Etude de Cas

Vidor Fabien, Doussiet Lou, Corin Gaetan

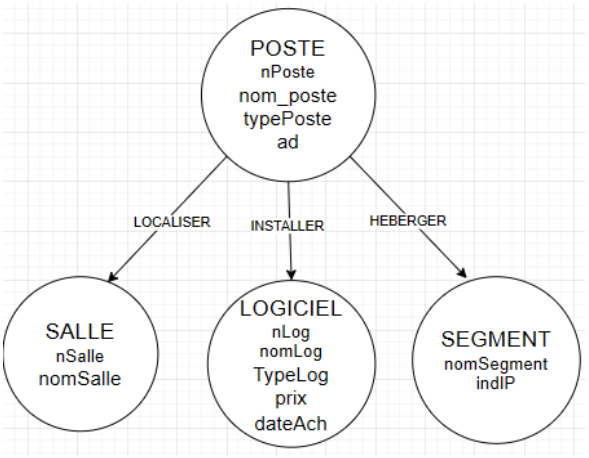
Etude de cas

1. Faire une étude comparative des trois solutions.

Neo4j	Cassandra	MongoDB
+ Fait en langage Java, rapidité de requête sur relations complexes + Langage de requête très performant, optimisé pour requête les plus courantes + Modularité dans la hiérarchisation des données - Mauvaise maturité pour export massif de données - Difficulté de sauvegarde et restauration dans un contexte d'automatisation	+ Disponibilité continu, même en cas de panne + Optimisé pour la surveillance des données grâce à des métriques et logs pertinents + Ajout de noeuds à chaud parfaitement scalable - Peu adapté aux relations complexes - Peu adapté aux évolutions fréquentes du schémas	+ Modèle Maître-esclave performant face aux pannes et répartition des charges + Système orienté document, aide à la modularité des éléments à intégrer dans la BDD + La scalabilité horizontale permet de gérer un grand nombre de données sur plusieurs machines avec une performance remarquable - Mal adapté aux relations complexes - Le modèle favorise la consistance au détriment de la disponibilité immédiate

2. proposer deux modélisations basées sur deux moteurs différents de votre choix, et 3. Proposer une structuration des données sous le moteur MongoDB ou Neo4J

La modélisation en Neo4j fonctionne sous un principe de nœuds et de liens. Il est donc possible de mettre des valeurs dans les nœuds et dans les liens. Il est aussi très simple de faire des relations en manytomany. Dans notre modélisation neo4J, le fait de contenir les valeurs uniquement dans les nœuds est suffisant.



La modélisation en MongoDB fonctionne sur un principe de collection.
MongoDB n'est pas très adapté pour les données structurées en entité ayant des relations, mais offre la possibilités de réaliser des imbrications d'entités grâce à l'architecture json.
Nous utilisons donc une architecture en imbrication afin de construire notre modélisation.

```
Poste {
        nomPoste
        typePoste
        ad
        Salle {
                nSalle
                nomSalle
        }
        Logiciel {
                nomLog
                typeLog
                prix
                dateAch
        }
        Segment {
                indIP
        }
}
```

4. Traduire les requêtes dans le langage MongoDB (ou Neo4j) et vérifier qu'elles sont passantes (apporter les transformations si nécessaire)

Requêtes réalisés en MongoDB

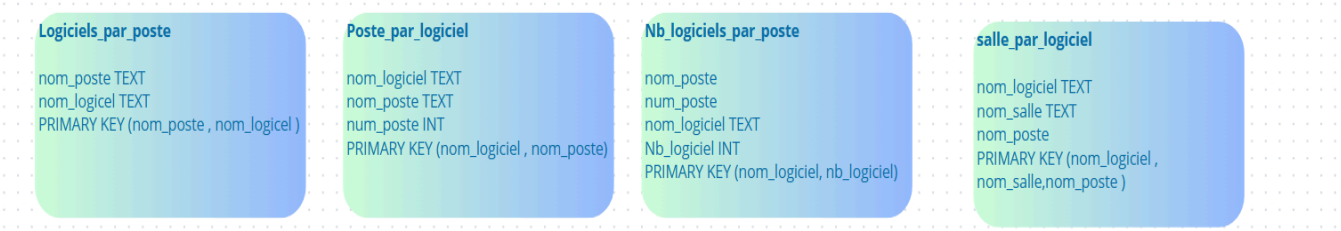
—Noms des logiciels UNIX

```
db.Poste.aggregate([
  { $unwind: "$Logiciel" },
  { $match: { "Logiciel.typeLog": "UNIX" } },
  { $project: { "Logiciel.nomLog": 1, "_id": 0 } }
```

```
])
—Type du poste p8
db.Poste.find({ nPoste: "p8" }, { typePoste: 1, _id: 0 });
—Nom, adresse IP, numéro de salle des postes de type UNIX ou PCWS
db.Poste.aggregate([
  { $match: { $or: [{ "typePoste": "UNIX" }, { "typePoste": "PCWS" }] } },
  { $unwind: "$Salle" },
  { $project: {
      "nomPoste": 1,
      "Segment.indIP": 1,
      "Salle.nSalle": 1,
      "_id": 0
    }
  }
])
—Postes du segment 130.120.80 triées par numéro de salle décroissant
db.Poste.aggregate([
  { $match: { "Segment.indIP": { $regex: "^130.120.80" } } },
  { $unwind: "$Salle" },
  { $sort: { "Salle.nSalle": -1 } },
  { $project: {
      "nomPoste": 1,
      "Segment.indIP": 1,
      "Salle.nSalle": 1,
      "_id": 0
    }
  }
])
—Numéros des logiciels installés sur le poste p6
db.Poste.aggregate([
  { $match: { "nomPoste": "p6" } },
  { $unwind: "$Logiciel" },
  { $project: { "Logiciel.prix": 1, "_id": 0 } }
])
—Numéros des postes qui hébergent le logiciel log1
db.Poste.aggregate([
  { $match: { "Logiciel.nomLog": "log1" } },
  { $project: { "nomPoste": 1, "_id": 0 } }
])

—Nom et adresse IP complète des postes de type TX
db.Poste.aggregate([
  { $match: { "typePoste": "TX" } },
  { $project: {
      "nomPoste": 1,
      "Segment.indIP": 1,
      "_id": 0
    }
```

```
  }
])
```
—Nombre de logiciels par poste
```
db.Poste.aggregate([
  { $project: {
      "nomPoste": 1,
      "nombreLogiciels": { $size: "$Logiciel" },
      "_id": 0
    }
  }
])
```
—Noms des salles hébergeant au moins un poste avec le logiciel ’Oracle 6’
```
db.Poste.aggregate([
  { $unwind: "$Logiciel" },
  { $match: { "Logiciel.nomLog": "Oracle 6" } },
  { $group: {
      _id: "$Salle.nomSalle"
    }
  })
])
```
5. Proposer une structuration des données sous le moteur Cassandra



6. Traduire les requêtes dans le langage de Cassandra et les adapter si besoin

7. Choisir le moteur NoSQL et la modélisation adéquate selon vous, et justifier votre choix et 8. Vérifier si votre modélisation peut répondre à l'évolution des traitements. Argumenter.

MongoDB offre le système le plus adapté pour l'exportation et la restauration de données massives, un point crucial pour IADATA dans le cadre de ses traitements critiques de facturation.

Sa facilité d'administration et sa scalabilité horizontale permettent de déployer facilement plusieurs clusters de pré-production, sans perte de performance au niveau du requêtage.

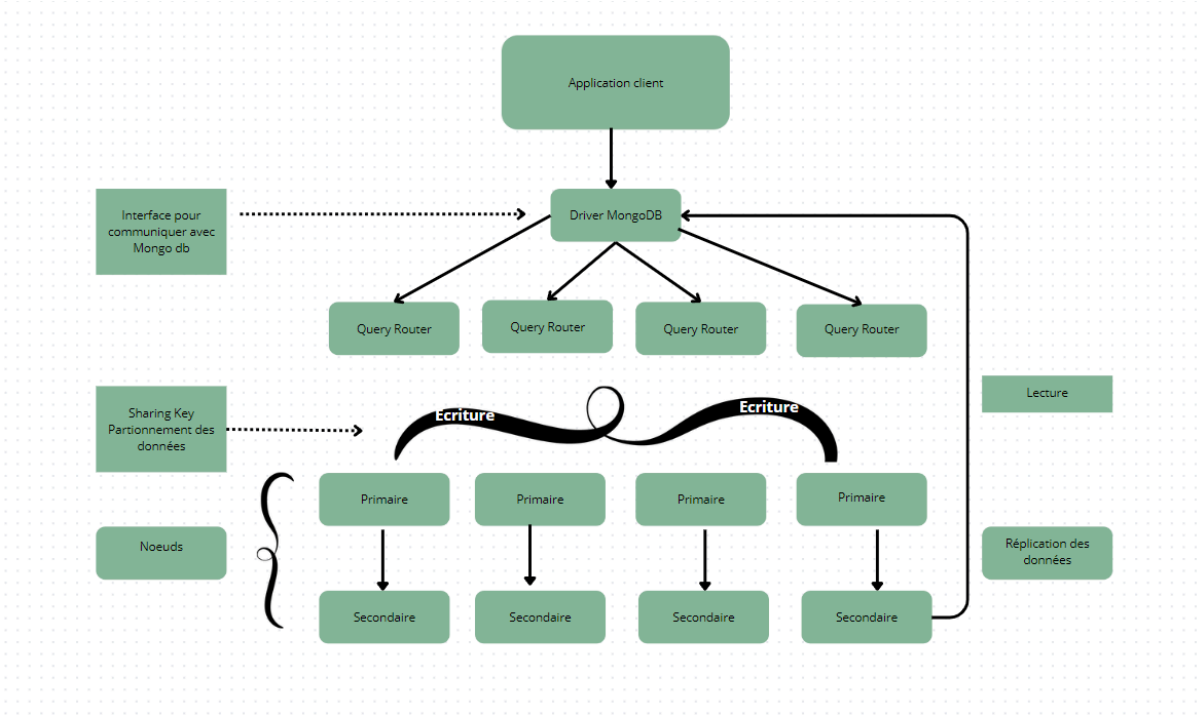
Enfin, MongoDB facilite l'import dans d'autres environnements sans configurations complexes, idéal pour tester et simuler différents workflows métier.

Nous suggérons une architecture mono collection car elle répond efficacement aux besoins de requêtage (filtrage par type, IP, salle, logiciels, etc.). Elle reste facile à étendre et performante pour la majorité des cas d'usage MongoDB.

On aurait une collection "poste" avec pour chaque poste ses données et une imbrication de ses dépendances (salles, logiciels, segments, etc…).

Une montée en charge du volume de données se traduira seulement par l'ajout de nœuds, sans refonte de l'architecture déjà en place.

9. Proposer une architecture de cluster, en précisant les aspects de cohérence, de résilience, de réplication, etc.



10. Justifier votre architecture à la lumière du modèle CAP.

Théorème CAP
Cohérence, Disponibilité, Tolérance aux partitions
Nous pouvons garantir la **cohérence** des données grâce aux Réplicats et c'est-à-dire que les données sont répliquées des nœuds primaire aux nœuds secondaires.
Tous les nœuds d'un système distribué ont la même vue des données en même temps.
Les opérations de lecture sur le système distribué doivent renvoyer l'écriture la plus récente ou une erreur.

Disponibilité
Si le nœud primaire tombe, MongoDB élit un nouveau nœud primaire pour garder la disponibilité. Pendant ce processus d'élection, il peut y avoir une indisponibilité temporaire. Cela s'appelle le failover. Les écritures reprennent sans intervention humaine après un delai de 5 à 30 secondes.

Tolérance au Partition:
MongoDB sacrifie la disponibilité pour garantir la cohérence dans le cas où la majorité des nœuds est indisponible.
MongoDB est classé CP (cohérence + tolérance au partition) dans le théorème CAP.

11. Sachant que le client exprime un contexte où la cohérence de données est un point très critique, proposer une politique de consistance (en lecture et écriture) pour répondre à cette exigence.

Etant donnée que nous travaillons en cluster MongoDB ayant plusieurs nœuds, nous avons donc plusieurs répliques de données au sein de ces nœuds.

Politique de consistance en écriture:
Write Concern “majority”:
collection.with_options(write_concern=WriteConcern('majority'))
permet de propagées à la majorité des répliques avant que l’écriture ne soit confirmée.

Politique de consistance en lecture:
Read Concern “majority”:
collection.with_options(read_concern=ReadConcern('majority'))
permet que les données qui sont lues proviennent de la majorité des répliques

Contexte d’exploitation, sauvegarde et restauration

1. Définir une politique de sauvegarde adaptée au moteur choisi (fréquence, format, cohérence, gestion des exports).

La politique de sauvegarde aura 2 sauvegardes quotidienne:

- La première avant la nuit (et donc avant les traitements critiques de facturation)
- la seconde après le nuit (après les traitements critiques de facturation)

La première sauvegarde permet de facilement revenir à l'état antérieur juste avant les traitements critiques, en cas de problème causé durant la nuit.
La seconde sauvegarde est juste après les traitements critiques. Elle permet de facilement analyser la situation, afin de comprendre les potentiels problèmes d'exécution.

Il y a 2 types de format de sauvegarde demandés:

- Une première sauvegarde en copie complète. Pour cela, “mongodump” va créer une copie complète des données de la base MongoDB en fichier binaire, qui pourra être envoyé au partenaire externe.
- La seconde sauvegarde en format CSV: Pour cela, “mongoexport” permet de réaliser des extractions spécifiques sans forcément envoyer la base complète. Le ou les CSV pourront donc être envoyés au partenaire externe.

2. Proposer une politique de restauration pour :

- La production en cas d'échec nocturne.
  En cas d’échec nocturne des traitements critiques de facturation durant la nuit, il est possible de restaurer la donnée avec “mongodump”. Nous pouvons utiliser la première sauvegarde (avant la nuit) pour restaurer la donnée juste avant l’échec nocturne.
- La préparation d’un cluster de pré-production à partir d’une sauvegarde.

La pré-production peut être lancée à partir d'une sauvegarde. Celle-ci peut être la première ou la seconde sauvegarde, la aussi en utilisant "mongodump"

3. Décrire comment gérer les cas de sauvegardes incomplets (certaines instances non sauvegardées).

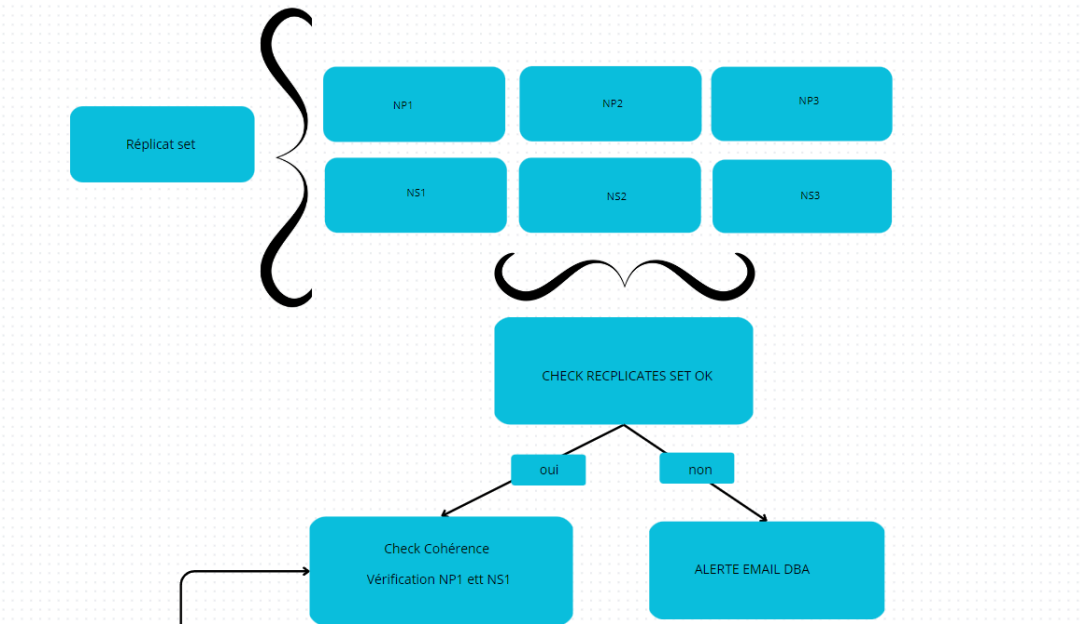
Voici les points a suivre lors de sauvegarde incomplets:

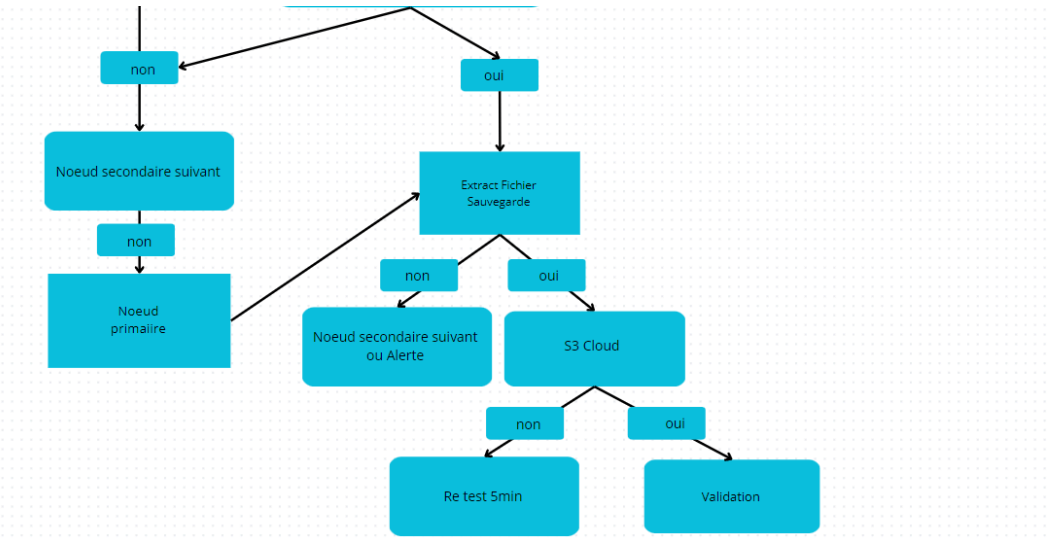
- Vérification de la complétude de la sauvegarde (création de rapport de vérification, avec un système d'alerte par email par exemple en cas d'échec de sauvegarde)
- Mongodump génère des logs qui peut être inspectés régulièrement pour détecter toute anomalie
- Si une sauvegarde incomplète de données est détectée, une nouvelle sauvegarde complète doit être lancée immédiatement, et de manière automatique, en vérifiant que toutes les collections sont bien prises en compte.
- Si la seconde sauvegarde ne s'exécute pas convenablement, une alerte doit être envoyée afin de pouvoir analyser le problème le plus rapidement possible.

4. Préciser les conditions pour qu'une sauvegarde soit considérée comme valide et exploitable

Il y a plusieurs conditions qui doivent être remplis afin qu'une sauvegarde soit considérés comme valide et exploitable:

- Les données doivent être cohérentes à l'instant de la sauvegarde (en ayant une cohérence des écritures sur les différents noeuds au moment de la sauvegarde)
- Une complétude de la sauvegarde de la données (la données doit inclure toutes les collections de la base de données sans omissions)
- accessibilité des sauvegardes (la sauvegarde doit être disponible hors ligne ou en cloud, afin que celle ci soit disponible si la prod crash ou est perdu)
- validation des sauvegardes (le système doit créer des validations automatiques afin que celle-ci soit récurrentes et systémiques)



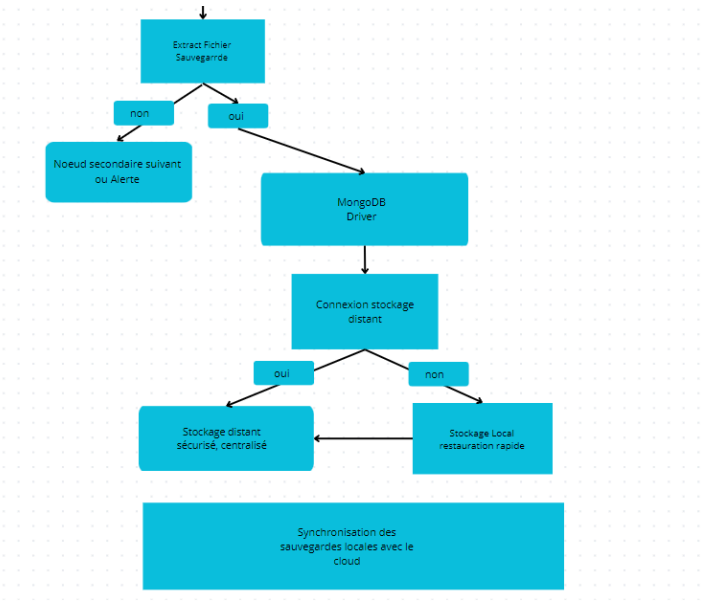


2.1.

Sauvegarde localisée vs externalisée

1. Décrire une solution de sauvegarde compatible avec ces contraintes (local fallback + push différé).

Une solution de sauvegarde localisée permet de revenir à un état de fonctionnement antérieur ou alternatif quand il rencontre un problème.
La solution serait d'automatiser une sauvegarde journalière aux heures où l'activité est la plus calme avec un push différé pour effectuer un envoi vers une solution de stockage cloud par exemple.



2. Définir les implications de ce modèle sur la politique de restauration : comment garantir l'accès aux données si une partie est locale et une autre sur backend distant ?

L'accès aux données se fait autant sur le cloud qu'en local. Le double stockage évite la perte de données en cas de défaillance d'un des deux emplacements.
Le push différé garantit que ces deux copies sont synchronisées de manière fiable sans compromettre la performance du système principal.

3. Proposer un mécanisme de suivi ou d'orchestration de l'état des sauvegardes et de leur synchronisation.

Un ordonnanceur pour automatiser les sauvegardes et la mise en place d'un service de monitoring avec une gestion des logs et un système d'alerte pourrait être une solution.

Démarrage consistant et automatique

Etape 1 Vérification de l'intégration entre les noeuds
Etape 2 Synchronisation des noeuds
Etape 3 Vérification que ⅔ soient opérationnels
Etape 4 Ecriture confirmée par la majorité
Etape 5 Lecture confirmée par la majorité
Etape 6 Synchronisation

Il est possible d'utiliser Oracle Data Guard, qui permet de faire une bascule automatique en cas de panne (avec Fast Start Failover).

-