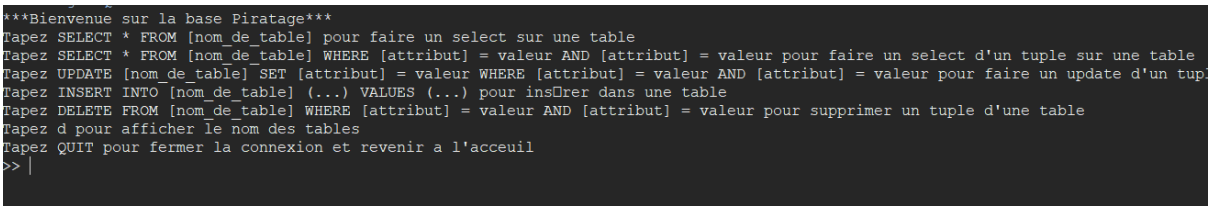
Nous sommes bien d’accord que l’utilisation de ces deux bases est très limitée avec notre application. En effet pour la base PostgreSQL on peut juste select la table bsa, insérer, supprimer et modifier un tuple particulier et ça c’est pareil pour MongoDB via l’insertion d’un document, on ne peut pas choisir les champs. Nous sommes conscients que pour rendre l’application complète il faudrait implémenter des actions et des méthodes pour chaque table et chaque possibilité de changements de tuples, ou alors écrire des méthodes aux instructions pré compilées que l’on pourra modifier en changeant les valeurs et les champs qui nous intéressent. Via notre proposition, nous voulons montrer que nous savons gérer les différentes actions CRUD, le manque de temps ne nous permet malheureusement pas d’implémenter l’étendue complète des traitements normalement possibles sur des SGBDs classiques.

L’application se présente comme tel :

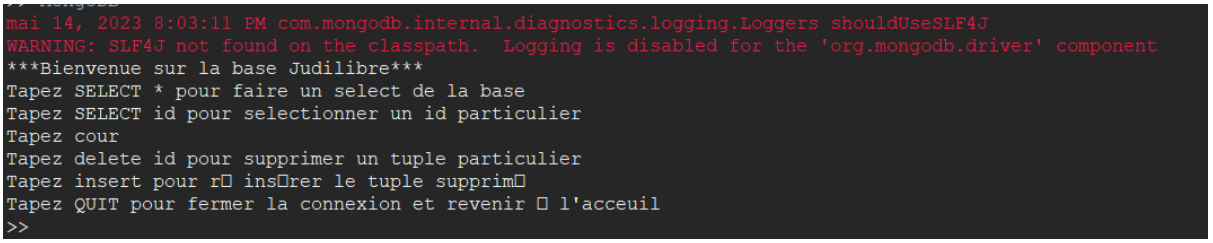
```
ApplicationMain (2) [Java Application] C:\Program Files\Java\jdk-18\bin\javaw.exe (14 mai 2023 à 19:59:10)
***Integration Database Application by OUMEZZAOUCHE Mohamed***
Veuillez vous connecter a l'une des bases en saissisant PostgreSQL ou MongoDB
>> PostgreSQL
```

Ici je choisis par exemple de me connecter à la base PostgreSQL donc je rentre le “PostgreSQL” et je clique sur entrée.

Il faut bien sûr faire attention à bien être connecté au VPN avant la connexion sinon ça ne marche pas et la connexion n'est pas trouvée.



L'interface MongoDB :



Attention à bien se déconnecter du VPN sinon la connexion est impossible.

Conclusion :

PostgreSQL et MongoDB ont des différences significatives dans leur approche de stockage des données, leur langage de requête et leur modèle de données. Pour notre première solution proposée, c'est à dire, la fusion des deux base de données sur PostgreSQL pourrait être un bon choix car les données de la base 'judilibre' restent des données assez structuré même s'il y a des valeurs manquantes. En effet, la plupart des documents suivent la même structure avec les même types de données. Alors, en basculant toutes les données sur Postgresql, ce sera plus facile de générer des statistiques sur les deux bases et nous n'aurons pas besoin de maintenir 2 bases séparés.

MongoDB reste quand même un bon choix pour les applications nécessitant une grande évolutivité, de la rapidité à la récupération de données et un stockage flexible de données semi-structurées ou non structurées. Cependant il était préférable de convertir des données semi structurées en donnés structurées que le contraire.

Il est important de considérer la deuxième solution, ou une solution médiateur car cela impliquerait qu'on aurait pas à faire de transformation et donc il n'y aura pas de risques de perte de données.