**Projet de fin d’études**

Victor BONNIN – Joris CASADAVANT

*BLOC 2 – Concevoir et déployer une solution de traitement des données*

Une architecture de traitement distribué de données :

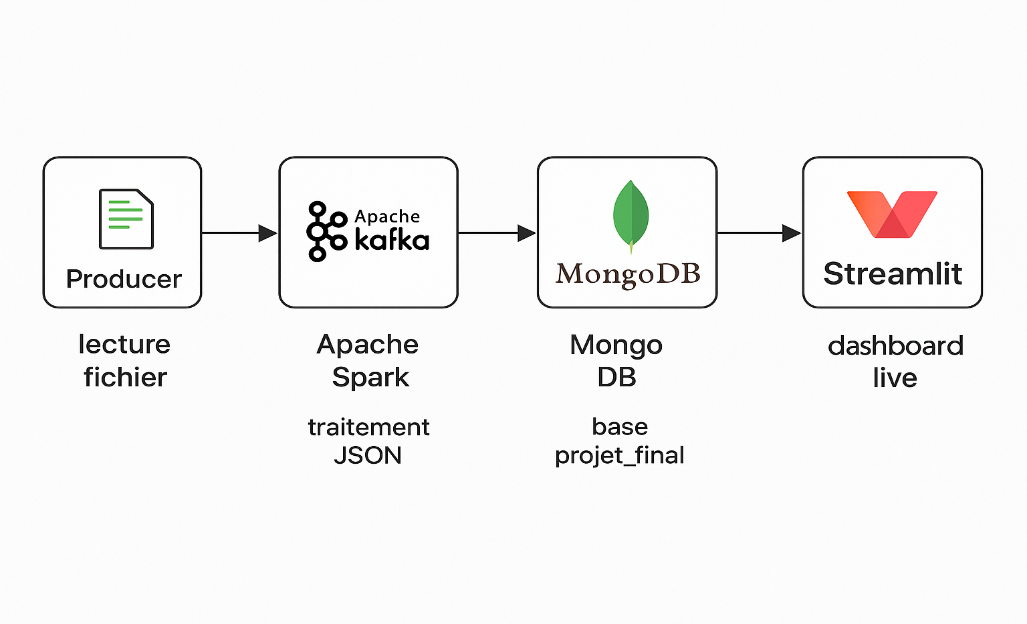
Afin de construire une architecture distribuée de traitement de données massives, nous avons retenu l'association Apache Kafka, Apache Spark Structured Streaming et MongoDB.

Kafka a été choisi pour sa capacité à gérer efficacement de larges flux de données en continu grâce à son architecture distribuée native, garantissant la résilience et la scalabilité horizontale des traitements.

Spark Structured Streaming a été sélectionné pour son aptitude à traiter en micro-batch ou en temps quasi réel de grands volumes de données en offrant une faible latence et une optimisation automatique des ressources, tout en garantissant la tolérance aux pannes par ses mécanismes de checkpoint et de reprise.

Enfin, MongoDB a été intégré comme solution de persistance grâce à son modèle NoSQL orienté documents, particulièrement adapté pour stocker des données semi-structurées provenant du streaming, et offrant une grande souplesse pour des opérations d'analytique rapide sans schéma rigide.

Cette combinaison de technologies permet ainsi d'assurer l'ingestion, le traitement en temps réel et le stockage scalable des données tout en préparant une future exploitation analytique efficace et modulaire.



Nous allons maintenant voire plus en détail le fonctionnement de chaque composants de l’architecture.

**1. Producer (Producteur de données)**

Le producer lit les données brutes depuis les fichiers textes, les structures au format JSON, et les envoie dans le topic Kafka dédié.

C’est la porte d’entrée du système qui simule un flux de données continu en micro-batchs.

**2. Kafka (Système de messagerie distribuée)**

Kafka joue le rôle d’intermédiaire de transport : il reçoit les messages du producteur et les stocke temporairement dans un topic (donnees\_topic).

Cela permet de découpler l’ingestion de données de leur traitement, garantissant une haute disponibilité et une résilience à la panne.

**Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.**

**3. Spark Structured Streaming (Moteur de traitement temps réel)**

Spark consomme les messages du topic Kafka en flux continu.

Il parse les données JSON, corrige dynamiquement les types de données (ex: valeur foncière convertie en float), remplace les valeurs nulles, et prépare les données pour un stockage structuré.

Grâce au micro-batching, Spark garantit un traitement rapide et scalable.

Une image contenant texte, capture d’écran, art

Le contenu généré par l’IA peut être incorrect.

**4. MongoDB (Base de données NoSQL)**

Les données transformées sont stockées dans MongoDB, dans une base appelée projet\_final et une collection streaming\_resultats.

MongoDB est adapté aux flux semi-structurés car il permet d’enregistrer sans schéma rigide, tout en assurant une bonne rapidité d'accès aux données pour l'analyse.

**Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.**

**5. Streamlit (Interface utilisateur de visualisation)**

Streamlit propose une interface graphique simple et interactive pour consulter en temps réel les données présentes dans MongoDB.

L'application permet d’afficher un tableau complet des données ingérées, de rafraîchir manuellement la vue, et de consulter des statistiques globales sur les volumes de données traitées.

Pour conclure :

L'architecture mise en œuvre est nativement évolutive et scalable.

Chaque composant est indépendant et conçu pour fonctionner en environnement distribué :

* Kafka permet de gérer un nombre illimité de producteurs et de consommateurs en ajoutant simplement des partitions et des brokers supplémentaires.
* Spark Structured Streaming peut s'étendre horizontalement en ajoutant des workers pour paralléliser le traitement des données sans modifier la logique applicative.
* MongoDB, en mode cluster, permet d'absorber des volumes croissants de données tout en maintenant des temps d'accès rapides grâce à la réplication et au sharding.
* Streamlit peut être facilement mis à l'échelle via un déploiement en conteneurs pour supporter plus d'utilisateurs si nécessaire.

**6. PostgreSQL (Stockage de données structuré)**

Nous avons utilisé PostgreSQL comme base de données relationnelle pour notre projet. En effet, cette BDD nous permet de stocker des données transformées dans une structure de données optimisée pour l’analyse multidimensionnelle. Contrairement à MongoDB que nous avons utilisé précédemment pour le streaming, PostgreSQL est ici utilisé pour sa capacité à stocker les données de manière structurée. Cet environnement va nous permettre de pouvoir générer des scripts SQL pour, dans la finalité, utiliser des outils de visualisation ou ouvrir le projet vers d’autre analyses plus complexes (Machine Learning, models de prédictions, …).

Ainsi, nos données issues du streaming de notre bronze\_layer vont pouvoir directement arriver dans des tables structurées, desquelles nous allons ensuite pouvoir générer différents cubes.

L’utilisation de PostgreSQL s’explique aussi par sa facilité de mise en place, sa richesse fonctionnelle assurant la prise en charge de fonctions d’agrégations, création de vues, utilisation de requêtes complexes, modélisation multidimensionnelle. Aussi, PostgreSQL est très largement rependu dans le monde la data, permettant ainsi la compatibilité avec de nombreux d’ETL et de BI, dont nous aurons très probablement besoin dans les futures parties du projet (Tableau, PowerBI, …).

Pour finir, la richesse de nos datasets nous inquiétait beaucoup. PostgreSQl nous est alors parvenu comme une solution très fonctionnelle, car cet outil est capable de gérer les millions de lignes.

Une image contenant texte, capture d’écran, menu, conception

Le contenu généré par l’IA peut être incorrect.

*Ingestion des données dans notre base PostgreSQL*

Une image contenant capture d’écran, Logiciel multimédia, logiciel, Logiciel de graphisme

Le contenu généré par l’IA peut être incorrect.

*Illustration des données dans notre base (DBeaver)*

**Environnement de test :**

GitHub Actions (Intégration continue et automatisation des tests) :

Afin d'assurer la robustesse et la maintenabilité du pipeline de traitement, nous avons intégré un processus d’intégration continue (CI) avec GitHub Actions. Cette plateforme permet de déclencher automatiquement des tests à chaque mise à jour du code, garantissant ainsi la non-régression du projet.

Plutôt que d’exécuter tout l’environnement distribué (Kafka, Spark, Mongo), le choix a été fait de tester de manière isolée certaines briques fonctionnelles à travers des scripts de test unitaires. Ces tests, écrits en Python avec pytest, vérifient notamment l'importabilité des modules, la structure du schéma des données JSON, ou la transformation correcte de types. Cela permet une vérification rapide et fiable à chaque modification du code source, sans devoir reconstruire l’architecture complète.

Ce choix s’inscrit dans une bonne pratique d’ingénierie logicielle : tester les modules en amont, valider la cohérence du projet, et préparer une intégration continue plus large si nécessaire (tests d’intégration ou déploiement sur un environnement de staging).

Fichiers txt souples et réduits :

Les fichiers .txt utilisés pour tester le pipeline intégral ont été réduits au maximum, sont légers et structurés comme les fichiers globaux. Ainsi, ils permettent de simuler un flux de donnée réaliste tout en facilitant le débogage et l’analyse des traitements.

Nous pouvons grâce à cela faire des tests rapides et bien simuler le comportement de tout notre pipeline sans avoir à perdre du temps de génération des fichiers. Ce choix nous permet donc d’être flexible dans notre manière de travailler, et de pouvoir jouer avec les différentes étapes du projet tout en gardant un œil sur des données bien précises.

Cette architecture respecte les principes modernes du Big Data : décentralisation, résilience, élasticité et facilité de maintenance.

Elle permet de s'adapter dynamiquement à l'augmentation des volumes, des vitesses d’ingestion et des besoins métiers, tout en minimisant les coûts d'évolution.

*BLOC 1 – Concevoir et développer une architecture de stockage de données*

*Mettre en place une base de données relationnelle :*

Maintenant que nous avons des données qui sont injectées dans une base relationnelle, il va falloir les ordonnancer. Pour se faire, nous allons faire tourner une succession de scripts SQL qui vont nous permettre de créer notre base de données finale, notre gold\_layer.

**1. Modélisation de la base gold\_layer**

Après avoir structuré nos données dans une table unique dans la *Silver Layer* (appelée silver\_resultats), nous avons souhaité aller plus loin dans la modélisation pour répondre aux exigences d’un stockage relationnel propre et évolutif. Cela nous a permis de valider les compétences du bloc 1 en construisant un vrai modèle relationnel basé sur des tables normalisées et des vues métiers prêtes à l’analyse.

* Étape 1 – Analyse et identification des entités

Nous avons commencé par identifier les redondances dans notre table silver\_resultats. Certaines colonnes comme commune, type\_local ou nature\_mutation contiennent des valeurs répétées qui auraient tout intérêt à être isolées dans des tables de référence. C’est ce que nous avons fait en créant trois nouvelles tables :

* communes (avec un identifiant unique pour chaque nom de commune),
* types\_locaux (types de biens immobiliers),
* Une image contenant texte, nombre, capture d’écran, conception

  Le contenu généré par l’IA peut être incorrect.natures\_mutation (nature de la transaction, comme vente ou échange).

*diagramme effectué sur dbdiagram.io*

* Étape 2 – Construction de la table transactions

Nous avons ensuite créé une table centrale transactions, qui contient les informations propres à chaque opération immobilière (valeur foncière, surface, date, etc.). Cette table référence les tables précédentes grâce à des clés étrangères (id\_commune, id\_type, id\_nature), ce qui permet d’éviter la redondance et garantit l’intégrité des données.

Un point important ici a été la gestion de l’identifiant unique id\_unique, généré en amont dans la Silver Layer. Comme nous voulions le réutiliser directement dans la table transactions (qui utilise le type UUID), nous avons dû adapter l’insertion en veillant à respecter les types de données attendus.

**2. Création des tables dans Dbeaver**

Voici les commandes à exécuter une par une :

Table des communes, avec un Id qui sera connecté à notre table principale (transactions)

-- Table des communes

CREATE TABLE IF NOT EXISTS communes (

id\_commune SERIAL PRIMARY KEY,

nom\_commune TEXT UNIQUE NOT NULL

);

Table des locaux (infrastructures), avec un Id qui sera connecté à notre table principale (transactions)

-- Table des types de locaux

CREATE TABLE IF NOT EXISTS types\_locaux (

id\_type SERIAL PRIMARY KEY,

type\_local TEXT UNIQUE NOT NULL

);

Table des types de transactions, avec un Id qui sera connecté à notre table principale (transactions)

-- Table des natures de mutation

CREATE TABLE IF NOT EXISTS natures\_mutation (

id\_nature SERIAL PRIMARY KEY,

nature\_mutation TEXT UNIQUE NOT NULL

);

Table des transactions, qui sera notre table principale.

-- Table des transactions (Gold Layer principale)

CREATE TABLE IF NOT EXISTS transactions (

id\_transaction UUID PRIMARY KEY,

id\_unique VARCHAR(255),

code\_postal VARCHAR(10),

date\_mutation DATE,

fichier\_source TEXT,

part\_date DATE,

valeur\_fonciere FLOAT,

surface\_reelle\_bati FLOAT,

nombre\_pieces\_principales INT,

id\_commune INT REFERENCES communes(id\_commune),

id\_type INT REFERENCES types\_locaux(id\_type),

id\_nature INT REFERENCES natures\_mutation(id\_nature)

);

**2. Injection des données**

Maintenant que nos tables ont bien été créées, il faut y injecter les données. Nous pouvons, pour faire cela, utiliser les données de notre table ‘silver\_layer’, car nous savons que les données ont déjà été nettoyées. Ainsi, grâce a ce travail, l’injection ne se résume plus qu’en quelques lignes.

-- Insérer les communes

INSERT INTO communes (nom\_commune)

SELECT DISTINCT commune

FROM silver\_resultats

WHERE commune IS NOT NULL;

-- Insérer les types de locaux

INSERT INTO types\_locaux (type\_local)

SELECT DISTINCT type\_local

FROM silver\_resultats

WHERE type\_local IS NOT NULL;

-- Insérer les natures de mutation

INSERT INTO natures\_mutation (nature\_mutation)

SELECT DISTINCT nature\_mutation

FROM silver\_resultats

WHERE nature\_mutation IS NOT NULL;

-- Insérer les données globales de la table principale

**INSERT** **INTO** transactions (

id\_transaction,

id\_unique,

code\_postal,

date\_mutation,

fichier\_source,

part\_date,

valeur\_fonciere,

surface\_reelle\_bati,

nombre\_pieces\_principales,

id\_commune,

id\_type,

id\_nature

)

**SELECT**

gen\_random\_uuid() **AS** *id\_transaction*,

*sr*.id\_unique,

*sr*.code\_postal,

*sr*.date\_mutation,

*sr*.fichier\_source,

*sr*.part\_date,

*sr*.valeur\_fonciere,

*sr*.surface\_reelle\_bati,

*sr*.nombre\_pieces\_principales,

*c*.id\_commune,

*t*.id\_type,

*n*.id\_nature

**FROM** silver\_resultats *sr*

**JOIN** communes *c* **ON** *sr*.commune = *c*.nom\_commune

**JOIN** types\_locaux *t* **ON** *sr*.type\_local = *t*.type\_local

**JOIN** natures\_mutation *n* **ON** *sr*.nature\_mutation = *n*.nature\_mutation;

Nous pouvons, maintenant que les données ont été injectées, vérifier qu’elles sont conformes dans nos nouvelles tables :

-- Aperçu des transactions les plus récentes

SELECT \*

FROM transactions

ORDER BY date\_mutation DESC

LIMIT 10;

-- Vérifier les communes référencées dans les transactions

SELECT *t*.id\_transaction, *c*.nom\_commune, *t*.valeur\_fonciere, *t*.surface\_reelle\_bati, *nm*.nature\_mutation

FROM transactions *t*

JOIN communes *c* ON *t*.id\_commune = *c*.id\_commune

join natures\_mutation *nm* on *t*.id\_nature = *nm*.id\_nature

order by *t*.valeur\_fonciere DESC;

-- Vérifier la liste complète des types de locaux

SELECT \* FROM types\_locaux ORDER BY id\_type;

**Étape 3 – Création des vues métiers (Gold Layer)**

Pour faciliter l’analyse, nous avons créé plusieurs vues SQL. Ces vues sont pensées comme des couches de lecture métier, prêtes à être utilisées dans un outil de BI ou par un utilisateur métier :

* Une vue enrichie vue\_gold\_transactions\_detaillees qui rassemble toutes les informations d’une transaction en effectuant les jointures nécessaires.
* Une vue de statistiques par commune (vue\_gold\_stats\_par\_commune) pour suivre les volumes et la valeur moyenne des ventes.
* Une vue par type de bien immobilier (vue\_gold\_stats\_par\_type\_local) pour observer les tendances de prix selon la nature du logement.

De cette manière, en cas de forte sollicitation de la base de données, celle-ci ne sera pas saturée, et les futurs rapports BI pourront s’exécuter sans monopoliser toutes les ressources du système. Cette architecture relationnelle est donc scalable et parfaitement adaptée à un modèle d’analyse comme le nôtre.

Voici maintenant les codes des vues SQL que nous avons créés pour renforcer notre gold\_layer :

Vue principale :

CREATE OR REPLACE VIEW dl\_gold\_transactions\_detaillees AS

SELECT

    t.id\_transaction,

    t.code\_postal,

    c.nom\_commune,

    t.date\_mutation,

    t.fichier\_source,

    t.part\_date,

    t.valeur\_fonciere,

    t.surface\_reelle\_bati,

    t.nombre\_pieces\_principales,

    tl.type\_local,

    nm.nature\_mutation

FROM transactions t

JOIN communes c ON t.id\_commune = c.id\_commune

JOIN types\_locaux tl ON t.id\_type = tl.id\_type

JOIN natures\_mutation nm ON t.id\_nature = nm.id\_nature;

Vue des statistiques par commune :

CREATE OR REPLACE VIEW dl\_gold\_stats\_par\_commune AS

SELECT

    c.nom\_commune,

    COUNT(\*) AS nb\_transactions,

    ROUND(AVG(t.valeur\_fonciere)) AS valeur\_moyenne,

    ROUND(SUM(t.surface\_reelle\_bati)) AS surface\_totale

FROM transactions t

JOIN communes c ON t.id\_commune = c.id\_commune

GROUP BY c.nom\_commune

ORDER BY nb\_transactions DESC;

Vue des tendances par type de bien :

CREATE OR REPLACE VIEW vue\_gold\_stats\_par\_type\_local AS

SELECT

tl.type\_local,

COUNT(\*) AS nb\_ventes,

ROUND(AVG(t.valeur\_fonciere)) AS prix\_moyen

FROM transactions t

JOIN types\_locaux tl ON t.id\_type = tl.id\_type

GROUP BY tl.type\_local

ORDER BY prix\_moyen DESC;

**Étape 4 – Vérification et tests**

Chaque insertion et création de vue a été testée dans DBeaver, avec des scripts SQL simples de lecture (SELECT \* FROM ...). Cela nous a permis de vérifier la cohérence des jointures, l’absence de doublons, et la validité des clés étrangères. À terme, ces vues permettront de construire des tableaux de bord plus efficaces et d’alimenter notre API.

Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran

Le contenu généré par l’IA peut être incorrect.Voici un exemple de données que nous pouvons retrouver dans notre dernier niveau de DBB :

Nous pouvons bien voir ici que les données ont bien été préparées et normalisées. Chaque colonne respecte un format adapté à son usage :

* les UUIDs assurent l’unicité de chaque transaction sans ambiguïté
* les dates sont bien typées au format date
* les champs numériques (valeur foncière, surface bâtie, nombre de pièces) sont bien convertis en types numériques exploitables

Ce niveau de préparation permet d’offrir un socle de données fiable, propre et directement exploitable pour la création de rapports ou l'entraînement de modèles prédictifs.

*Mettre en place une base de données non-relationnelle :*

*C1 & 2*

Pour la partie de la base de donnée non relationnelle, nous avons comme décrit précédemment utilisé une base MongoDB. Nous avons choisis cette dernière pour sa souplesse face aux données semi-structurées issue des traitements en streaming.

L’objectif de cette dernière était de pouvoir stocker rapidement et efficacement les données JSON envoyées par Kafka, et traitées en continu par Spark. MongoDB c’est ainsi montrée comme une solution parfaitement adaptée à ce besoin. Etant une base orientée documents,  elle nous permettait ainsi de stocker au format JSON mes enregistrement envoyés sans la contrainte d’un schéma rigide, nous permettant ainsi d’avoir un flux efficace.

Cette approche nous a ainsi apporté plusieurs avantages :

* Ingestion facile et rapide des données
* Analyse des données possible et visualisation en temps direct
* Intégration avec des outils de visualisation tels que Steamlit

Ce choix d’utiliser MongoDB nous a ainsi permis de répondre aux exigences de traitements des données en quasi temps réel tout en assurant la scalabilité horizontale du projet.

Pour vérifier que nos données existent bien dans MongoDB, on pouvons vérifier en allant dans son container.

Ainsi, une fois à l’intérieur, nous pouvons coder en NoSQL pour naviguer dans notre db, et y chercher les données :

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.Connexion à notre base mongo : *‘docker exec -it partie\_1-mongodb mongosh’*

Ensuite, il faut nous connecter à notre db : *‘use projet\_final’*

Une image contenant texte, Police, capture d’écran, conception

Le contenu généré par l’IA peut être incorrect.

Et pour finir, maintenant que nous sommes connectés au bon environnement, il nous suffit d’afficher les données présentes dans la base : *‘db.streaming\_resultats.find().pretty()’*

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Nous voyons bien avec cette capture d’écran que le données sont parfaitement disponibles dans notre base MongoDB.

*C3*

L’utilisation de Redis dans notre projet nous permet d’illustrer le fonctionnement d’une base de données clé-valeurs. En effet, Redis est un moteur NoSQL en mémoire, qui stocke les données sous forme de paires : une clé unique associée à une valeur.

Nous avons donc souhaité utiliser Redis pour suivre l’état de la base MongoDB, en enregistrant les valeurs suivantes :

* Nombre total de documents (lignes) -> *stats\_total\_documents*
* La moyenne des valeurs foncières -> *stat\_moyenne\_valeur\_fonciere*
* La date de dernière mise à jour -> *last\_update\_stats*
* Les 5 communes les plus fréquentes -> *top\_communes*

Ces données sont donc, comme Redis le permet, stockées à chaque exécution du script python et consultables n’importe quand. Par exemple, nous pouvons exécuter sur un moteur Redis la commande ‘*r.get("last\_update\_stats")*’ qui va nous permettre de récupérer la dernière date de mise à jour.

Nous avons aussi décidé d’un point de vue pratique, de garder une trace de toutes les exécutions du fichier, nous permettant ainsi d’avoir les données de manière persistante dans un fichier .txt, pour assurer un suivit manuel et durable.

Dans notre container principal, nous avons donc ainsi juste à exécuter le script redis pour que ce dernier nous retourne les informations que nous avons décrit ci-dessus :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Pour finir, au besoin, nous pourrons aussi consulter les statistiques sauvegardées en dur dans un fichier txt :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

*C4*

Nous avons utilisé dans ce projet une base MongoDB orientée document. Elle stocke les données au format JSON, une structuration souple, sans schéma strict. Ce format s’adapte parfaitement avec le traitement des données issues de Spark.

Nous avons donc stocké nos données dans une base que nous avons appelée ***‘projet\_final’***, structurée selon le modèle natif de Mongo :

* Une base de données qui va regrouper l’ensemble des collections.
* Une ou plusieurs collections. Ici, nous avons créé notre collection ***‘streaming\_resultats’***, dans laquelle sera inscrit chaque flux au format JSON.
* Des documents : chaque transaction immobilière, donc chaque ligne issue des fichiers sources est un document indépendant.
* Des champs : Ces derniers sont des clés à l’intérieur des documents dans lesquels les données sont ensuite insérées.
* Un Id unique : Chaque document possède son id unique généré automatiquement.

Ainsi, notre structuration respecte parfaitement un modèle orienté documents. Elle s’appuie exclusivement sur des objets JSON. Notre bronze\_layer est ainsi parfaitement scalable, souple et adapté à l’ingestion de données en temps réel. Nous pouvons aussi, grâce à l’utilisation de cette structure, intégrer de nouvelles pipelines prenant leurs sources dans cette base.

Une image contenant texte, capture d’écran, Police, conception

Le contenu généré par l’IA peut être incorrect.

*C5*

L’objectif de la compétence C5 est de démontrer notre capacité à utiliser une base de données orientée colonne. Ces dernières sont des bases conçues pour le traitement analytique à grande échelle. Dans notre projet, nous utilisons déjà un environnement Spark. La solution la plus adaptée à notre structure serait alors d’utiliser Parquet, qui est nativement supporté par Spark.

De plus, Parquet permet de stocker de grands volumes de données, et est hautement compressé, ce qui garantit un faible besoin de stockage. Enfin, parquet se base sur des fichiers colonnaires, c’est-à-dire des fichiers qui se lisent en colonne. Grâce à cela, lors de la lecture des données via une requête, Parquet ne va lire que les colonnes dont il a besoin, réduisant considérablement le temps de lecture et de chargement.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

*Dossier/fichiers générés automatiquement par Parquet pour enregistrer les données*

Dans notre projet, nous avons décidé d’utiliser Parquet pour lire nos données depuis MongoDB, pour ensuite y faire un premier travail d’analyse :

* Affichage de lignes,
* Calculs de statistiques (moyennes, nombre total de documents, …
* Ecriture des résultats dans un nouveau fichier de logs, pour pouvoir faire une analyse comparative plus tard.

Une image contenant texte, capture d’écran

Le contenu généré par l’IA peut être incorrect.

Nous avons pu ainsi découvrir l’intérêt d’une base orientée colonne, qui permet une analyse instantanée des données, tout engardant un maximum d’efficacité grâce aux fichiers colonnaires.

Nous avons pu aussi montrer grâce a cette courte analyse l’importance de nettoyer les données que nous allons traiter ensuite.

*C6*

La partie C6 du bloc 1 concerne l’utilisation d’une base orientée graphe. Cette dernière à pour objectif de modéliser et stocker les données sous forme de nœuds et de relations. Contrairement à une base de données classique utilisant des jointures et tables, les bases de données graphes reposent sur les connexions entre les éléments. Cette approche est intéressante dans le cadre de structures complexes où tous les éléments s’entremêlent.

Pour l’utilisation d’une base orientée graphe dans notre projet, nous avons décidé d’utiliser Neo4j, une base relationnelle dynamique.

Nous avons donc développé un script qui nous permet de se connecter directement à notre base MongoDB pour y récupérer les données. Ainsi, nous allons pouvoir extraire les informations clés de chaque transactions immobilières (commune, valeur foncière, date de mutation, nature de la mutation, type de bien, code postal, etc.).

Voici donc le schéma auquel nous avons réfléchis pour pouvoir intégrer notre structure de données à une base en graphes :

(:NatureMutation)

└─[:A\_POUR\_TYPE]→ (:TypeLocal)

└─[:A\_POUR\_LIEU]→ (:Commune)

└─[:A\_ENREGISTRE]→ (:Mutation {date, valeur, id})

Nous allons donc maintenant afficher les données dans Neo4j, pour voir si notre script correspond bien avec l’architecture qu’on voulait avoir affiché. Pour se faire, il nous suffit :

* d’exécuter le fichier *neo4j\_transactions.py*
* aller sur le localhost://7474
* rentrer les identifiants de connexion :
  + user : neo4j
  + mot de passe : spark123

Une image contenant capture d’écran

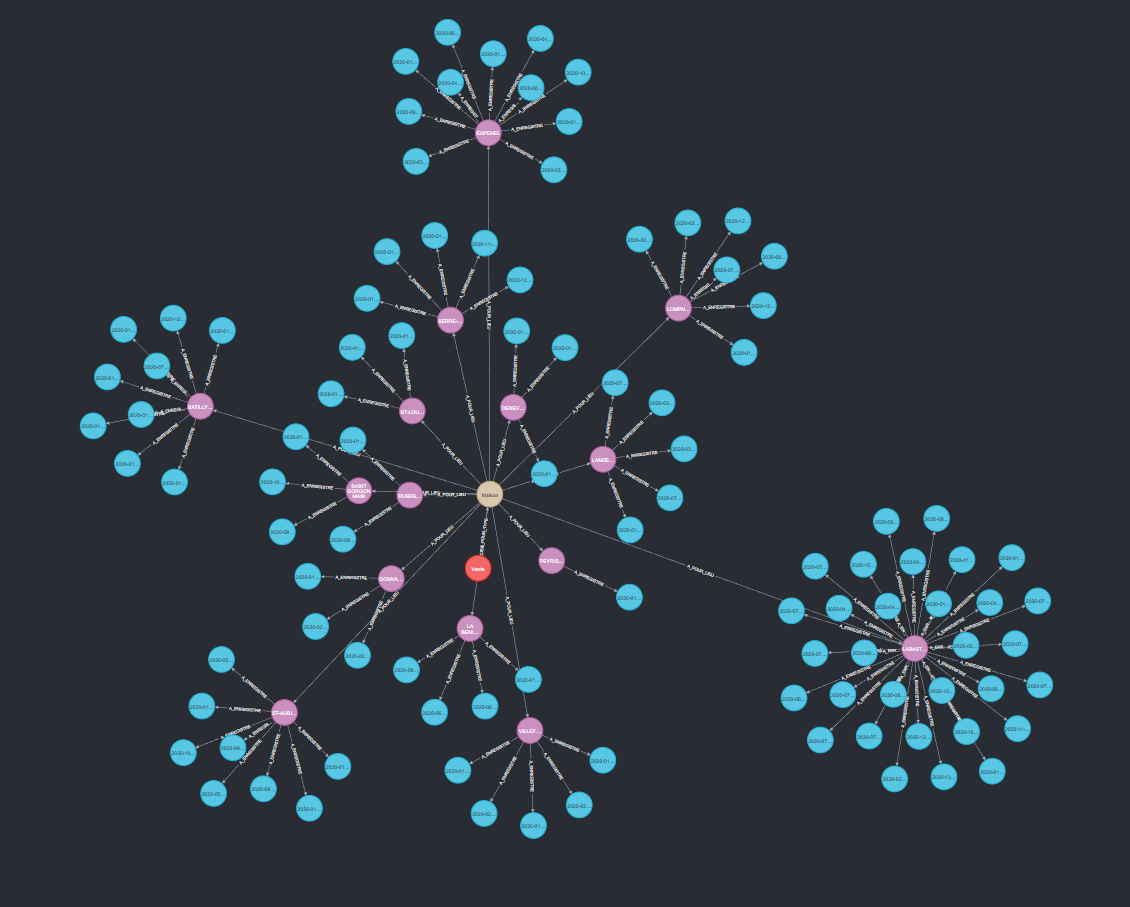
Le contenu généré par l’IA peut être incorrect.Nous arrivons sur l’interface, et nous pouvons alors exécuter un code basique pour afficher les données présentes : MATCH (n) RETURN n LIMIT 100;

On peut voir qu’il y en a beaucoup, et que ça devient vite brouillon. Nous pouvons donc améliorer notre requêtes pour afficher, par exemple, les transactions qui concernent les maisons. Ainsi, nous pouvons avoir un affichage plus simple, plus clair avec cette requête :

MATCH (n:NatureMutation)-[:A\_POUR\_TYPE]->(t:TypeLocal {nom: "Maison"})

      -[:A\_POUR\_LIEU]->(c:Commune)-[:A\_ENREGISTRE]->(m:Mutation)

RETURN n, t, c, m

LIMIT 100

Maintenant, nous voyons clairement comment notre structure se décompose. Nous pouvons aussi voir les villes les plus présentes dans notre jeu de donnée.

Pour aller plus loin, on peut, encore une fois, afficher les communes par nombre de ventes. Cela nous donnerait donc la ligne de commande suivante :

MATCH (c:Commune)-[:A\_ENREGISTRE]->(m:Mutation)

RETURN c.nom AS Commune, count(m) AS NombreMutations

ORDER BY NombreMutations DESC

LIMIT 100

Une image contenant texte, logiciel, Logiciel multimédia, Logiciel de graphisme

Le contenu généré par l’IA peut être incorrect.

De cette manière, nous voyons que nous avons des résultats cohérents avec les données que nous avons dans notre base, ce qui montre que ces dernières ont bien été importées et que notre requête renvoie les bons résultats.

Maintenant, on peut essayer de chercher à faire une ville en particulier pour vérifier que les données ont toujours bien conformes :

MATCH (c:Commune {nom: "FOISSIAT"})-[:A\_ENREGISTRE]->(m:Mutation)

RETURN m.date AS Date, m.valeur AS Valeur

ORDER BY m.date

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.LIMIT 100

On peut aussi modifier un peut la requête les détails dans le return pour garder l’affichage en graphs :

MATCH (n:NatureMutation {nom: "Vente"})-[:A\_POUR\_TYPE]->(t:TypeLocal)

MATCH (t)-[:A\_POUR\_LIEU]->(c:Commune {nom: "FOISSIAT"})

MATCH (c)-[:A\_ENREGISTRE]->(m:Mutation)

WHERE t.nom <> "Inconnu"

Une image contenant capture d’écran, texte

Le contenu généré par l’IA peut être incorrect.RETURN n, t, c, m

Nous pouvons aussi essayer de filtrer sur les transactions supérieures ou inférieures à un certain montant :

MATCH (c:Commune)-[:A\_ENREGISTRE]->(m:Mutation)

WHERE m.valeur > 1000000

RETURN c, m

LIMIT 100

Une image contenant capture d’écran

Le contenu généré par l’IA peut être incorrect.

Pour finir, on peut afficher les différents types de biens ayant engrangé la plus haute valeur totale :

MATCH (t:TypeLocal)<-[:A\_POUR\_TYPE]-(n:NatureMutation),

      (t)-[:A\_POUR\_LIEU]->(c:Commune)-[:A\_ENREGISTRE]->(m:Mutation)

RETURN t.nom AS TypeLocal, sum(m.valeur) AS ValeurTotale

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.ORDER BY ValeurTotale DESC

*Un datalake est mis en place :*

Pour répondre au besoin de stockage et de traitement de nos données massives, nous avons donc mis en place un datalake. Ce dernier nous permet d’avoir différents types de données provenant de nos différentes sources (différents fichiers txt) et nous permet de les rendre accessible dans nos divers outils d’analyse.

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.

*C1*

Notre datalake permet l’intégration de données de différents types. Nous avons :

* des données non structurées, issues de nos fichiers txt
* des données semi-structurées, issues du streaming des données. Nous stockons ces données dans notre base MongoDB.
* des données structurées, issues de notre base de données MongoDB et envoyées vers notre base PostgreSQL.

Ainsi, nous pouvons gérer tout type de données via notre structure.

*C2*

Notre solution est entièrement distribuée. Nous avons tous nos outils qui fonctionnent sur des conteneurs différents et dédiés (Spark, MongoDB, PostgreSQL, Redis, Neo4j, …). Spark assure ainsi le traitement et la distribution des tâches garantissant la scalabilité de notre pipeline.

La redondance de notre architecture est assurée via notre couche silver\_layer. Un prétraitement est effectué pour nettoyer les données qui sont envoyé de mongoDB vers PostgreSQL. Ainsi, l’architecture assure la redondance des données vers les différents systèmes. L’utilisation des volumes dockers garantit également la persistance des données au redémarrage de l’application (des scripts ont également été effectués pour redémarrer l’architecture partiellement ou totalement).



*C3*

Pour ce qui est de la surveillance de l’état du système, nous avons mis en place un monitoring des applications les plus critiques. De cette manière, nous pouvons avoir un suivit constant sur les données qui circulent dans notre infrastructure, et nous pouvons également, à l’avenir, faire une analyse dessus.

Exemples de fichiers que nous créons pour suivre les données :

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.

Ces fichiers sont donc stockés en dur dans notre dossier *logs/*, et nous pouvons les consulter à tout instant. Nous avons donc un suivit en temps réel du fonctionnement du système et nous pouvons voir à tout instant le moindre point de blocage.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

*Une API WEB Restfull est fonctionnelle :*

Pour répondre au besoin de ce bloc, nous avons conçu une API Web RESTful permettant de gérer l’accès à des données foncières stockées dans une base PostgreSQL. L’objectif était double : rendre ces données consultables sous forme structurée, tout en garantissant leur sécurité via un système d’authentification et de limitation de requêtes.

*C1*

Nous avons utilisé FastAPI pour concevoir cette dernière, car il permet une gestion simple des dépendances et une performance élevée. Nous avons associé cette API à notre base PostgreSQL pour le stockage des données, et utilisé SQLAlchemy comme ORM (Objet rationnel de mapping). Enfin, nous avons encapsulé toute la logique dans des conteneurs Docker pour garantir un déploiement reproductible, et assurer la sécurité fonctionnelle (conteneurisation) de notre projet global.

*C2*

Nos données retravaillées de la base PostgreSQL sont accessibles via une route GET /*resultats*, avec la possibilité de filtrer selon plusieurs critères (code\_postal, commune). Grâce à Pydantic, les réponses sont structurées, validées, et retournées au format JSON. Notre API est disponible sur cette adresse, une fois le conteneur lancé : ‘*http://localhost:8000/docs#/*’.

*C3*

Nous avons limité la logique aux besoins réels : lecture filtrée des données de notre base, création d’utilisateurs et génération de tokens. La séparation claire des fichiers (routes, sécurité, modèles) assure une architecture maintenable. Nous avons choisi une complexité simple, tout en gardant une sécurité et une fiabilité des données traitées.

*C4*

Chaque utilisateur peut se créer un compte via POST /*utilisateurs*/. Les mots de passe sont hachés avec bcrypt, ce qui assure la sécurité des ces derniers. L’identification se fait ensuite avec POST /*login*, qui renvoie un token JWT. Ce token est nécessaire pour accéder aux données protégées. Une fois le tocken renseigné pour l’authentification, l’utilisateur peut alors faire des requêtes sur les données disponible dans la route /*résultats*.

*C5*

Les routes sensibles sont protégées via @Depends(verify\_token), qui vérifie le token JWT. Cela nous assure que seulement une personne avec un compte renseigné peut accéder aux données. Nous avons aussi mis en place une ‘date de péremption’ de chaque tokens. En effet, ces derniers ont une durée de vie de 30 minutes (modifiable). Il faudra ensuite que l’utilisateur en génère un nouveau. Pour finir, nous avons également mis en place une limitation de requêtes avec SlowAPI (5 requêtes/minute). De cette manière, nous nous assurons qu’aucune surcharge de l’API n’aura lieu.

*C6*

La rapidité d’accès a été assurée via des requêtes SQL optimisées et des limites de pagination (limit(100)). L’API répond en quelques millisecondes, et le Swagger UI permet de tester sans surcharger le serveur.

*C7*

Nous avons, tout au long du développement de l’API, voulu nous assurer de l’automatisation de cette dernière. Ainsi, nous avons veillé à garantir un processus d’intégration des données automatisé, lancement de l’application automatique, une conservation des données utilisations et sécurité des données sans intervention humaine.

Voici donc comment nous avons répondu à ces besoins :

* Conteneur docker automatisé : l’API est intégrée via un environnement docker, via un docker-compose. Ainsi, à chaque démarrage du projet, l’ensemble des services est lancé et interconnecté.
* Connexion automatique à notre base PostgreSQL : au lancement de notre conteneur, la connexion à la base de donnée se fait automatiquement.
* Ecriture automatique des logs : toutes les requêtes générant des logs sont automatiquement transférées dans notre fichier de logs. Cela nous facilite la suivit de notre API.
* Sécurisation et accès contrôlé : notre application inclus donc également une connexion/ création d’utilisateur qui est entièrement automatisé par nos services. Une table utilisateur est créée dans notre base PostgreSQL pour répondre à ce besoin. Aucune intervention humaine n’est nécessaire lors de ce processus.

*C8*

Grâce à FastAPI, la documentation Swagger est générée automatiquement. Elle est disponible directement sur l’API, ce qui facilite la compréhension. Elle décrit les routes, les schémas attendus, et permet l’envoi de tokens dans l’interface. Un développeur externe peut utiliser l’API immédiatement.

Les réponses de l’API respectent également les codes HTTP standards (200, 401, 429). Les erreurs sont explicites : "token invalide", "trop de requêtes", etc. Cela facilite le débogage et l’utilisation côté client.

Nous avons aussi mis en place un fichier de logs, qui va stocker les messages générés par l’API. Ainsi, nous pouvons a tout moment récupérer ces données là pour vérifier que tout est bien fonctionnel. Cela nous permet de bien monitorer notre API.

*C9*

Nous utilisions git tout au long de notre projet. Git permet, au fur et à mesure de son utilisation, de gérer l’historique des versions de chaque fichiers du projet. Ainsi, nous nous servons de Git comme contrôleur de version de notre projet. Ce choix nous apporte plusieurs bienfaits :

* Une bonne traçabilité des modifications de l’API.
* Un travail efficace à plusieurs, tout en assurant un suivit simple et clair.
* Restauration à d’anciennes versions de l’application.

Démonstration du fonctionnement de l’API :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.Tout d’abord, il faut s’assurer que l’API tourne bien. Nous pouvons vérifier tout simplement après le lancement de l’application, en allant dans un terminal et en lançant : *docker ps*

*docker ps – Ici, le conteneur de l’API est surligné en jaune*

Maintenant qu’on voit que notre API tourne bien, on peut se rendre sur son adresse, à savoir [*http://localhost:8000/docs#/*](http://localhost:8000/docs#/). Une fois dessus, un interface comme ça se présente :

Une image contenant texte, capture d’écran, logiciel, nombre

Le contenu généré par l’IA peut être incorrect.

[*http://localhost:8000/docs#/*](http://localhost:8000/docs#/)

Nous pouvons voir les différentes routes dont nous aurons besoin :

* /utilisateurs : Nous pourrons nous créer un compte grâce à cette route.
* /login : Une fois notre compte créé, en nous connectant grâce à cette route, nous pourrons récupérer notre token, indispensable pour pouvoir faire des requêtes sur notre base de données.
* /resutlats : Route principale de notre API, qui va nous retourner des résultats en fonction des éléments que nous allons lui donner.

Pour commencer, nous devons donc nous rendre sur la route */utilisateurs*. Nous allons nous créer un compte. En cliquant sur la route, un menu va dérouler. Il faut cliquer sur ‘ Try it out’, en haut à droite de la route.

Nous devons ensuite renseigner les informations essentielles de notre compte, à savoir le nom, l’adresse mail et le mot de passe. Les informations sont envoyées en JSON sur l’API, et les données seront ensuite cryptées (pour les sensibles) via une méthode bcrypt. Cette dernière fait un hachage du mot de passe, qui le transforme ainsi en chaine de caractère. Chaque chaine de caractère est unique, même si deux utilisateurs ont le même mot de passe.

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.

Après cela, il faut exécuter le script json, et normalement, une message avec un ’response 200’ devrait apparaitre.

Une image contenant texte, capture d’écran, Logiciel multimédia, logiciel

Le contenu généré par l’IA peut être incorrect.

Une image contenant Logiciel multimédia, logiciel, texte, Logiciel de graphisme

Le contenu généré par l’IA peut être incorrect.Nous pouvons aussi aller vérifier dans la base de données postgre SQL que notre utilisateur à bien été créé.

Maintenant que l’utilisateur existe bien, il faut se connecter avec ses identifiants pour récupérer le token, qui nous permettra de faire des requêtes sur notre API.

Ainsi, nous pouvons aller dans la route */login*, cliquer sur ‘Try it out’, et renseigner les valeurs suivantes :

grant\_type : password

username : mettre votre email (test@email.com)

password : mettre votre mot\_de\_passe (testmdp)

scope : laisser/mettre vide

client\_id : laisser/mettre vide

client\_secret : laisser/mettre vide

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.Une fois que l’on a bien renseigné ces informations, nous pouvons exécuter en cliquant en dessous :

Une image contenant texte, capture d’écran, logiciel, ligne

Le contenu généré par l’IA peut être incorrect.On peut voir sur la réponse que le code de retour est bien 200, ce qui indique que la requête a correctement fonctionnée.

Il nous suffit maintenant de récupérer le token disponible dans la réponse, et de se connecter avec. Tout en haut de la page, il y a un petit onglet avec marqué ‘Authorize’, et en cliquant dessus il nous suffira d’entrer le token :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Après avoir bien renseigné le token, nous pouvons voir que nous sommes bien connectés, et que nous pouvons ensuite faire des requêtes sur notre API. Il faut tout de même noter que chaque token, pour des raisons de sécurité, n’est disponible que 30 minutes. Il faudra donc se reconnecter de cette manière toutes les 30 minutes.

Pour faire une requête sur l’API, il faut simplement se rendre sur la route */resultats*, indiquer le code postal d’une ville ainsi que son nom, et l’API va nous retourner toutes les informations disponibles sur cette dernière :

*Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.exemple de retour de l’API sur les données de ‘La Ravoire’*

*BLOC 4 –*