Table de matières

Introduction générale

Plusieurs administrations et entreprises font face à une explosion de leurs bases de données. Ces masses d'informations manipulées et stockées amènent naturellement à une dégradation conséquente dans les performances, ce qui pousse les différents acteurs à chercher des remèdes en multipliant leurs infrastructures en basculant et en évoluant dans des environnements distribués avec tous ses types et ses modèles de services offerts.

Cette tendance n'est pas une mode, mais elle est liée au fait que cet environnement permet de rendre une infrastructure (plateforme et software) dynamique et flexible en exposant les capacités des machines et les centres des données (Data Centers) comme étant un réseau de services virtuelles. Les utilisateurs comme les développeurs peuvent ainsi accéder et déployer des applications et des services de n'importe où dans le réseau en se guidant par la demande et les exigences de la qualité de service.

Pour les administrations, le basculement vers les environnements distribués permet d'optimiser et de consolider les ressources, et donc réduire les coûts. Les propriétés les plus importantes offertes pour les différents utilisateurs dans un environnement distribué sont la haute disponibilité et l'élasticité (Scalability) qui peuvent être assurées grâce à la possibilité d'ajouter ou retirer des nœuds (machines physiques ou virtuelles). La haute disponibilité garantie le fonctionnement continu d'un service, tandis que l'élasticité assure la capacité du service demandé à passer à l'échelle de manière dynamique lors de l'ajout ou retrait à chaud des nœuds.

Les bases de données figurent parmi les services les plus importants pour toutes les administrations et les entreprises, car elles représentent la mémoire collective qui regroupe toutes les informations sensibles de l'entreprise. L'implantation directe des bases de données dans un environnement distribué, a fait apparaître de nouvelles problématiques liées surtout à la performance et l'efficacité de la gestion de ces bases de données. Malheureusement, cette difficulté provient principalement du modèle dont sont représentées les données.

Introduction Générale

La plus part des bases de données déployés sont bâties sur une conception relationnelle qui n'est pas bien adéquate à l'évolution et à la monté en charge offerte dans ce type d'environnement distribué.

Le Big Data est un ensemble de techniques permettant le passage à l'échelle en volumes, en nombres et en types de données. Les problématiques du Big Data s'inscrivent dans un contexte complexe, à la croisée de 2 préoccupations majeures :

- La mise en œuvre de nouvelles solutions de stockage de masse ;
- La capture d'information à grande vitesse et si possible en temps réel ;

Pour répondre à ces trois préoccupations, il faut développer des solutions technologiques comprenant :

- Des outils innovants de restitution ;
- Des outils innovants de visualisation de données adaptés aux volumétries ;
- De nouvelles solutions de stockage.

De ce fait, et face à ces contraintes, des nouvelles systèmes de gestion de données nommés «NoSQL» viennent occuper leurs places en proposant des alternatives au modèle relationnel. Les bases de données NoSQL offrent de nouvelles solutions de stockage dans les environnements à grande échelle

Beaucoup d'utilisateurs des SGBD classiques dits « SQL » veulent basculer vers ces nouvelles solutions « NoSQL » pour anticiper l'explosion de leurs données dans le futur, néanmoins ils ne veulent certainement pas commencer avec des bases vides mais chercher comment récupérer les données hébergées dans leurs bases de données relationnelles.

Dans le cadre de notre PFE, nous proposons une approche de migration de la base de données relationnelle des passeports biométriques sous Oracle vers une base de données NoSQL MongoDB, une application qui peut être déployée par le ministère de l'intérieur et des collectivités locales dans les prochaines échéances.

La démarche proposée permettra de :

- Maîtriser la migration des bases de données relationnelles vers des bases de données NoSQL.
- Manipuler et héberger un système NoSQL choisi.

Notre mémoire sera organisé comme suit : après une introduction, nous présenterons dans le premier chapitre un aperçu sur les bases de données relationnelles ainsi que les limites atteintes par ces modèles dans les environnements distribués.

Le deuxième chapitre, va être consacré à la technologie NoSQL, les différents types de bases de données NoSQL vont être recensés ainsi que leurs avantages.

Dans la première partie du troisième chapitre, nous décrirons le domaine d'étude qui est le service de la biométrie au sein de la daïra de Sebdou en premier lieu, en deuxième lieu nous essayons de justifier notre choix du système NoSQL à savoir MongoDB. Cette dernière sera décrite en détail par la suite.

La deuxième partie du troisième chapitre va faire objet de l'implémentation et la mise en œuvre de la migration du SGBD relationnel Oracle vers le NoSQL MongoDB.

Enfin, nous clôturons cette étude par une conclusion générale sur le travail effectué ainsi que les perspectives éventuelles et possibles à projeter dans cet axe de recherche.

CHAPITRE 1 :

Bases de données relationnelles et leurs limites

1. INTRODUCTION

Dans ce premier chapitre, nous présentons les bases de données relationnelles ainsi que leurs limites atteintes dans les environnements à grande échelle. Il ne s'agit pas de détailler exhaustivement ce type de bases de données et les notions liées à ce modèle, mais juste faire un survol global sur les principes et les concepts les plus importants, ensuite nous montrons les limites liées à l'usage des bases de données relationnelles face aux nouveaux besoins des systèmes d'information d'un part, et face aux nouvelles solutions de stockage et de manipulation des données d'autre part. Ces limites nous poussent à chercher d'autres architectures pour une éventuelle migration.

2. DEFINITIONS ET CONCEPTS DE BASES

2.1 Définition d'une base de données

De façon informelle, on peut considérer une base de données comme un ensemble structuré de données, centralisées ou non, servant pour les besoins d'une ou plusieurs applications, interrogeables et modifiables par un groupe d'utilisateurs en un temps opportun.

Plus formellement, une base de données est un ensemble d'informations exhaustives, non redondantes, structurées et persistantes, concernant un sujet [1]. Une donnée est dite "persistante" si sa durée de vie excède la durée d'exécution du programme qui l'a créée. Une donnée persistante peut être rechargée en mémoire principale à tout instant. En pratique la persistance de données est assurée par leur stockage sur un support permanent (disque…) [2].

2.2 Définition d'un système de gestion de base de données

Un **S**ystème de **G**estion de **B**ases de **D**onnées **(SGBD)** peut être défini comme un ensemble de logiciels prenant en charge la structuration, le stockage, la mise à jour et la maintenance des données. Autrement dit, il permet de décrire, modifier, interroger et administrer les données. C'est, en fait, l'interface entre la base de données et les utilisateurs (qui ne sont pas forcément informaticiens) [1].

Ce sont là les fonctions primaires, complétées par des fonctions souvent plus complexes, destinées par exemple à assurer le partage des données mais aussi à protéger les données contre tout incident et à obtenir des performances acceptables [3].

3. OBJECTIFS DES SGBD

3.1 Administration facilitée des données

Un SGBD doit fournir des outils pour décrire les données, à la fois leurs structures de stockage et leurs présentations externes. Il doit permettre le suivi de l'adéquation de ces structures aux besoins des applications et autoriser leur évolution aisée. Les fonctions qui permettent de définir les données et de changer leur définition sont appelées outils d'administration des données.

3.2 Efficacité des accès aux données

Les performances en termes de débit (nombre de transactions types exécutées par seconde) et de temps de réponse (temps d'attente moyen pour une requête type) sont un problème clé des SGBD. L'objectif de débit élevé nécessite un temps de latence minimal dans la gestion des tâches accomplie par le système. L'objectif de bons temps de réponse implique qu'une requête courte d'un utilisateur n'attende pas une requête longue d'un autre utilisateur. Il faut donc partager les ressources (unités centrales, unités d'entrées-sorties) entre les utilisateurs en optimisant l'utilisation globale et en évitant les pertes en commutation de contextes.

3.3 Redondance contrôlée des données

Dans les systèmes classiques à fichiers non intégrés, chaque application possède ses données propres. Cela conduit généralement à de nombreuses duplications de données avec, outre la perte en mémoire secondaire associée, un gâchis important en moyens humains pour saisir et maintenir à jour plusieurs fois les mêmes données. Avec une approche base de données, les fichiers plus ou moins redondants seront intégrés en un seul fichier partagé par les diverses applications. L'administration centralisée des données conduisait donc naturellement à la non duplication physique des données afin d'éviter les mises à jour multiples.

3.4 Cohérence des données

Au niveau d'ensemble de données, il peut exister certaines dépendances entre les données. Par exemple, une donnée représentant le nombre de commandes d'un client doit correspondre au nombre de commandes dans la base. Plus simplement, une donnée élémentaire doit respecter un format et ne peut souvent prendre une valeur quelconque. Par exemple, un salaire mensuel doit être supérieur à 18 000 DA et doit

raisonnablement rester inférieur à 700 000 DA. Un système de gestion de bases de données doit veiller à ce que les applications respectent ces règles lors des modifications des données et ainsi assurer la cohérence des données. Les règles que doivent explicitement ou implicitement suivre les données au cours de leur évolution sont appelées contraintes d'intégrité.

3.5 Partage des données

L'objectif est ici de permettre aux applications de partager les données de la base dans le temps mais aussi simultanément. Une application doit pouvoir accéder aux données comme si elle était seule à les utiliser, sans attendre mais aussi sans savoir qu'une autre application peut les modifier concurremment.

3.6 Sécurité des données

Cet objectif a deux aspects. Tout d'abord, les données doivent être protégées contre les accès non autorisés ou mal intentionnés. Il doit exister des mécanismes adéquats pour autoriser, contrôler ou enlever les droits d'accès de n'importe quel usager à tout ensemble de données. D'un autre côté, la sécurité des données doit aussi être assurée en cas de panne d'un programme ou du système, voire de la machine. Un bon SGBD doit être capable de restaurer des données cohérentes après une panne disque, bien sûr à partir de sauvegardes. Aussi, si une transaction commence une mise à jour (par exemple un transfert depuis votre compte en banque sur celui de l'auteur) et est interrompue par une panne en cours de mise à jour, le SGBD doit assurer l'intégrité de la base (c'est-à-dire que la somme d'argent gérée doit rester constante) et par suite défaire la transaction qui a échoué [3].

4. PROPRIETES D'UN SGBD RELATIONNEL

Les fonctions citées ci-dessus découlent les propriétés fondamentales d'un SGBDR :

- Base formelle reposant sur des principes parfaitement définis.
- Organisation structurée des données dans des tables interconnectées (d'où le qualificatif relationnelles), pour pouvoir détecter les dépendances et les redondances des informations.
- Implémentation d'un langage relationnel ensembliste permettant à l'utilisateur de décrire aisément les interrogations et manipulation qu'il souhaite effectuer sur les données.

- Indépendance des données vis-à-vis des programmes applicatifs (dissociation entre la partie "stockage de données" et la partie "gestion" - ou "manipulation").
- Gestion des opérations concurrentes pour permettre un accès multi-utilisateur sans conflit.
- Gestion de l'intégrité des données, de leur protection contre les pannes et les accès illicites [1].

5. CONCEPTS PRINCIPAUX

5.1 Table (ou Relation) et Attribut

Une table (ou relation) est un tableau à deux dimensions dans lequel chaque colonne (appelée Attribut) porte un nom différent et où les données figurent en ligne. Une table peut en contenir un nombre quelconque et leur ordre est indifférent. On trouve encore les vocables d'enregistrement, n-uplet ou tuple pour désigner une ligne et de champ pour une colonne ; un enregistrement est donc un ensemble de valeurs, chacune renseignant un champ [4].

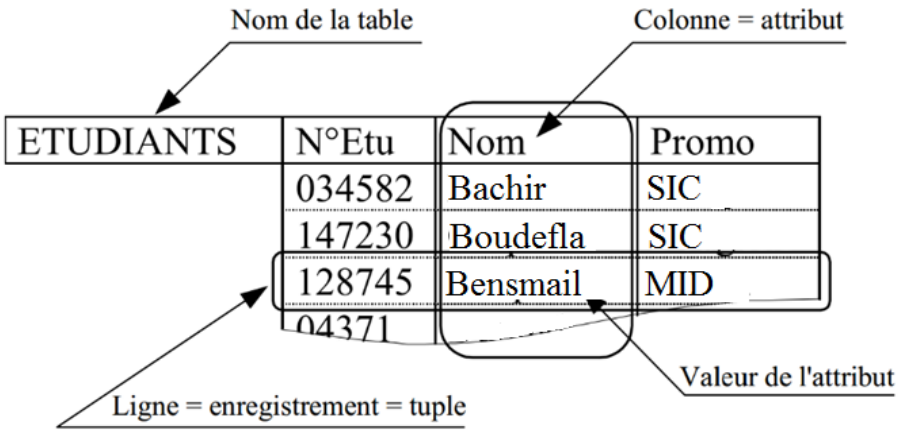

Figure I.1 : Exemple de relation

5.2 Clés

5.2.1 Clés primaires

Dans la table de l'exemple ci-dessus, le numéro d'étudiant est un attribut constitué d'un code artificiel. Cet attribut particulier, appelé clé, est indispensable pour pouvoir identifier chaque étudiant de manière unique, car aucun autre attribut ni aucune combinaison d'attributs n'auraient pu garantir cela. En effet, l'attribut nom peut prendre plusieurs fois la même valeur et n'est donc pas un bon candidat pour être une clé de la

table, de même si l'on avait disposé du prénom des étudiants, la composition d'attributs <nom, prénom> n'aurait pas été davantage satisfaisante comme clé, car il est possible que 2 étudiants portent à la fois le même nom et le même prénom [1].

Une clé d'identification ou clé d'une table est un attribut ou une combinaison d'attributs qui satisfait :
- une contrainte d'unicité : chaque valeur clé désigne de manière unique une ligne de la table autrement dit, deux lignes ne peuvent posséder des valeurs identiques pour le(s) attribut(s) de la clé
- une contrainte de minimalité : si une clé est composée d'un ensemble d'attributs, cette combinaison doit être minimale (aucun attribut ne peut en être retiré sans violer la règle d'unicité).

5.2.2 Clés étrangères

Une clé étrangère est la duplication de la clef primaire d'une table dans une autre table dans le but de traduire une association conceptuelle entre les deux entités représentées par les deux tables. Ce concept est un des concepts majeurs du modèle relationnel [2].

6. LANGAGES DE MANIPULATION DES DONNEES RELATIONNELLES

6.1 Introduction

Les langages utilisés dans les bases de données relationnelles se fondent sur l'algèbre relationnelle et/ou le calcul relationnel. Ce dernier est basé sur la logique des prédicats (expressions d'une condition de sélection définie par l'utilisateur). On peut distinguer 3 grandes classes de langages :

- **les langages algébriques**, basés sur l'algèbre relationnelle d'Edgar Frank Codd, dans lesquels les requêtes sont exprimées comme l'application des opérateurs relationnels sur des relations. Le langage SQL, standard pour l'interrogation des bases de données, fait partie de cette catégorie.

- **les langages basés sur le calcul relationnel de tuples**, construits à partir de la logique des prédicats, et dans lesquels les variables manipulées sont des tuples (ex. : QUEL, Query Language).

- **les langages basés sur le calcul relationnel de domaines**, également construit à partir de la logique des prédicats, mais en faisant varier les variables sur les domaines des relations.

Les langages de manipulation de données sont dits déclaratifs, ou assertionnels, c'est-à-dire que l'on spécifie les propriétés des données que l'on manipule et pas – comme dans un langage impératif – comment y accéder [1].

6.2 Langage SQL

SQL (Structured Query Language) a été créé au début dans les années 1970 par la société américaine IBM (International Business Machines) et sa première version commercialisable a vu le jour en 1979. Depuis il a été repris par le groupe ORACLE qui a lancé son premier SGBD Relationnel et a été normalisé par l'Institut de normalisation américaine ANSI en 1986 et ratifié par l'Organisation internationale de normalisation ISO en 1987 [5].

Le langage SQL peut être considéré comme le langage d'accès normalisé aux bases de données. Il est aujourd'hui supporté par la plupart des produits commerciaux que ce soit par les systèmes de gestion de bases de données micro tel qu'Access ou par les produits plus professionnels tels qu'Oracle. Il a fait l'objet de plusieurs normes ANSI/ISO dont la dernière est la norme SQL3 qui a été définie en 1999, et qui a subi plusieurs révisions dont la plus récente date de 2011. Le succès du langage SQL est dû essentiellement à sa simplicité et au fait qu'il s'appuie sur le schéma conceptuel pour énoncer des requêtes en laissant le SGBD responsable de la stratégie d'exécution. SQL est un langage déclaratif qui permet d'interroger une base de données sans se soucier de la représentation interne (physique) des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires. A ce titre il s'adresse à une large communauté d'utilisateurs potentiels (pas seulement des informaticiens) et constitue un des atouts les plus spectaculaires des SGBDR [6].

Néanmoins, le langage SQL ne possède pas la puissance d'un langage de programmation : entrées/sorties, instructions conditionnelles, boucles et affectations. Pour certains traitements il est donc nécessaire de coupler le langage SQL avec un langage de programmation plus complet [7].

SQL est un langage qui permet :

- la définition des données (CREATE, ALTER, DROP, RENAME, …)
- la manipulation de données (INSERT, UPDATE, DELETE, …)
- l'interrogation de la base (SELECT)

- le contrôle des données (GRANT, REVOKE, COMMIT, ROLLBACK, …)

Ordres SQL	Aspect du langage
CREATE - ALTER - DROP - RENAME - TRUNCATE	Définition des données (LDD)
INSERT - UPDATE - DELETE - LOCK TABLE	Manipulation des données (LMD)
SELECT	Interrogation des données (LID)
GRANT - REVOKE - COMMIT - ROLLBACK - SAVEPOINT - SET TRANSACTION	Contrôle des données (LCD)

Figure I.2 : Classification des ordres SQL [7]

6.3 SQL au sein d'un SGBD

Tout SGBD contient un dictionnaire (voir figure I.3) qui contient à tout moment le descriptif complet des données, sous la forme d'un ensemble de tables comprenant les bases de données présentes, les tables et leurs attributs, les clés candidates, primaires et étrangères, les différentes contraintes et les vues créées sur les tables. SQL utilise ce dictionnaire lors de l'exécution de requêtes pour vérifier les contraintes, optimiser les requêtes et assurer le contrôle des accès [1].

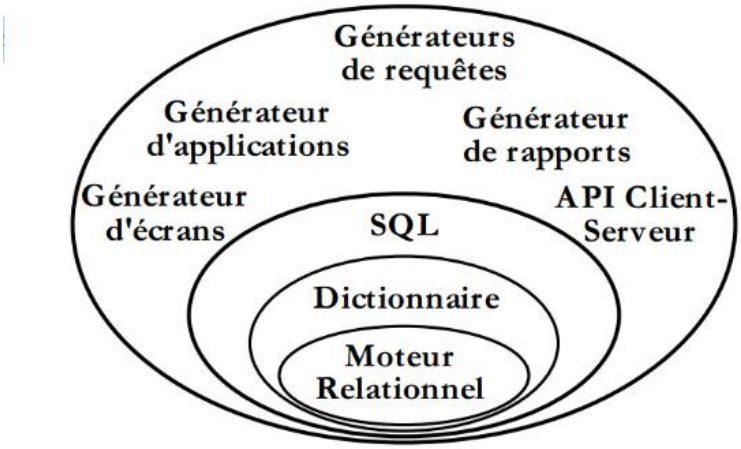

Figure I.3 : SQL au sein d'un SGBD [1]

7. PROPRIETES ACID

La plupart des SGBD relationnels sont transactionnelles. Une transaction est une suite d'opérations faisant passer l'état du système d'un point A (antérieur à la transaction) à un point B (postérieur à la transaction). Dans les SGBD Relationnels une transaction doit respecter les propriétés **ACID** (**A**tomicité, **C**onsistance, **I**solation,

Durabilité) afin que celle-ci soit validée ; l'acronyme ACID permet de garantir la fiabilité d'une transaction informatique telle qu'un virement bancaire par exemple.

- **Atomicité** : cette propriété assure qu'une transaction soit effectuée complètement, ou pas du tout. Quelle que soit la situation, par exemple lors d'une panne d'électricité, du disque dur ou de l'ordinateur, si une partie de la transaction n'a pu être effectuée, il faudra effacer toutes les étapes de celle-ci et remettre les données dans l'état où elles étaient avant la transaction.
- **Cohérence** : la cohérence assure que chaque transaction préserve la cohérence de données, en transformant un état cohérant en un autre état de mêmes données cohérentes. La cohérence exige que les données concernées par une potentielle transaction soient protégés sémantiquement.
- **Isolation** : l'isolation assure que chaque transaction voit l'état du système comme si elle était la seule à manipuler la base de données. En d'autres termes, si pendant la transaction T1, la transaction T2 s'exécute au même moment, T1 ne doit pas la voir, tout comme T2 ne doit pas voir T1.
- **Durabilité** : La durabilité assure qu'une transaction confirmée le demeure définitivement, peu importe les différents imprévus (panne d'électricité, panne d'ordinateur etc.). Pour ce faire, un journal contenant toutes les transactions est créé, afin que celles-ci puissent être correctement terminées lorsque le système est à nouveau disponible.

8. LIMITES DE SGBD RELATIONNELS

Le modèle relationnel est fondé sur des concepts simples qui font sa force en même temps que sa faiblesse. Nous expliquerons ici les principales limites des SGBD Relationnels :

8.1 Modélisation des entités réelles

Les relations ne correspondent pas toujours à des entités du monde réel (notamment pour les objets complexes). Les processus de normalisation et de décomposition des schémas relationnels entraînent la multiplication des relations, et par conséquent des opérations de jointure lors du traitement des requêtes [8].

8.2 Surcharge sémantique

Le modèle relationnel s'appuie sur un seul concept (la relation) pour modéliser à la fois les entités et les associations / relations entre ces entités. Il existe donc un décalage entre la réalité et sa représentation abstraite [8].

8.3 Contraintes d'intégrité

Les SGBD Relationnels sont limités à des contrôles de cohérence simples. Il est donc difficile de modéliser des contraintes réelles correspondant aux données d'une entreprise. Ce problème n'est que partiellement résolu avec SQL2 qui permet d'exprimer des contraintes dans la partie langage de définition [8].

8.4 Langage de manipulation

Il est limité aux opérateurs de base du langage SQL, qu'il n'est pas possible d'étendre à de nouveaux opérateurs.

8.5 Scalabilité limitée

Atteinte par les groupes comme Yahoo, Google, Facebook...etc. Cette limite est la plus gênante pour les entreprises. Dans le cas de traitements de données en masse, le constat est simple : les SGBD Relationnels ne sont pas adaptés aux environnements distribués requis par les volumes gigantesques de données et par les trafics tout aussi gigantesques générés par ces opérateurs [9].

8.6 Propriétés ACID en milieu distribue

Les propriétés ACID, bien que nécessaires à la logique du relationnel, nuisent fortement aux performances surtout la propriété de cohérence. En effet, la cohérence est très difficile à mettre en place dans le cadre de plusieurs serveurs (environnement distribué), car pour que celle-ci, pour pouvoir satisfaire cette cohérence, il faut que tous les serveurs doivent être des miroirs les uns des autres, de ce fait deux problèmes apparaissent :

- Le coût en stockage est énorme car chaque donnée est présente sur chaque serveur.
- Le coût d'insertion/modification/suppression est très grand [9].

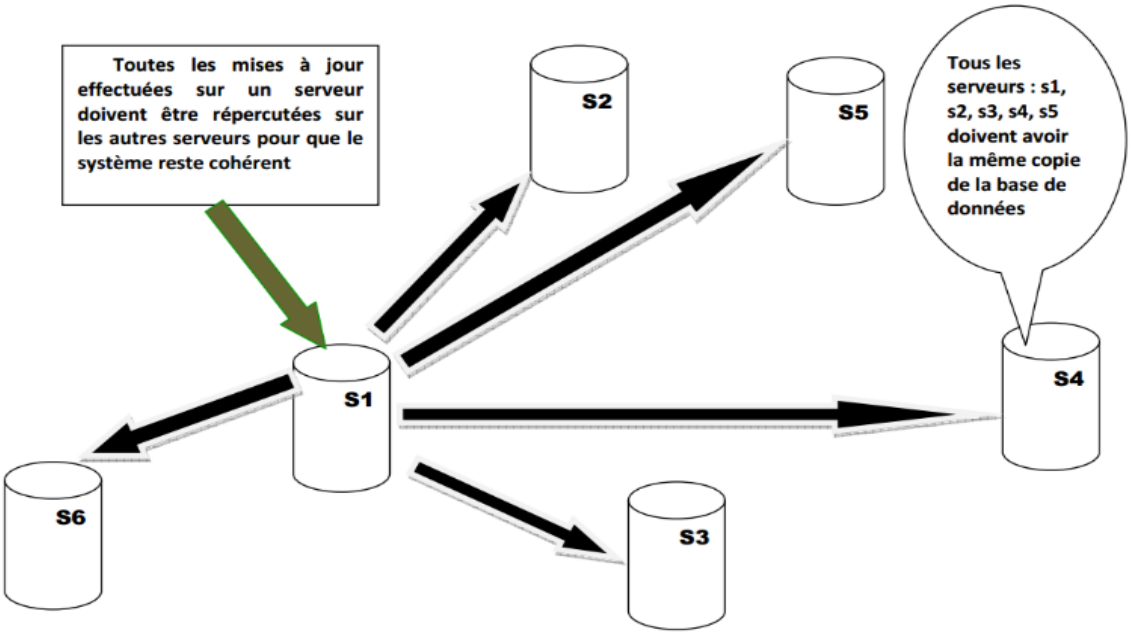

Figure I.4 : Problème lié aux propriétés ACID en milieu distribué [9]

8.7 Limites face aux usages

Les structures de données manipulées par les systèmes sont devenues de plus en plus complexes avec en contrepartie des moteurs de stockage peu évoluant. Le principal point faible des modèles relationnels est l'absence de gestion d'objets hétérogènes ainsi que le besoin de déclarer au préalable l'ensemble des champs représentant un objet [10].

9. CONCLUSION

Les bases de données sont considérées comme les éléments les plus importants dans le monde informatique. Le modèle relationnel est imposé comme une norme et il est très répandu grâce à son efficacité et sa robustesse issues par les contraintes imposées par ce modèle.

Ces mêmes principes qui constituent la force de ce modèle, le rendent moins performant et moins réactif dans un environnement distribué car elles empêchent les données d'être distribuées efficacement dans les nœuds constituants l'environnement distribué à grande échelle. De ce fait, et face à ces contraintes, des nouvelles systèmes de gestion de données nommés «NoSQL» viennent occuper leurs places en proposant des alternatives au modèle relationnel.

CHAPITRE 2 :

Bases de données NoSQL

1. INTRODUCTION

Après avoir exposé les bases de données relationnelles dans le chapitre précédent, et montrer les limites de ce type surtout dans des environnements distribués et a forte charges, nous établirons dans ce chapitre un état de l'art sur les nouveaux systèmes dites NoSQL. Les systèmes NoSQL viennent pour garantir certains critères que les bases de données classiques ont montré des faiblesses.

Nous allons voir leur émergence, leurs différents types et nous allons montrer leur utilité face aux nouvelles exigences ainsi un bref aperçu des solutions NoSQL disponibles actuellement sur le marché.

2. HISTORIQUE

Bien que le terme NoSQL soit entendu pour la première fois en 1988 par **Carlo Strozzi** qui présentait ça comme un modèle de base de donnée plus léger et sans interface SQL, c'est en 2009, lors de la rencontre NoSQL de San Francisco, qu'il prend un essor important. Durant cette conférence, les développeurs ont présenté des projets tels que Voldemort, Cassandra Project, Dynomite, HBase, Hypertable, CouchDB, MongoDB. Cette conférence était considérée comme l'inauguration de la communauté des développeurs de logiciel NoSQL [11].

NoSQL est une combinaison de deux mots : No et SQL. Qui pourrait être mal interprétée car l'on pourrait penser que cela signifie la fin du langage SQL et qu'on ne devrait donc plus l'utiliser. En fait, le « No » est un acronyme qui signifie « Not only », c'est-à-dire en français « non seulement », c'est donc une manière de dire qu'il y a autre chose que les bases de données relationnelles [12].

3. EMERGENCE DU NOSQL

Avec l'augmentation de la bande passante sur internet, ainsi que la diminution des coûts des matériels informatiques, de nouvelles possibilités ont vu le jour dans le domaine de l'informatique distribuée. Le passage au 21ème siècle par la révolution du WEB 2.0 a vu le volume de données augmenter considérablement. Ces données proviennent essentiellement de réseaux sociaux, base de données médicales, indicateurs économiques, etc. La gestion de ces volumes de données est donc devenue un problème que les bases de données relationnelles n'ont plus été en mesure de gérer. Le NoSQL

regroupe donc de nombreuses bases de données qui ne se reposent donc plus sur une logique de représentation relationnelle. Il n'est toutefois pas simple de définir explicitement ce qu'est une base de données NoSQL étant donné qu'aucune norme n'a encore été instaurée.

Le besoin fondamental auquel répond le NoSQL est la performance. Afin de résoudre les problèmes liés à la gestion de grandes quantités de donnée « Big data », les systèmes NoSQL sont optimisés pour la scalabilité horizontale. Un autre aspect important du NoSQL est qu'il répond au théorème de CAP qui est plus adapté pour les systèmes distribués. Servant à manipuler de gros volumes de données, il est donc destiné aux grandes entreprises. Dès 2010, le NoSQL a commencé à s'étendre vers les plus petites entreprises n'ayant pas les moyens d'acquérir des licences Oracle qui ont donc développé leurs propres SGBD en imitant les produits Google et Amazon [11].

4. DIFFERENCES ENTRE LE NOSQL ET LE RELATIONNEL

Pour commencer, il est important de savoir que le SQL n'est pas un modèle relationnel en soit, mais un langage de manipulation de données conçu autour du modèle relationnel. Les bases de données NoSQL n'ont pas pour but de s'éloigner de ce langage mais du modèle relationnel. Nous allons voir les facteurs différenciateurs.

4.1 Caractéristiques du modèle relationnel

4.1.1 Modèle de données

Le modèle relationnel est basé est sur un modèle mathématique qui est celui de la notion des ensembles. Chaque ensemble ici est représenté par une table, et ces attributs sont quant à eux représentés par des colonnes. L'un des principes fondamentaux est justement cette notion de relation entre tables à l'aide de cardinalités, clés primaires et clé étrangères, ceci implique au préalable une étude minutieuse sur la modélisation du schéma de la base de données pour identifier les tables dont le système aura besoin, ses attributs, les relations possibles entre différentes tables, etc. Une fois ce modèle mis en place dans un SGBD Relationnel, il est difficile de changer la structure de celui-ci et ceci pose par conséquent des problèmes lors de sa réingénierie.

Le mouvement NoSQL lui est plus pragmatique, basé sur des besoins de stockage de données ainsi qu'une liaison plus forte avec les différents langages clients. La plupart

des moteurs de base de données NoSQL n'utilisent pas de schémas prédéfinis à l'avance, d'où l'appellation « schema-less », ce qui signifie sans schéma. Ainsi le moteur de la base de données n'effectue pas de vérification et n'impose pas de contrainte de schéma, les données étant organisées du coté code de client. Toutefois le principe « schema-less » est théorique et ne se vérifie pas entièrement en pratique. Le maintien d'une structure de données homogènes est important, pour des questions d'indexation, de recherche par critère ou tout simplement pour des raisons de logique.

4.1.2 Gestion de la valeur NULL

En SQL, lors de la définition d'une table, il est possible de décider si un attribut peut contenir une valeur ou non lors de l'enregistrement d'un tuple en définissant la colonne à «Null» ou «Not Null». Ceci est utile pour contraindre l'utilisateur à renseigner sur certains attributs que l'administrateur de la base de données considère comme indispensables. Cela dit, dû au fait de sa représentation en deux dimensions (ligne, colonne) et de sa rigidité dans sa structure, il est indispensable de signaler l'absence de valeur à l'aide d'un marqueur Null, ce qui coûte en espace mémoire. Un autre souci du Null est sa gestion dans le langage SQL. Etant donné que le Null n'est pas une valeur, il ne peut donc pas être comparable. En voici une illustration avec la clause «WHERE» :

Produit	
Type	Prix
Viande	10
Chocolat	20
Salade	20
Pain	NULL
Fromage	10

Test	Résultat
Select * From Produit WHERE Prix <> 10	Chocolat, Salade
Select * From Produit WHERE Prix <> 20	Viande, Fromage
Select * From Produit WHERE (Prix <> 20) OR (Prix IS NULL)	Viande, Fromage, Pain

Figure II.1 : Gestion de la valeur Null [11]

L'exemple ci-dessus démontre qu'il faut explicitement citer la valeur Null dans la requête. En effet, le test sur une valeur Null ne retourne pas TRUE ou FALSE mais la valeur UNKNOW. Dans nos 2 premiers tests il va donc sélectionner les produits qui répondront TRUE à leur requête respective. Dans le troisième test il faut explicitement

signaler une valeur Null possible pour que la valeur Null soit prise en considération. Dans des bases de données NoSQL il n'y a pas une gestion spécifique du Null, étant donné que dans des bases de type document ou clé-valeur la notion de schéma n'existe pas, si l'on veut exprimer qu'un attribut n'a pas de valeur il suffit de ne pas le mettre. Même chose dans les bases de type colonne, les colonnes étant dynamiques, elles ne seront présentes qu'en cas de nécessité.

4.2 Transactions

Dans les SGBD Relationnels, une transaction doit respecter les propriétés ACID afin que celle-ci soit validée. Le respect de ces propriétés dans un environnement distribué est pénible et le risque de diminution des performances proportionnelle aux nombres de serveurs est important. Les moteurs NoSQL font l'impasse sur ces propriétés, leur but étant d'augmenter les performances par l'ajout de serveurs à travers la scalabilité horizontale.

4.2.1 Scalabilité

La scalabilité est le terme utilisé pour définir l'aptitude d'un système à maintenir un même niveau de performance face à l'augmentation de charge ou de volumétrie de données, par augmentation des ressources matérielles. Il y a deux façons de rendre un système extensible : la scalabilité horizontale (externe) ainsi que la scalabilité verticale (interne).

Scalabilité horizontale

Le principe de la scalabilité horizontale consiste à simplement rajouter des serveurs en parallèle, c'est la raison pour laquelle on parle de croissance externe. On part d'un serveur basique et on rajoute des nouveaux serveurs pas chers «low cost» pour répondre à l'augmentation de la charge.

Les avantages :

- Achat de serveurs quand le besoin s'en fait sentir.
- Une panne de serveur ne pénalise pas le système.
- Flexibilité du système.
- Rendu financier du à l'achat de matériel pas cher

Scalabilité verticale

Le principe de la scalabilité verticale consiste à modifier les ressources d'un seul serveur, comme par exemple le remplacement du processeur par un modèle plus puissant ou par l'augmentation de la mémoire système RAM. C'est ce que l'on appelle la croissance interne. On a une machine et on l'a fait évoluer dans le temps.

Les inconvénients :

- Aucune tolérance à la panne.
- Achat d'une machine coûteuse.
- Le système est limité dans la flexibilité.
- Impossibilité de faire des mises à jour sans interrompre le système.

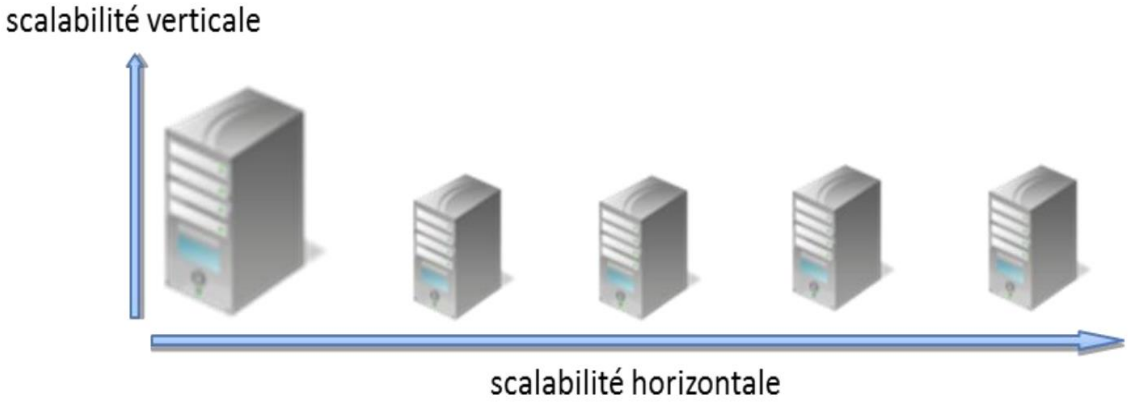

Figure II.2 : Scalabilité horizontale et verticale [12]

4.3 Cohérence contre disponibilité

Dans les environnements distribués, une question très récurrente se pose sur le choix entre la cohérence de données ou la disponibilité de celles-ci. C'est l'un des sujets les plus sensibles entre les deux mondes que constituent le relationnel et le NoSQL. Dans un système de gestion de base de données relationnelle, les différents utilisateurs voient la base de données dans un état cohérent, les données en cours de modification par un utilisateur n'étant pas visibles aux autres utilisateurs, on dit qu'un verrou a été posé. Le maintien de cette cohérence de données dans une base contenue dans un seul serveur

(ou autrement dit architecture centralisée) est tout à fait envisageable (les SGBD Relationnels ont longuement fait leurs preuves), cela devient cependant problématique dans le contexte d'une architecture distribuée.

Les fondateurs du NoSQL tel que Google, Amazon, Facebook etc., dont les besoins en disponibilité priment sur la cohérence des données, ont décidés de privilégier celle-ci au détriment de la cohérence. L'opposition de ces deux propriétés a été présentée par **Eric Brewer** dans ce qu'on appelle aujourd'hui le théorème de CAP [11].

5. THEOREME DE CAP

Le théorème de CAP ou théorème de Brewer a été énoncé pour la première fois en tant que conjecture par le chercheur en informatique **Eric Brewer**. En 2002, deux chercheurs au Institut de technologie du Massachusetts MIT, **Seth Gilbert et Nancy Lynch**, ont formellement démontré la vérifiabilité de la conjecture de Brewer afin d'en faire un théorème établi. Ce théorème énonce qu'une base de données d'un système distribué ne peut garantir les trois contraintes suivantes en même temps, à savoir :

- **Cohérence (Consistency)** : tous les nœuds (serveurs) du système voient exactement les mêmes données au même moment.
- **Haute disponibilité (Availability)** : garantie que toutes les requêtes reçoivent une réponse avec une très courte latence.
- **Tolérant à la partition (Partition Tolerence)** : le système doit être en mesure de répondre de manière correcte à toutes requêtes dans toutes circonstances sauf en cas d'une panne générale du réseau. Autrement dit si une partie du réseau n'est pas opérationnelle, cela n'empêche pas le système de répondre.

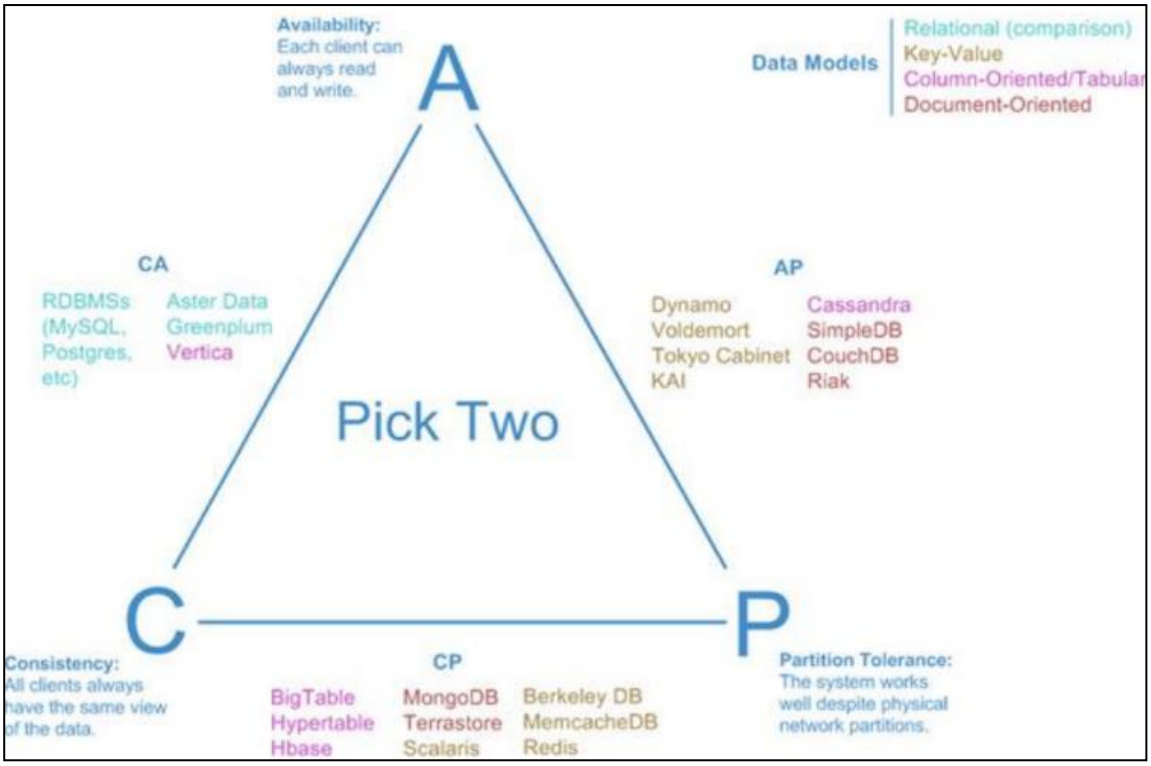

Figure II.3 : guide visuel de Théorème de CAP [13]

Le théorème de CAP dit que pour les systèmes distribués, seules deux de ces trois contraintes peuvent être respectées à un instant T car elles se trouvent en opposition. De ce fait on peut distinguer 3 types de système de bases de données :

- Toutes les bases de données relationnelles sont de type **CA** : elles garantissent la cohérence et la Haute disponibilité des données. Toutes les lectures/écritures des nœuds concernent les mêmes données mais si un problème de réseau apparait, certains nœuds seront désynchronisés au niveau des données et perdront donc la consistance.

Les systèmes NoSQL sont de natures des systèmes distribués, leur but est de renforcer la scalabilité horizontale, il faut pour cela que le principe de tolérance au partitionnement soit respecté, ce qui exige l'abandon soit de la cohérence, soit de la haute disponibilité, du coup on a deux types de bases de données NoSQL :

- Des bases de données de type **CP** : elles garantissent la cohérence et la tolérant à la partition.

- Des bases de données de type **AP** : elles concentrent plus sur la haute disponibilité et la tolérant à la partition, afin de garantir une disponibilité en tout temps et un temps de réponse très rapide et par conséquent abandonnent la cohérence.

Le choix de type de bases de données NoSQL dépend aux besoins de l'utilisateur. Comme il n'existe pas de solution miracle, il faut trancher entre les deux types de bases de données NoSQL à utiliser (CP ou AP).

Pour les entreprises et les sites de e-commerce, le choix est facile, les base de données NoSQL AP sont élues. En effet dans ce cas la consistance peut ne pas être un problème immédiat : si une mise à jour survient, tous les serveurs n'ont pas les mêmes données juste après la mise à jour qui va se propager plus ou moins lentement suivant le type de base de données et les paramètres échéant, donc «au bout d'un moment» les serveurs auront tous effectués cette mise à jour. Prenons l'exemple de Amazon : celui-ci préférera que le client commande un produit qui n'est plus en stock au risque de devoir le contacter pour annuler la transaction plutôt que de voir tout le site ralenti et ainsi perdre de nombreux clients. Prenons un autre exemple : deux utilisateurs qui sont amis sur Facebook, Djamel et Samir. Si Djamel partage une photo sur son « mur » et que Samir au même instant ne la voit pas dans sa « fil d'actualité » elle apparaîtra dans les prochaines secondes. Dans ce cas précis, est-ce un problème pour Samir d'attendre environ une minute pour voir la photo que vient de publier son ami Djamel ?

Les sites ont fait leur choix ! Amazon a d'ailleurs constaté qu'un dixième de seconde de latence (temps d'attente) leur fait diminuer les ventes de 1% [13], et Google a remarqué qu'une demi seconde de latence fait chuter le trafic d'un cinquième.

Pour les données sensibles et stratégiques, on privilège la sécurité et la cohérence des données, dans ce cas les systèmes NoSQL CP sont utilisés [14] [15]. Lorsqu'on est confronté à un système cohérent et tolérant à la partition, on pourrait croire que, du fait que la disponibilité soit compromise, le système ne soit jamais disponible, en d'autres termes un système qui ne sert à rien. Bien entendu, cela n'est pas le cas. La plupart du temps, les systèmes NoSQL qui garantissent une forte cohérence des données sont architecturés en maître/esclave. Cela veut dire que toutes les écritures doivent être faites sur le serveur maître, et en cas de panne de ce dernier, les opérations d'écriture, de

modification et de suppression ne deviennent plus disponibles. Seule la lecture des données sur les serveurs esclaves est possible [12].

6. PROPRIETES B.A.S.E

Les SGBD NoSQL, selon le théorème CAP, qui privilégient la disponibilité ainsi que la tolérance à la partition plutôt que la cohérence, répondent aux propriétés de B.A.S.E. Cette notion de B.A.S.E (**B**asically **A**vailabe, **S**oft State, **E**ventuallty Consistent) est donc à l'opposé d'ACID. Les deux principes ne sont cependant pas mutuellement exclusifs. Le principe B.A.S.E abandonne donc la consistance au profit de ces nouvelles propriétés :

- **Basically Available :** Le système garantie bien la disponibilité dans le même sens que celle du théorème de CAP.
- **Soft-State :** Indique que l'état du système peut changer à mesure que le temps passe, et ce sans action utilisateur. C'est dû à la propriété suivante.
- **Eventually consistent :** Spécifie que le système sera consistent à mesure que le temps passe, à condition qu'il ne reçoive pas une action utilisateur entre temps [16].

BASE se veut comme une alternative à ACID. Ces dernières sont faites pour garantir l'intégrité des données entre elles à tout instant, alors que B.A.S.E se dit garant de l'intégrité dans le temps. C'est-à-dire qu'une base de données deviendra intègre après un certain moment, le temps que l'information se propage entre les différents nœuds.

7. TYPES DE BASES DE DONNEES NOSQL

Dans la mouvance NoSQL, il existe une diversité d'approches classées en quatre catégories. Ces différents systèmes NoSQL utilisent des technologies fortes distinctes. Les différents modèles de structure sont décrits comme suit :

7.1 Bases de données clé-valeur

Le système de base de données associatif de type clé/valeur est probablement le plus connu et le plus basique que comporte la mouvance NoSQL. Son principe est extrêmement simple : il permet de stocker des valeurs, ces valeurs peuvent être de tous type (entier, chaine de caractères, flux binaire, etc.). Chaque objet (valeur) est identifié par une clé unique. Celle-ci représente la seule manière de solliciter la valeur c.-à-d. les

requêtes ne portent que sur la clé associée à cette valeur. L'absence de structure ou de typage a un impact important sur la manière de faire les requêtes à la base de données. Cela vient du fait qu'auparavant toute l'intelligence était portée par les requêtes SQL, alors que dans ce système c'est au niveau de l'applicatif que l'interrogation de la base de données a lieu. Toutefois, la communication avec la base de données se résume aux quatre opérations Create, Read, Update, Delete (CRUD). La plupart des bases de données de type clé/valeur disposent d'une interface HTTP REST qui permet de procéder très facilement à des requêtes depuis n'importe quel langage de développement. Ce protocole est sans état. Il ne garde aucune mémoire de la requête précédente. Le passage, ci-dessous, explique ce qu'est REST [16] :

"REST est un style d'architecture qui repose sur le protocole HTTP : On accède à une ressource (par son URI unique) pour procéder à diverses opérations (GET lecture / POST écriture / PUT modification / DELETE suppression), opérations supportées nativement par HTTP" [17].

Ce système de base de données est conçu pour être très fortement répliqué de manière à augmenter la disponibilité et les performances en lecture et en écriture. Toutefois, la réplication des données est plus ou moins partielle pour trouver un bon compromis entre nombre de serveurs, disponibilité et espace disque.

Ces systèmes sont souvent utilisés comme dépôts de données si toutefois les besoins en termes de requêtes restent de niveau simple et que l'intégrité relationnelle des données est non significative.

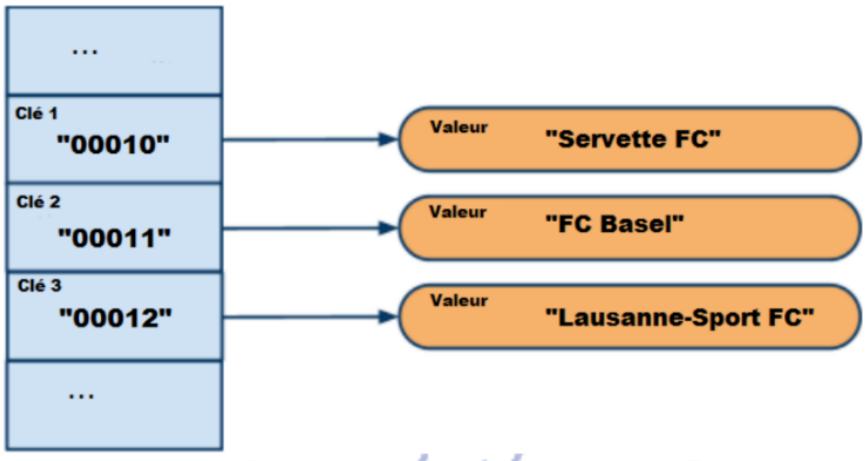

Figure II.4 : Base de données Clé valeur [12]

Pour résumer voici les caractéristiques principales du système associatif :

- Pas d'utilisation du langage SQL.
- Meilleure scalabilité horizontale par rapport aux autres solutions NoSQL.
- Opérations de lecture et écriture très performantes.
- Peut ne pas fournir un support des propriétés ACID
- Peut offrir une architecture distribuée et tolérante aux pannes.

En revanche son point faible c'est :

- Qu'on ne peut agir que sur la valeur dans son entier. On ne peut pas réaliser d'opérations CRUD sur une partie de la valeur.

Exemples Implémentations

DynamoDB : Solution d'Amazon à l'origine de ce type de base. Système ultra scalable basé sur des disques électroniques SSD (pour Solid State Drive) rapides. Design de type AP selon le théorème de CAP mais peut aussi fournir une consistance éventuelle.

Voldemort : Implémentation open-source de Dynamo. Très scalable via un stockage sur disques. Possibilité d'en faire une base embarquée.

7.2 Bases de données orientées colonnes

Les bases de données orientées colonnes ont été conçues par les géants du web afin de faire face à la gestion et au traitement de gros volumes de données s'amplifiant rapidement de jours en jours. Elles intègrent souvent un système de requêtes minimalistes proche du SQL. Bien qu'elles soient abondamment utilisées, il n'existe pas encore de méthode officielle ni de règles définies pour qu'une base de données orientée colonnes soit qualifiée de qualité ou non. Les bases de données orientées colonnes ne diffèrent pas beaucoup des bases SQL classique du point de vue architectural.

La principale différence est la façon dont les données sont stockées en mémoire : dans une base de données relationnelles les données sont stockées par ligne, de ce fait une requête portant sur peu de lignes mais beaucoup de colonnes sera rapide car le système n'aura pas à beaucoup se déplacer en mémoire, alors que dans une base orientée colonnes les données sont stockées par colonnes, de ce fait le système sera beaucoup plus rapide de parcourir les lignes d'une table. Concrètement cela signifie que, contrairement aux bases orientées lignes, les utilisateurs qui souhaitent des

informations précises n'auront pas à s'encombrer des informations inutiles des autres colonnes. En effet les bases orientées lignes se doivent de lire entièrement les lignes avant de sélectionner les colonnes désirées par la requête.

Ce type de structure permet d'être plus évolutif et flexible ; cela vient du fait qu'on peut ajouter à tout moment une colonne ou une super-colonne (une super-colonne est une colonne contenant d'autres colonnes) à n'importe quelle famille de colonnes. Autrement dit, et contrairement à une base de données relationnelle ou les colonnes sont statiques et présentes pour chaque ligne, celles des bases de données orientée colonnes sont dite dynamiques et présentes donc uniquement en cas de nécessité. De plus le stockage d'un « Null » est inexistant. Prenons l'exemple de l'enregistrement d'un client nécessitant son nom, prénom et adresse. Dans une base de données relationnelle il faut donc au préalable créer le schéma (la table) qui permettra de stocker ces informations. Si un client n'a pas d'adresse, la rigidité du modèle relationnel nécessite un marqueur Null, signifiant une absence de valeur qui coûtera toutefois de la place en mémoire. Dans une base de données orientée colonne, la colonne adresse n'existera tout simplement pas [16].

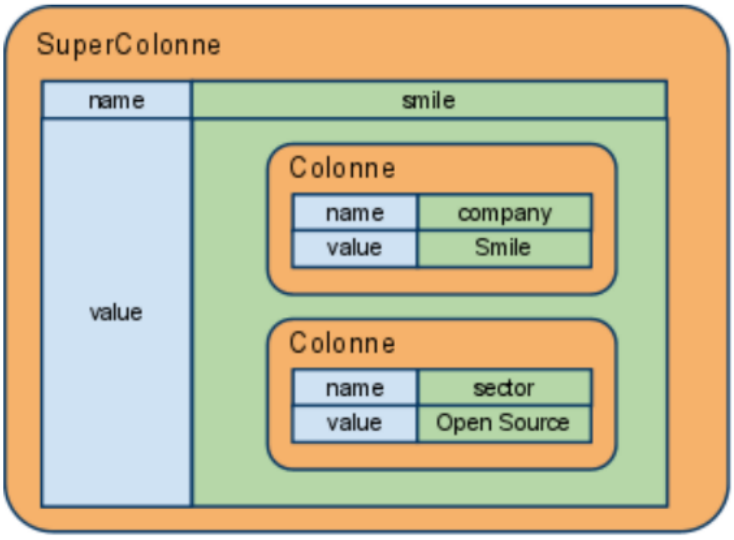

Figure II.5 : Bases de données orientées colonnes [17]

Pour conclure, **Michael Morello** résume bien dans le passage ci-dessous, les avantages et inconvénients de ce type de base, vois quelques-uns :

"**Avantages** :

- L'ajout de données se fait sur une seule dimension (Ndlr : la colonne) ce qui est techniquement plus simple et plus rapide ...
- On devient à la mode en étant "scalable", comme le développement des données ne se fait que sur une seule dimension leurs partitionnements est plus simple à réaliser et on peut les distribuer sur plusieurs serveurs …
- Ajouter une colonne devient trivial car il s'agit au final seulement d'ajouter un nouveau tuple…

Inconvénient :

- …Autrement dit : il y a une seul clé (Ndlr : clé primaire), seul champ véritablement indexé, les recherches sur les autres colonnes seront moins efficaces…
- Si ce type de base de données est très efficace en écriture elle l'est moins en lecture, non seulement on se retrouve avec une seule clé mais on voit aussi assez vite que si l'on veut être performant et pouvoir distribuer les données il va falloir mettre un peu d'ordre en les triant et user d'outil tels que les filtres de M. Bloom afin d'optimiser les recherches.
- Une mise à jour n'écrase pas directement l'ancienne valeur, il faut donc prévoir à un moment ou un autre un processus appelé "compaction" pour réellement supprimer les données… " [18].

Exemples Implémentations

HBase : Utilise un API Java. Adopte un design CA.

Cassandra : Beaucoup d'API disponibles. Adopte un design AP avec consistance éventuelle. Moins performant que HBase sur les insertions de données.

7.3 Bases orientées document

Les bases de données orientées document sont une extension des bases orientées clé-valeur, à la place de stocker une valeur, nous stockons un document. Un document peut contenir plusieurs valeurs et d'autres documents, qui peuvent à leur tour en contenir d'autres et ainsi de suite. Un document peut donc posséder plusieurs niveaux de

profondeur. Tous les documents de niveau 0 sont identifiés par une clé et sont regroupés dans une collection. Des champs et des valeurs associées composent le document. Ces valeurs associées peuvent soit être de type simple (entier, chaine de caractères, date…), soit de type composé, c'est-à-dire de plusieurs couples clé/valeur.

Bien que les documents soient organisés de manière hiérarchique semi structurée dont la structure est libre à l'image de ce que permettent les fichiers de type XML ou JSON, ces bases sont dites « Schemaless » ce qui signifie sans schéma défini. Cela veut tout simplement dire qu'il n'est pas nécessaire de définir au préalable les champs dans le document : on peut très bien en rajouter en cours de développement. Les documents peuvent être très différents les uns des autres au sein de la base. Le fait que les documents soient structurés permet d'effectuer des requêtes sur le contenu des d'un ou plusieurs documents, le système étant partiellement répliqué, la requête va être envoyée à tous les serveurs simultanément et chacun va effectuer celle-ci et rendre des résultats différents, car toutes les données ne sont pas présentes sur tous les serveurs, ces résultats vont être ré-agrégés pour former le résultat finale à la requête. Une interface d'accès HTTP REST est aussi disponible afin de permettre d'effectuer des requêtes sur la base. Mais cette interface est un peu plus complexe que l'interface CRUD des bases clé/valeur :

- Ajout, modification, lecture ou suppression de seulement certains champs dans un document.
- Indexation de champs de document permettant ainsi un accès rapide sans avoir recours uniquement à la clé.
- Requêtes élaborées pouvant inclure des prédicats sur les champs. Ces bases disposent de fonctionnalités très intéressantes en termes de flexibilité du modèle documentaire par rapport aux bases de données relationnelles.

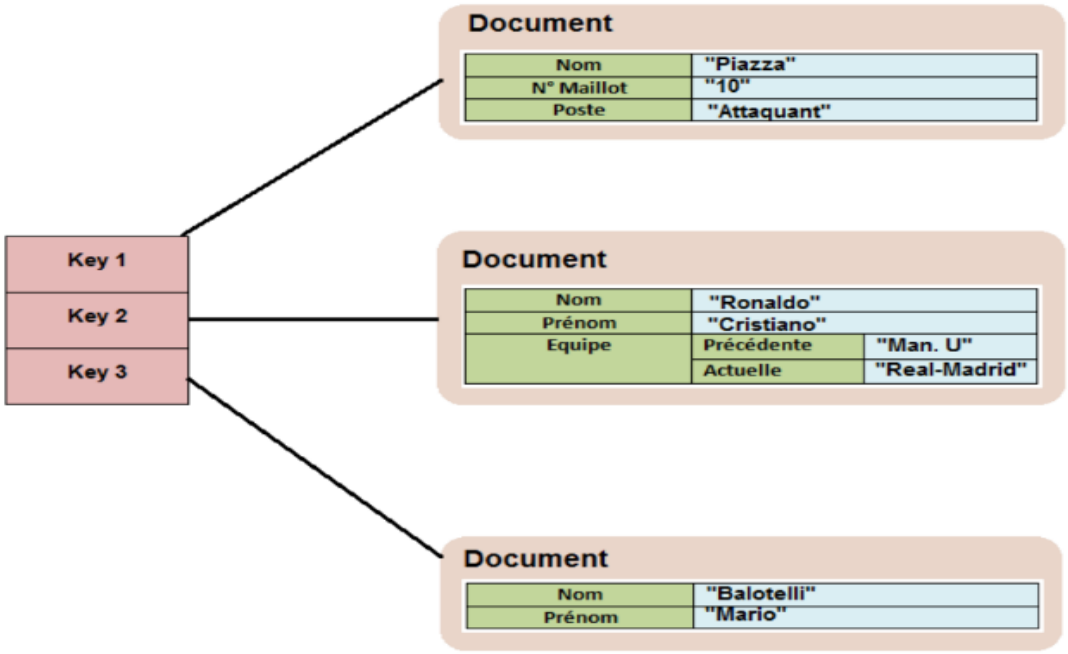

Figure II.6 : Bases de données orientées document [11]

Fort de son expérience dans le domaine, **Michaël Figuière** cite quelques cas d'utilisations particulièrement adaptés au modèle document :

- Persistance de profils utilisateurs, de regroupement de contenus destinés à une même page, de données de sessions.
- Cache d'informations sous une forme structurée.
- Stockage de volumes très importants de données pour lesquelles la modélisation relationnelle aurait entraîné une limitation des possibilités de partitionnement et de réplication [19].

Un autre avantage des bases orientées documents est le versioning. Dans un moteur relationnel, si vous devez changer la structure des tables pour faire évoluer votre application, vous devrez changer votre schéma et faire des choses horribles comme ajouter un numéro de version dans le nom même de la table. Par exemple, si vous avez créé une table "TarifIntervention" en 2011, et que votre direction estime en 2012 que la gestion des tarifs doit changer, vous allez probablement créer une table "TarifIntervention2", parce que vous devez maintenir en parallèle l'ancienne et la nouvelle gestion des tarifs. La gestion d'une telle base de données deviendra de plus en

plus complexe au fil du temps. Avec un moteur de bases de données orienté documents, vous pouvez simplement insérer des nouveaux documents comportant les modifications de structure, et maintenir dans le document un numéro de version, qui sera testé dans l'application cliente pour une prise en charge adaptée [20].

En résumé, les avantages de ce type de base sont :

- Gérer très bien le versioning.
- Permettre de stocker des données non-planes.
- Approprié pour les applications web.
- Permet de structurer la valeur à stocker.
- Ajouter, modifier, lire ou supprimer de seulement certains champs dans un document.
- Indexer de champs de documents permettant ainsi un accès rapide sans avoir recours uniquement à la clé.
- Approprié comme cache d'informations sous une forme structurée.
- Stocker d'importants volumes de données distribuées et répliquées.

Exemples Implémentations

MongoDB : Développé en C++, APIs officielles pour beaucoup de langages, Protocol personnalisé BSON, Réplication master/slave et auto-sharding, Licence commercial et gratuite.

CouchDB : Développé en Erlang, Protocol http, Réplication master/master, Licence Apache.

7.4 Bases orientées graphes

Les bases orientées graphe sont les moins connues de la mouvance NoSQL. Ces bases permettent la modélisation, le stockage ainsi que le traitement de données complexes reliées par des relations sous forme de graphe, elle permet de relier les entités entre elles et de manière optimisée. Ce modèle est composé d'un :

- Moteur de stockage pour les objets : c'est une base documentaire où chaque entité de cette base est nommé nœud.
- Mécanisme qui décrit les arcs : c'est les relations entre les objets, elles contiennent des propriétés de type simple (entier, chaine de caractères, date…).

Les bases de données orientées graphe permettent de résoudre des problèmes liés aux réseaux qui sont très difficiles voire impossibles à résoudre dans une base de données relationnelle. Prenons exemple sur Facebook, si nous devions stocker dans une base relationnelle classique les relations entre les personnes la base serait très grosse et donc longue à parcourir pour rechercher les amis d'une personne.

Le cas d'utilisation classique pour ce type de base de données est le stockage des informations des réseaux sociaux (relations d'amis sur Facebook, connaissances sur LinkedIn), néanmoins les bases orientées graphe peuvent aussi être apte à modéliser de nombreuses autres situations qui ne pourraient être représentées que de manière réductrice dans une base de données relationnelles. La performance de ce type de base de données est très difficile à évaluer car trouver les relations d'une entité est très rapide, mais trouver une entité spécifique si on a aucune connaissance du système est très long, ce système est donc utilisé en parallèle d'une base de données SQL, celle-ci stockera les informations sur les entités et où aller la chercher dans le graphe, alors que la base orientée graphe stockera les relations entre les entités.

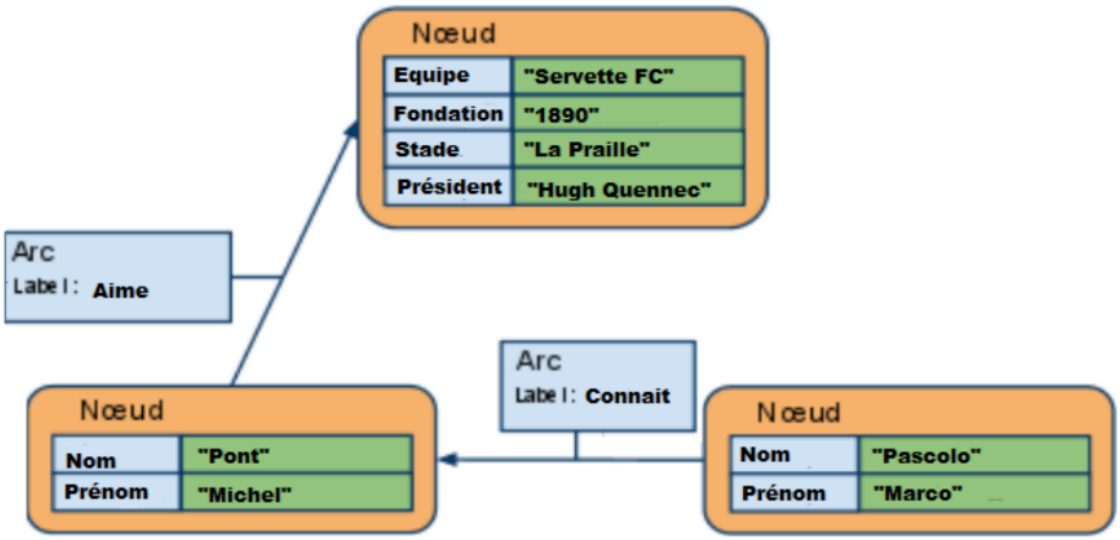

Figure II.7 : Bases de données orientées graphes [17]

Exemples Implémentations

Neo4J : Développé en Java. Supporte beaucoup de langages, Réplication master/slave, Propriétés ACID possibles, Langage de requêtes personnalisé « Cypher ».

Titan : Scalability horizontal très poussée grâce au support d'une base orientée colonnes pour les données, Haute disponibilité avec réplication master/master, Prise en compte d'ACID avec consistance éventuelle, Intégration native avec le framework TinkerPop.

8. AVANTAGES DU NoSQL

8.1 Scalabilité horizontale (élasticité, sharding automatique)

Au lieu d'avoir misé sur une scalabilité horizontale, pendant longtemps les administrateurs de bases de données ont opté pour la scalabilité verticale. Ceci vient du fait qu'il est difficile d'adapter cette stratégie avec une base de données relationnelle. Les systèmes NoSQL ont été conçus pour distribuer les données de manière automatique et transparente sur plusieurs nœuds et ainsi former un cluster pour garder une performance linéaire lors de montées en charge. De plus des serveurs peuvent être ajoutés ou retirés à la volée. Pour faire simple, un système NoSQL ne devrait jamais avoir à redémarrer [13].

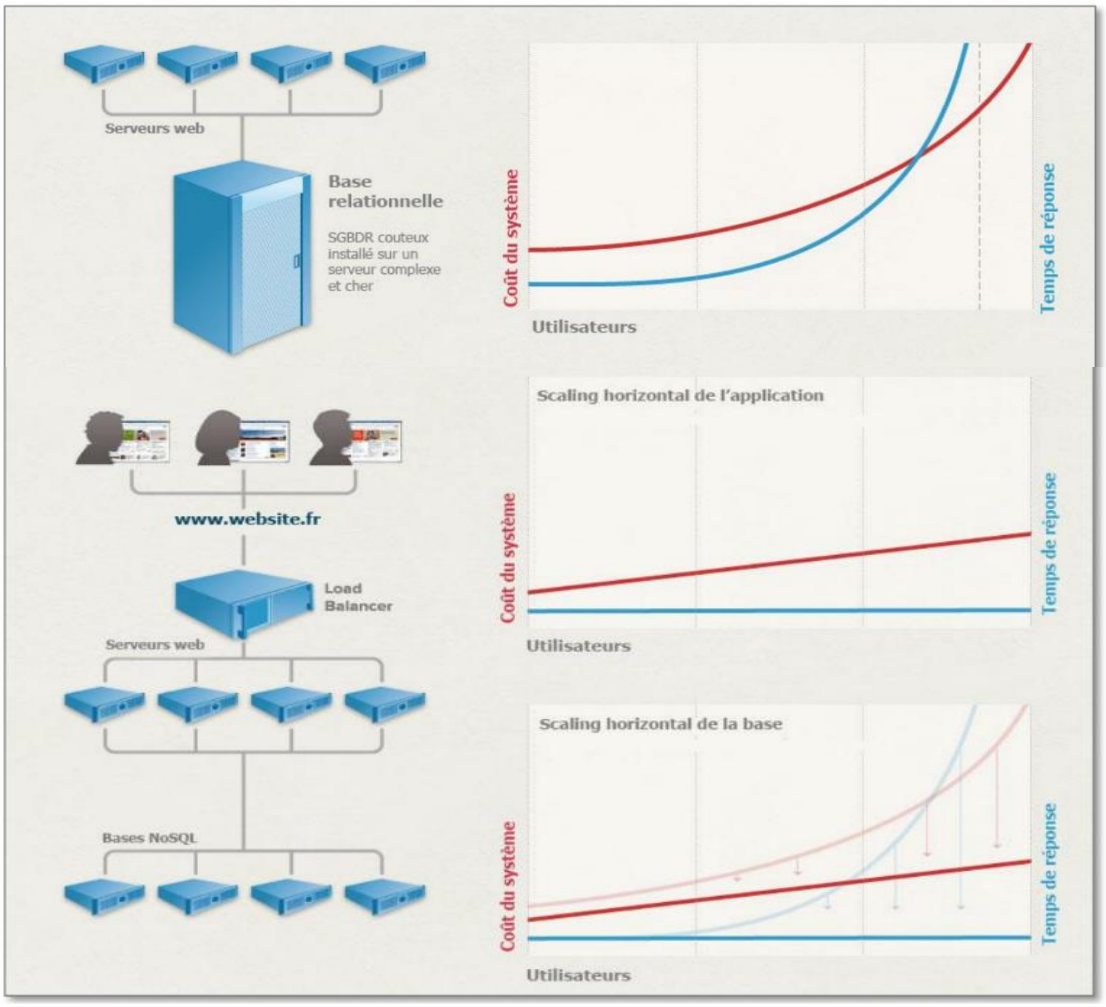

Figure II.8 : Scalabilité verticale contre scalabilité horizontale [13]

8.2 Solution économique

Les bases de données NoSQL ont tendance à utiliser des serveurs bas de gamme dont le coût est moindre afin d'équiper les « clusters », tandis que les SGBD Relationnels, eux, tendent à utiliser des serveurs ultra puissants dont le coût est extrêmement élevé. En plus les SGBD professionnels requièrent l'acquisition d'une licence très chère, quant au SGBD NoSQL sont des solutions open source alors gratuites [12].

8.3 Gestion de gros volume de données

Face aux gros volumes de données, il est devenu quasi impossible, pour un unique serveur de base de données relationnelle, de répondre aux exigences des entreprises en terme de performance. Aujourd'hui, ces gros volumes de données ne sont plus un problème pour les SGBD de type NoSQL, même le plus grand des SGBD Relationnel ne peut rivaliser avec une base NoSQL.

8.4 Administrateur de bases de données moins indispensable

Malgré les nombreuses améliorations vantées pendant des années par les fournisseurs de SGBD Relationnels concernant la gestion de bases de données, un SGBD Relationnel haut de gamme demande une certaine expertise pour sa maintenance. C'est la raison pour laquelle on fait généralement appel à un administrateur de bases de données DBA, ce qui représente donc un certain coût. Il est erroné de dire que la présence d'un DBA n'est plus requise pour gérer une base de données NoSQL. Mais ses tâches seront facilitées notamment grâce à la distribution des données et surtout grâce à la simplicité du schéma de données. Il y aura toujours une personne responsable des performances ainsi que de la disponibilité quand des données critiques sont en jeu [12].

8.5 Schémas de la base ne sont pas figés "shémaless"

Les données sont dorénavant bien plus flexibles car la structure et le type des données peuvent changer à tout moment sans forcément impacter l'application [13]. N'étant pas enfermée dans un seul et unique modèle de données, une base de données NoSQL est beaucoup moins restreinte qu'une base SQL. Les applications NoSQL peuvent donc stocker des données sous n'importe quel format ou structure, et changer de format en production. En fin de compte, cela équivaut à un gain de temps considérable et à une meilleure fiabilité.

L'intérêt des systèmes de stockage NoSQL réside surtout dans les choix d'architecture logicielle qui ont été pris lors de leurs conceptions. Parmi les raisons principales qui ont mené à la création de ces systèmes, on retrouve surtout :

- La possibilité d'utiliser autre chose qu'un schéma fixe sous forme de tableaux dont toutes les propriétés sont fixées à l'avance [21].

8.6 Gestion de données hétérogènes

Grâce à la flexibilité de leur model de données, les bases de données NoSQL peuvent gérer, stocker et traiter les données de type non-structuré (vidéo, images, son, tweet etc…) et semi-structurés (format XML et JSON) et de faire de l'analyse prédictive en temps réel dessus. Ce type d'analyse est de plus en plus recherché par les entreprises. Elle a pour but de fournir des indicateurs clés tactiques (sur des données actuelles souvent non-structurées).

L'objectif, n'est pas seulement d'offrir à la direction des informations uniquement sur l'historique des données de l'entreprise mais aussi sur des flux de données diverses structurées ou non. De plus, l'évolution constante des types de données ainsi que la croissance des données non-structurées rendent impossible leurs intégrations dans un modèle relationnel avec un schéma fixe.

8.7 Fiabilité lors du stockage de données dites critiques

Certaines solutions NoSQL, lors du traitement des données, ont la caractéristique d'être cohérentes dans le temps « eventual consistency ». En effet la cohérence des données est souvent compromise pour faire place à la « haute disponibilité » et à la « tolérance au partitionnement ». La conséquence de cette caractéristique tend à rendre les gens sceptiques voire même réticents quant à l'adoption d'une solution NoSQL dans l'entreprise, car elle ne permet pas de garantir une protection suffisante des données dites critiques.

Cependant, cela n'est pas le cas de toutes les solutions NoSQL. En effet, plusieurs d'entre elles, telles que Riak et Cassandra, permettent de configurer le niveau de cohérence sur les opérations désirées, ce qui représente, en termes de flexibilité, un avantage majeur. Ainsi les développeurs/architectes peuvent décider, suivant la situation, si la cohérence doit être forte ou faible. En termes de cohérence, ces deux solutions peuvent se comporter comme des SGBD relationnels quand c'est nécessaire ou bien peuvent fournir une « cohérence finale » quand le cas le permet [12].

8.8 Haute disponibilité

Dans une entreprise, le système de base de données doit être disponible en permanence, même en cas de panne d'un des serveurs du cluster. La configuration doit être de type « No Single Point of Failure » (aucun point de défaillance). C'est la raison pour laquelle une solution NoSQL, qui intégrera parfaitement l'entreprise, doit posséder le principe de haute disponibilité.

9. RAISONS D'ADOPTATION D'UNE SOLUTION NOSQL

Selon **Julian Browne**, « Lead IT architect » au sein de la célèbre compagnie de télécommunications britannique O2, «Pour des raisons évidentes, il ne vaut vraiment pas la peine d'entreprendre un projet avec n'importe quel produit NoSQL, sauf si vous en avez besoin pour résoudre un problème». Les technologies NoSQL répondent à de nombreux problèmes, mais de manière générale aux deux cas suivant :

- Gros volumes de données :

Problème : Comment prend-t-on des montagnes de données afin de les stocker pour les manipuler et analyser ?

- Montées en charge :

Problème : Comment gère-t-on des millions d'utilisateurs qui accèdent aux données en lecture et écriture simultanément ?

Lors d'une enquête menée par le groupe Unisphere Research concernant les gros volumes de données dans l'entreprise, 87% des entreprises ont affirmé que la croissance des données avait des répercussions négatives sur les performances de leurs applications. C'est une des raisons pour lesquelles nombre d'entre elles ont adopté des solutions NoSQL. Vous n'avez certainement pas les montées en charge de Google ou d'Amazon, mais avez tout de même un nombre conséquent d'utilisateurs connectés simultanément à votre application et génèrent des transactions qui font augmenter le temps de latence. Alors, pour cette raison, une solution NoSQL pourrait être envisagée.

Si aucun de ces deux problèmes n'est identifié dans votre application actuelle ou future, il serait erroné de partir sur des conclusions type « le NoSQL ne s'adapte pas à mon application, donc je continue uniquement avec du relationnel ». En effet, il faut avoir un esprit visionnaire et penser sur le long terme afin de ne pas être victime de son propre succès. Ceci peut engendrer plusieurs coûts qui ne sont pas des moindres : l'exemple le plus concret serait l'achat d'un nouveau serveur plus puissant afin de remplacer celui qui est essoufflé. Le nouveau serveur tient la route ? Oui, mais jusqu'à quand ?

Cependant, une étude menée en décembre 2011, auprès de 1300 développeurs par la société de développement CouchBase - qui fournit la base de données orientée document CouchBase Server - démontre que le plus gros problème qui pousserait les développeurs à adopter une solution NoSQL vient de l'inflexibilité du schéma des bases de données relationnelles [22].

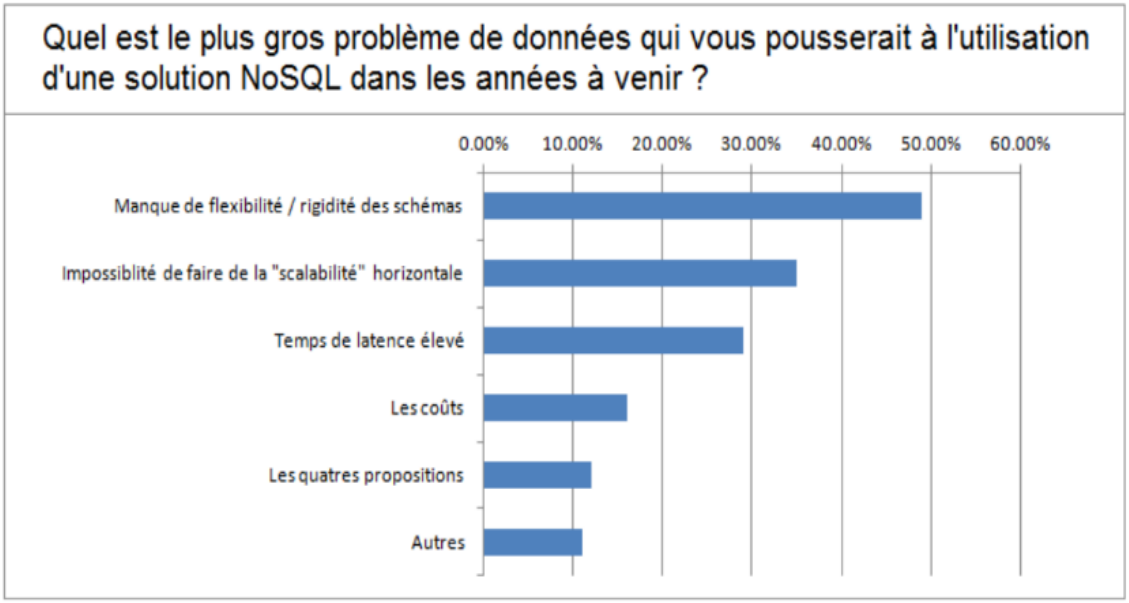

Figure II.9 : Les raisons d'adoption d'une solution NoSQL [22]

En effet, la flexibilité du schéma qui est proposé par les solutions NoSQL représente un grand avantage pour les sociétés dont certains modèles évoluent dans le temps. De plus, il est quasiment impossible, dans la plupart des entreprises, de définir un schéma qui répondrait aux besoins futurs. La gestion des changements dans les tables d'un modèle relationnel est une vraie prise de tête [12].

10. CONCLUSION

Les bases de données NoSQL ont gagné en popularité ces dernières années parce qu'elles ont pu venir à bout de nouveaux problèmes que les bases de données relationnelles étaient incapables de résoudre. On assiste à une explosion des types des systèmes NoSQL disponibles sur le marché, avec chacun ses propres propriétés et cas d'utilisations. Un administrateur doit par conséquent examiner soigneusement le type de bases de données le mieux adapté à ses besoins, car un mauvais choix pourrait avoir de très lourdes conséquences.

CHAPITRE 3 :
Réalisation

PARTIE 1 : Domaine d'étude et choix du système NoSQL

1. INTRODUCTION

Dans la première partie de ce chapitre, nous allons présenter le domaine dans lequel notre étude va être développée à savoir le service de la biométrie de la daïra de Sebdou et sa gestion des passeports biométriques sous Oracle, ensuite nous allons donner les motivations et les critères les plus importants qui ont favorisé le choix du système NoSQL MongoDB pour héberger les données de la base de données des passeports biométriques.

Dans la seconde partie, nous allons présenter brièvement la base de données en question, puis l'environnement de travail va être recensé pour ensuite expliquer la démarche à suivre pour aboutir à la mise en œuvre de la plateforme distribuée ainsi que l'implémentation de la migration des données biométriques.

Par la suite, nous réalisons l'installation de MongoDB et nous créons notre base de données MongoDB hébergeant les données de la biométrie une fois migrées. Ensuite, nous créons l'environnement distribué avec les nœuds nécessaires de MongoDB.

Une fois la plateforme préparée, nous présentons notre application développée appelée *OracleJson*. Cette application va réaliser la transformation des données Oracle vers des fichiers JSON.

Dans un dernier point, nous exploitons la nouvelle base migrée avec la distribution entre les nœuds (*Sharding*), et nous présentons certains résultats et bénéfices de ce travail en comparant les performances obtenus.

Nous terminons ce chapitre par une conclusion sur les résultats obtenus.

2. DOMAINE D'ETUDE

Pour illustration pratique de notre projet, nous allons utiliser le système d'information de la daïra de Sebdou. Présentons d'abord l'activité puis son système.

2.1 Daïra de Sebdou

La Daïra est une subdivision de la wilaya dans l'administration territoriale algérienne , elle regroupe plusieurs communes à savoir Sebdou, El-aricha et El-gor. Elle

a comme tâche l'émission des passeports internationaux, les permis de conduire et les cartes d'identité nationales pour les citoyens résidant dans son territoire, ainsi que le contrôle des travaux effectués par les différents services administratifs tels que les communes, les services techniques, etc.

La daïra aide l'administration dans ses fonctions comme étant un service externe de la wilaya qui la représente dans les activités des communes et assiste ces dernières dans leurs tâches. Juridiquement, la daïra représente un moyen de déconcentration puisqu'elle n'est dotée ni de personnalité morale ni d'autonomie financière. Le chef de daïra assiste le wali dans ses missions (concernant le contrôle, le suivi et l'animation des communes rattachées à chaque daïra).

La daïra en tant que organe déconcentré constitue le cadre territorial qui englobe quelques communes. Elle est la tutelle des communes et les contrôle. Le chef de daïra anime, oriente, coordonne et contrôle l'activité des communes qui lui sont rattachées. Il assiste le wali et est sous son autorité. Il est chargé d'encourager toute initiative individuelle ou collective des communes qu'il anime, et donne un avis consultatif sur la nomination des responsables des structures techniques de daïra relevant de l'administration de l'Etat. Le secrétaire général a pour mission de suivre les tâches qu'effectuent les structures rattachées à la daïra et coordonne entre eux.

2.2 Service des passeports biométrique

La daïra est composée de plusieurs services, parmi lesquels on trouve le service des passeports biométriques qui a pour tâche l'émission des passeports internationaux au citoyen algériens.

2.2.1 Organisation de Service

Ce service se compose de 05 bureaux (voir figure III.1) :

- **Bureau de Vérification :** Chargée de l'Accueil et Vérification de la composition du dossier physique, puis la création du dossier électronique de la demande et génération automatique du NIN (Numéro d'Identifiant National), par la suite il fait la numérisation de la photo d'identité du et le 12S du demandeur et enfin l'édition de la fiche du dossier.
- **Bureau de Saisie** : Chargée de Compléter la Saisie avec le reste des données renseignées sur le formulaire.

- Bureau de Certification : Chargée du contrôle des données saisies par rapport au dossier physique et la correction d'erreurs éventuelles ainsi que la certification du dossier.

- Bureau de capture des données biométriques : Chargé de Capturé les données biométriques : Les empreintes digitale, la photo et la signature numérique.

- la Salle Machine : contient le matériel suivant :

a. Deux serveurs (systèmes installés : Windows Server et ORACLE)
b. Un Poste de transfert des données.
c. L'armoire de brassage de réseaux.

2.2.2 Principaux objectifs de la plateforme technique déployée au niveau de la daïra :

- Gérer les rendez-vous de traitement des demandes de passeport via une application informatique.
- Constituer une base de données locale contenant toutes les informations sur les dossiers de demande des passeports.
- respecter un processus de traitement des demandes.
- Transférer les données vers le site central.
- Assurer la continuité de fonctionnement du système (mettre en place deux serveurs principal et secondaire).
- Appliquer une stratégie de sauvegarde du système et des données.

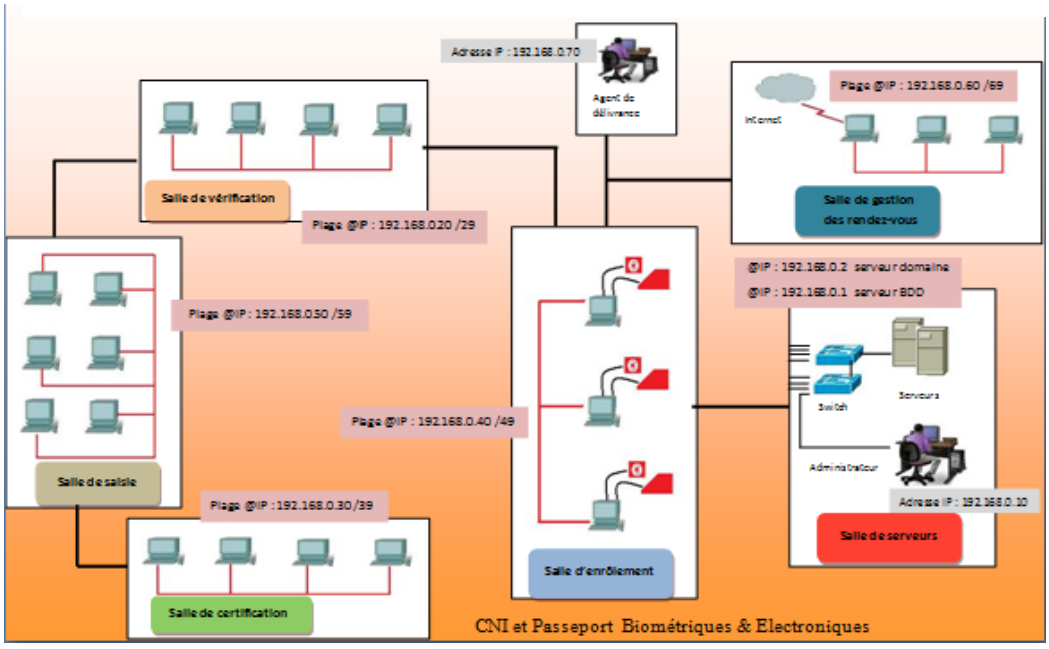

Figure III.1 : Architecture de service biométrie

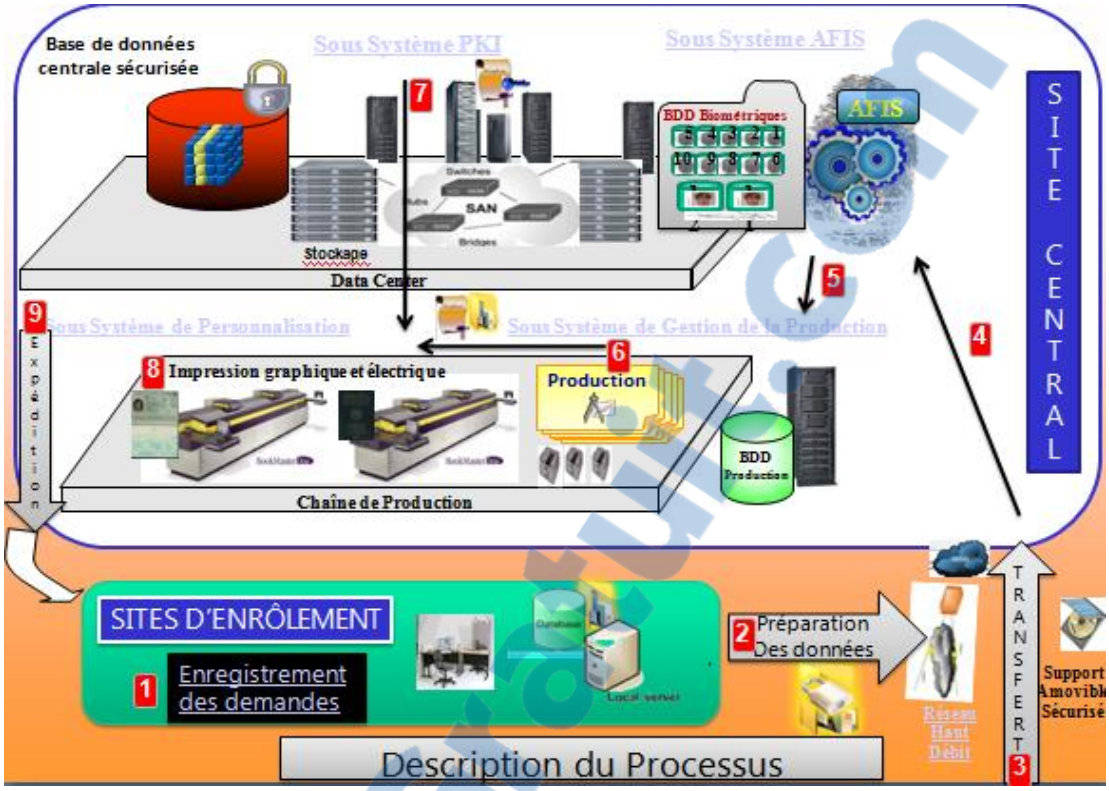

Figure III.2 : Description du processus d'établissement des passeports biométrique

2.3 Motivation du choix d'un tel domaine

Notre choix pour ce champ d'étude à savoir la biométrie a été arrêtée principalement pour les motifs suivants :

- L'importance de ce domaine stratégique et donc tout résultat serait appréciable par la suite pour l'administration algérienne.
- Le volume important des données prises en charge actuellement par le système, le rend candidat pour un éventuel traitement de Big Data ou déploiement sous Cloud.
- L'accroissement rapide des masses de données manipulées.
- Suite au nombre de demande dans des moments précis ou saisonnière, il y a toujours des risques de pannes.
- Le développement de cette application peut s'avérer utile pour étaler l'étude à d'autres domaines (Etat civil, Permis de conduire, Cartes grises, Dossiers médicales etc).

3. JUSTIFICATION DU CHOIX DE MONGODB

Après avoir présenté le domaine dans lequel les expérimentations seront faites, on va justifier notre choix du type de système NoSQL le mieux adapté à cette étude de cas.

« Le marché est aujourd'hui rempli de bases de données NoSQL – je pense que nous sommes confrontés à deux ou trois d'entre elles tous les jours », Ironise Michael Simone, en charge de l'ingénierie de la plateforme CitiData de Citigroup [23].

En raison du nombre important de bases NoSQL, le choix d'une solution NoSQL se voit un vrai casse-tête, donc pour y remédier, nous commençons par une bonne compréhension des différents types de bases de données NoSQL.

3.1 Convenance des types des systèmes NoSQL au projet

On peut classer les systèmes NoSQL en quatre catégories : les bases de données orientées document, les bases clé/valeur, les bases en colonnes et les bases orientées graphes. Elles ont toutes un point commun : le support de modèles plus flexibles et dynamiques que ceux réalisés avec les bases de données relationnelles traditionnelles, mais, chaque type de base NoSQL correspond à des usages spécifiques.

3.1.1 bases clé/valeur :

Les bases de données clé/valeur sont la forme la plus simple des bases NoSQL. Elles associent des clés uniques à des valeurs dans des données, avec pour objectif de renforcer fortement les performances des applications reposant sur des jeux de données relativement simple. « Les bases clé/valeur sont très légères », explique Joe Caserta, président de Caserta Concepts, une société de services techniques et de conseil. « Nous pouvons effectuer des recherches en quelques secondes ».

Une base de données clé/valeur convient parfaitement à un **accès aux données au moyen d'une clé**, comme dans le cas de la recherche d'un livre d'après son numéro international normalisé (ISBN). Dans cet exemple, le numéro ISBN est la clé et toutes les autres informations concernant le livre sont la valeur [23].

Exemples de bases clé/valeur : Aerospike, Redis et Riak.

3.1.2 Bases de données orientées document

Les bases de données orientées document sont une évolution des bases de données clé/valeur, Elle ressemble à celles-ci, mais affiche une structure imposée aux données : l'ensemble de données qui formait la valeur est ici décrit dans une structure de document, en général au format JSON ou XML. Elle peut être interrogée sur tous les

composants du document, alors qu'une base de données clé/valeur ne peut être interrogée que sur sa clé.

Elles sont souvent utilisées dans les **systèmes de gestion de contenus**, ainsi que pour **collecter et traiter des données à partir des applications mobiles ou Web** à fort trafic. Les schémas dynamiques des bases de données documentaires se sont avérés bien adaptés à **des applications qui évoluent dans le temps**, « Nous avons découvert que nous pouvions modéliser tout ce qui nous arrivait », explique ironise Michael Simone [24].

Exemples de bases documentaires : MongoDB, CouchDB, MarkLogic…

3.1.3 Bases en colonnes

Les bases de données en colonnes offrent des hauts niveaux de performance et de dimensionnement lorsqu'il faut traiter et parcourir d'importants jeux de données, ce sont les types NoSQL les plus performants. Une base de données orientée colonnes convient aux données faisant l'objet de peu d'écritures, pour lesquelles les attributs ACID (atomicité, cohérence, isolation, durabilité) ne sont pas impératifs et dont le schéma est variable.

Elles sont adaptées pour **la recherche sur Internet, les applications Web à très grande échelle** ainsi que **les applications analytiques** capables de traiter des péta-octets de données [24].

Accumulo, Cassandra et HBase en sont des exemples de bases en colonnes.

3.1.4 Bases orientées graphes

Les bases de données en graphes, comme InfiniteGraph et Neo4j, stockent des éléments de données dans des structures "en graphes" et permettent de créer des associations entre eux pour, au final, servir de socle à **des moteurs de recommandations ou des réseaux sociaux**. Par exemple, une technologie de graphes peut être utilisée pour identifier des relations entre différentes personnes via leurs centres d'intérêts, illustre Alex Trofymenko, de chez HealthUnlocked, une société spécialisée dans l'information sur la santé. Son équipe s'est adossée à Neo4j pour établir ce type de corrélations. « Nous pouvons obtenir de nombreuses informations

d'une base en graphes. Par exemple, qu'un utilisateur est très intéressé à la fois par le diabète et le sport » [23].

Pour cette première série, nous remarquons que les bases clé/valeurs comme Aerospike, Redis et Riak et les bases document comme MongoDB, CouchDB, MarkLogic sont les plus adaptées à notre besoin.

3.2 Critère favorisant la consistance dans le théorème de CAP

Dans le chapitre précédent, nous avons évoqué les 3 types de systèmes conformément au théorème CAP, à savoir CP, AP, et AC.

Une base de données relationnelle garantit toujours la consistance. De ce fait, la migration d'une base relationnelle transactionnelle (contenant des données importantes) vers NoSQL favorise un système NoSQL de type CP en priorité et non pas AP.
Par contre, la migration d'une base relationnelle ou autre informationnelle (comme les sites Web) favorise un système NoSQL de type AP [25]. On basant sur la Figure II.3 (guide visuel de Théorème de CAP) du chapitre II, nous constatons que les bases CP sont MongoDB, Redis, BigTable, Hbase, MemcacheDB, BerkeleyDB, Terrastore, Scalaris, Hypertable.

3.3 Critère de dernier classement annuel des SGBD

Le classement des bases de données **par DB-Engines Ranking** est effectué selon la popularité dont les critères sont les suivants :

– Nombre de mentions sur les sites Web mesurés comme les résultats de requêtes sur les moteurs Google et Bing.

– Intérêt général pour la base de données mesuré par la fréquence des recherches dans Google Trends.

– Fréquence des discussions techniques sur les bases de données avec utilisation des sites de – Questions/Réponses.

– Nombre d'emplois proposés dans lequel la base de données est mentionnée.

– Nombre de réseaux professionnels dans lesquels la base de données est mentionnée.

– Intérêt dans les réseaux sociaux : nombre de tweets dans lesquels sont mentionnées les bases de données.

DB-Engine ne prend pas en compte le nombre d'installations des bases de données. Il ne s'agit donc pas de mesure de bases installées mais il se présente comme un indicateur de tendance. À partir d'une formule interne combinant ces paramètres, le BD-Engines Ranking a pu attribuer des scores aux différents systèmes de gestion de base de données. À l'issue du classement, les SGBDR dominent le podium : Oracle se positionne comme le système le plus populaire de l'année 2016. Il est suivi par MySQL en 2 eme position, Microsoft SQL Server est en 3 eme position, MongoDB en 4eme position est le premier SGBD NoSQL, cassandra en 7eme position et Redis en 10 eme position.

309 systems in ranking, August 2016

Rank Aug 2016	Rank Jul 2016	Rank Aug 2015	DBMS	Database Model	Score Aug 2016	Score Jul 2016	Score Aug 2015
1.	1.	1.	Oracle	Relational DBMS	1427.72	-13.81	-25.30
2.	2.	2.	MySQL	Relational DBMS	1357.03	-6.25	+65.00
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1205.04	+12.16	+96.39
4.	4.	4.	MongoDB	Document store	318.49	+3.49	+23.84
5.	5.	5.	PostgreSQL	Relational DBMS	315.25	+4.10	+33.39
6.	6.	6.	DB2	Relational DBMS	185.89	+0.81	-15.35
7.	7.	↑ 8.	Cassandra	Wide column store	130.24	-0.47	+16.24
8.	8.	↓ 7.	Microsoft Access	Relational DBMS	124.05	-0.85	-20.15
9.	9.	9.	SQLite	Relational DBMS	109.86	+1.32	+4.04
10.	10.	10.	Redis	Key-value store	107.32	-0.71	+8.51

Figure III.3 : Top 10 des SGBD les plus populaires d'après DB-Engines (aout 2016)

Un autre classement élaboré par **Gartner** en Octobre 2015 pour les SGBD (tout type confondu) par rapport à 2 axes : l'habilité d'exécution (Ability To Execute) et la vision complète future (Completeness Of Vision). Ce classement (présenté dans Figure III.3) confirme les résultats obtenus précédemment : Oracle et Microsoft SQL Server leaders SGBD Relationnel, et MongoDB la meilleure des NoSQL.

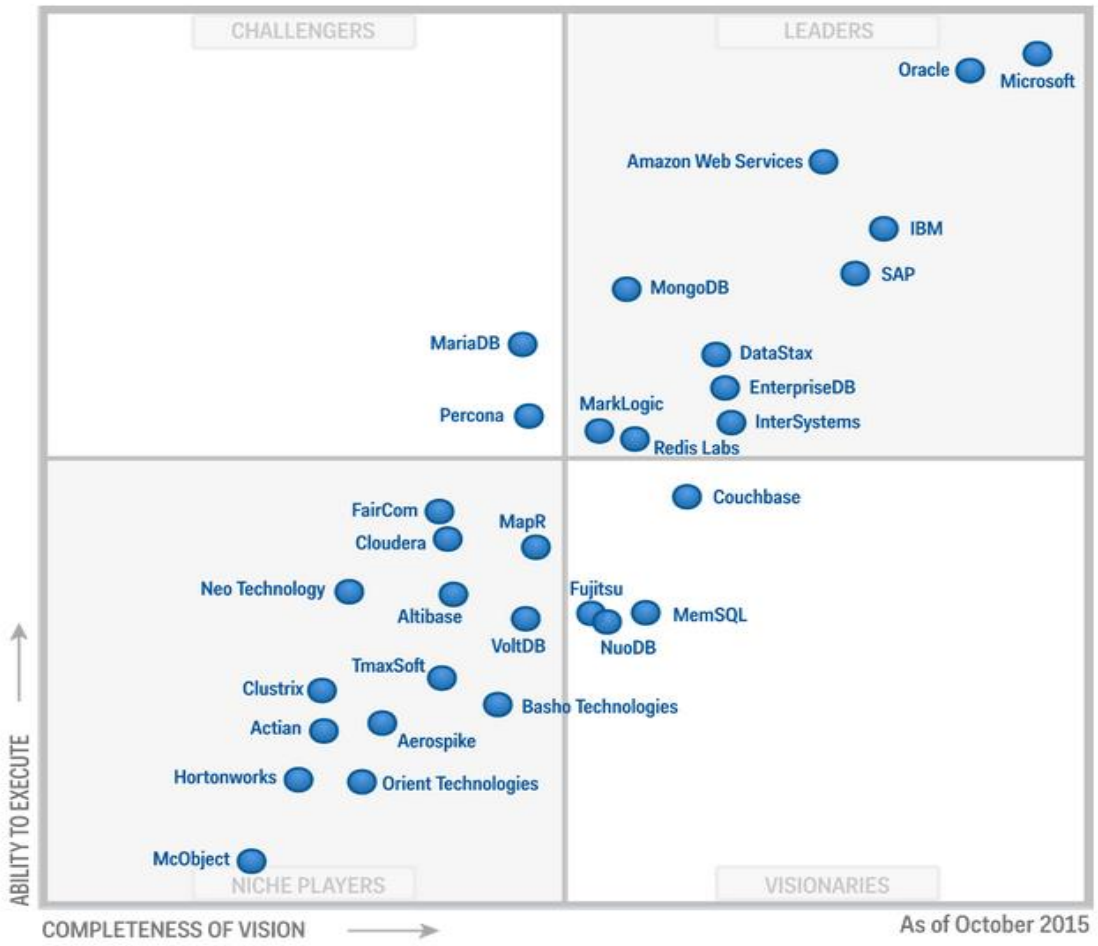

Figure III.4 : Rapport Gartner sur systèmes de bases de données (octobre 2015)

3.4 Synthèse

On basant sur les SGBD NoSQL CP qui convient à notre projet, deux entre eux sortent du lot : MongoDB et Redis, mais on ajoutant d'autres critères comme la maturité de la solution, la popularité, la documentation fournie, la grandeur de la communauté pour garantir la durabilité, la facilité d'implémentation et d'utilisation, le rythme d'évolution, le dynamisme,… notre choix sera très clairement et sans aucun doute pour Mongodb.

Si MongoDB est tant utilisée c'est qu'elle offre un bon compromis entre tous ces critères. Sa modélisation documentaire s'adapte parfaitement à de nombreuses applications. Toutes les applications de gestion de dossiers, factures, formulaires de saisie, commandes ou produits le modélisent plus facilement avec une base

documentaire. MongoDB offre un grand nombre de fonctions qui la rendent très polyvalente.

De plus, son éditeur MongoDB Inc, travaille intensivement à construire une communauté. La documentation est abondante et à jour. Il organise, notamment, des formations en ligne gratuites pour les nouveaux utilisateurs. Bref, s'il est difficile de pronostiquer les bases qui survivront à terme, MongoDB fait partie des candidats les mieux placés. Ses performances sont excellentes et sont estimées dans la plupart des benchmarks comme supérieures à la plupart des autres SGBD document équivalents. MongoDB est utilisé aujourd'hui par de nombreux gros acteurs, comme par exemple : Yandex, Ebay, McAfee, Adobe, Craiglist, Electronic Arts...

Et pour conclure, voici quelque avis d'experts concernant cette base de données et ses points forts :

« Pour 80 % des applications existantes aujourd'hui dans le monde, une base de données orientée documents comme MongoDB est tout indiquée. Et cette part va continuer de croître… » Yann Aubry, directeur France et Europe du Sud de MongoDB [26].

« MongoDB est à mi-chemin entre le SGBD relationnel (Oracle, MySQL) et les technologies clés-valeurs. Allier robustesse et simplicité d'utilisation. Performances brutes et richesse fonctionnelle. MongoDB propose son propre moteur permettant de faire des requêtes, agrégations, jobs MapReduce, … Cela en fait une base de données solide, offrant des performances impressionnantes. Elle est déjà utilisée par de grands noms comme Sourceforge, EA, le NewYork Times et bien d'autres. Elle me semble être particulièrement bien adaptée pour les applications web. Certains prédisent même qu'elle pourrait prendre la place de MySQL dans ce domaine. » Guillaume BODIOU, Manager et consultant a Arismore [27].

« Enfin le vrai intérêt de MongoDB selon moi, surtout par rapport à Riak ou CouchDB, c'est sa gestion de requêtages. On peut requêter de façon très fine grâce à une grande quantité de mots clés. On peut donc faire des requêtes aussi riches qu'en SQL, mais tout ça sur une base de données orientée Documents. » Philippe Rigaux, professeur au département d'informatique au CNAM [28].

4. MongoDB

4.1 Introduction

Les bases relationnelles sont adaptées à des informations bien structurées, décomposables en unités simples (chaînes de caractères, numériques), et représentables sous forme de tableaux. Beaucoup de données ne satisfont pas ces critères : leur structure est complexe, variable, et elles ne se décomposent par aisément en attributs élémentaires.

Les bases relationnelles répondent à cette question en multipliant le nombre de tables, et de lignes dans ces tables, pour représenter ce qui constitue conceptuellement un même "objet". C'est la fameuse normalisation (relationnelle) qui impose, pour reconstituer l'information complète, d'effectuer une ou plusieurs jointures assemblant les lignes stockées dans plusieurs tables.

Cette approche, qui a fait ses preuves, ne convient cependant pas dans certains cas, surtout pour l'échange de données dans un environnement distribué. Pour cela, il y a des modes de représentation plus riches permettant la réunion, en une seule structure, de toutes les informations relatives à un même objet conceptuel. C'est ce que nous appellerons document.

4.2 Représentation de Documents

Deux formats sont maintenant bien établis pour représenter les documents : XML et JSON. Ces formats sont, entre autre, conçus pour que le codage des documents soit adapté aux échanges dans un environnement distribué. Un document en JSON ou XML peut être transféré par réseau entre deux machines sans perte d'information et sans problème de codage/décodage.

JSON par rapport à XML est beaucoup plus concis, simple dans sa définition et très facile à associer à un langage de programmation. JSON est l'acronyme de JavaScript Object Notation. Comme cette expression le suggère, il a été initialement créé pour la sérialisation et l'échange d'objets Javascript entre deux applications. Cela dit, JSON est un format texte indépendant du langage de programmation utilisé pour le manipuler. C'est le format de données principal que nous aurons à manipuler. Il est utilisé comme modèle de données natif dans des systèmes NoSQL comme MongoDB ou CouchDB.

Commençons par la structure de base : **les paires (clé, valeur).** En voici un exemple, codé en JSON.

```
"nom" : "Bachir"
```

La construction (clé, valeur) permet de créer des **agrégats.** Voici un exemple JSON, dans lequel les agrégats sont codés avec des accolades ouvrante/fermante.

```
{"nom" : "Bachir Ahmed", "tél" : 0779815245, "email" : "bachirsdh@yahoo.fr"}
```

Nous avons parlé de la nécessité de composer des structures comme condition essentielle pour obtenir une puissance de représentation suffisante. On peut alors imbriquer les agrégats les uns dans les autres, comme le montre l'exemple ci-dessous.

```
{"nom" : {
            "prénom" : "Ahmed",
            "famille" : "Bachir"
          },
 "tél" : 0779815245,
 "email" : "bachirsdh@yahoo.fr"
}
```

Continuons à explorer les structures de base. Un tableau est une valeur constituée d'une séquence de valeurs. Les tableaux sont connus en JSON et codés avec des crochets ouvrant/fermant.

```
{"nom" : "Bachir Ahmed", "téls" : [0779815245, 0555876249]}
```

Et voilà pour l'essentiel. On peut faire des agrégats d'agrégats d'agrégats, sans limitation de profondeur. On peut combiner des agrégats, des listes et des paires (clé, valeur) pour créer des structures riches et flexibles. Enfin, le codage JSON donne à ce que nous appellerons maintenant "documents" l'autonomie et la robustesse (pour des transferts sur le réseau) attendu.

4.3 Collections de documents

Comment représenter des **collections** (ensembles) de documents ? En relationnel, l'équivalent d'un document est un **tuple** (une ligne), et les ensembles sont représentés dans des tables. La représentation arborescente est très puissante, plus puissante que la représentation relationnelle. Voici un exemple de table.

id	nom	prénom
1	El Hassani	Hassan
2	Kateb	Mustapha
3	Hamina	Mohamed Lakhdar
4	Chouikh	Mohamed
5	Kouiret	Sid Ali

Tableau III.1 : Table des Artistes

Il est clair qu'il est très facile de représenter une table par un document structuré :

```
[
artiste : { "id" : 1, "nom" : "El Hassani", "prénom" : "Hassan" },
artiste : { "id" : 2, "nom" : "Kateb", "prénom" : "Mustapha" },
artiste : { "id" : 3, "nom" : "Hamina", "prénom" : "Mohamed Lakhdar" },
artiste : { "id" : 4, "nom" : "Chouikh", "prénom" : "Mohamed" },
artiste : { "id" : 5, "nom" : "Kouiret", "prénom" : "Sid Ali" }
]
```

Cette représentation, pour des données régulières, n'est pas du tout efficace à cause de la redondance de l'auto description : à chaque fois on répète le nom des clés. En revanche, une représentation arborescente JSON est plus appropriée pour des données de structure complexe et surtout flexible.

Grâce à l'imbrication des structures, il est possible de représenter des données avec une complexité arbitraire. On a imbriqué un objet dans un autre, ce qui ouvre la voie à la représentation d'une entité par un unique document complet. Prenons l'exemple du film "Chronique des années de braise" et son réalisateur et ses acteurs. Voici sa représentation en relationnel. Outre la table des artistes ci-dessus, il nous faut aussi une table des films.

id	titre	Année de sortie	idRéal
1	Le Vent des Aurès	1966	3
2	Chronique des années de braise	1975	3

Tableau III.2 : Table des Films

idFilm	idArtiste	rôle
2	1	Slimane
2	3	Le conteur fou
2	5	Moudjahid

Tableau III.3 : Table des Rôles

La représentation d'une même "entité" (un film) dans plusieurs tables a deux conséquences importantes qui motivent (parfois) le recours à une représentation par document structuré

1. il faut effectuer plusieurs écritures pour une même entité, et donc appliquer une transaction pour garantir la cohérence des mises à jour,
2. il faut faire des jointures pour reconstituer l'information.

Par exemple, pour reconstituer l'ensemble du film "Chronique des années de braise", il faut suivre les références entre clés primaires et clés étrangères. C'est ce qui permet de voir que Mohamed Lakhdar Hamina (clé = 3) est réalisateur de Chronique des années de braise (clé étrangère idRéal dans la table Film) et joue également un rôle (clé étrangère idArtiste dans la table Rôle).

Les transactions et les jointures sont deux mécanismes assez compliqués à mettre en œuvre dans un environnement distribué. Pour éviter ces deux obstacles, les systèmes NoSQL codent les entités dans des documents autonomes, que l'on peut écrire en une seule opération et qui ne nécessitent pas de jointure. Voici un exemple de document assez complet pour représenter un film.

```
{
"titre" : "Chronique des années de braise",
"année de sortie" : "1975",
"genre" : "Drame",
"pays" : "Algérie",
"réalisateur" : {
                "nom" : "Hamina",
                "prénom" : "Mohamed Lakhdar",
                "date de naissance" :  "1930" },
"acteurs" : [
              { "nom" : "El Hassani",
                "prénom" : "Hassan",
                "date de naissance" : "1916",
                "rôle" : "Slimane" },
              { "nom" : " Kouiret",
                "prénom" : "Sid Ali",
                "date de naissance" : "1933",
                "rôle" : "Moudjahid"},
              { "nom" : "Hamina",
                "prénom" : "Mohamed Lakhdar",
                "date de naissance" :  "1930" ,
                "rôle" : "Le conteur fou"}
            ]
}
```

En observant bien le document ci-dessus, on réalise rapidement qu'il introduit cependant deux problèmes importants.

– Chemin d'accès privilégié : la représentation des films et des artistes n'est pas équilibrée ; les films apparaissent près de la racine des documents, les artistes sont enfouis dans les profondeurs ; l'accès aux films est donc privilégié (on ne peut pas accéder aux artistes sans passer par eux) ce qui peut on non convenir à l'application.

– Redondance : la même information doit être représentée plusieurs fois, ce qui est tout à fait fâcheux (Mohamed Lakhdar Hamina est représenté deux fois).

Une solution est de créer deux collections et d'introduire des références entre documents. Voici une partie de la collection pour films avec des documents de la forme :

```
{
"titre" : "Chronique des années de braise",
"année de sortie" : "1975",
"genre" : "Drame",
"pays" : "Algérie",
"réalisateur" : "artiste : 3",
"acteurs" : [ { "artiste :1", "role" : " Slimane "},
              { "artiste :3", "role" : "Le conteur fou" },
              { "artiste :5", "role" : " Moudjahid " }]
}
```

Notez que la référence à l'artiste 3 apparaît deux fois. Les artistes sont dans une autre collection.

```
[
    { "_id" : " 1", "nom" : " El Hassani ", " prénom" : " Hassan "},
    { "_id" : " 3", "nom" : "Hamina ", " prénom" : " Mohamed Lakhdar"},
    { "_id" : " 5", "nom" : "Kouiret", " prénom" : " Sid Ali"}
]
```

4.4 Requêtes avec Mongodb

Le langage de requête sur des collections est spécifique à MongoDB. Essentiellement, c'est un langage de recherche dit "par motif" (pattern). Il consiste à interroger une collection en donnant un objet (le "motif/pattern", en JSON) dont chaque attribut est interprété comme une contrainte sur la structure des objets à rechercher.

4.4.1 Opérations CRUD

Nous allons aborder maintenant les différentes opérations CRUD pour : CREATE, READ, UPDATE, DELETE (comme INSERT, SELECT, UPDATE et DELETE en SQL normalisé). Ces opérations vont nous permettre de réaliser la majorité des opérations sur les données (les documents). Tout comme dans un Shell de MySQL ou tout autre SGBD, nous pouvons nous connecter à notre base de données. Pour cela, dans notre Shell mongo, on va saisir la commande suivante :

```
use maBDD
```

Continuons, ici, par la création d'une collection dans laquelle nous allons, par la suite, stocker nos documents. La commande suivante permet de créer une collection :

```
db.createCollection("maCollection")
```

CREATE

La méthode permettant de créer des documents BSON est :

```
db.maCollection.insert( { "_id" : "1", "nom" : "nom1" } )
```

Cette méthode va simplement insérer le document BSON que vous lui passez en paramètre. Par exemple, si vous souhaitez insérer le document ci-dessous :

```
{
"_id" : "1",
"nom" : "nom1"
}
```

UPDATE

Premier exemple avec la fonction update() qui va être la fonction principale pour mettre à jour nos documents MongoDB. La structure de la fonction update() est la suivante :

```
db.maCollection.update(sélection, modification, options)
```

DELETE

La méthode remove() va être utilisée pour supprimer des documents dans une collection, tout comme la commande Delete du langage SQL. En voici sa structure

```
Db.maCollection.remove(sélection, unique)
```

READ

La méthode db.maCollection.find() prend en compte deux arguments : les critères et la projection. L'exemple suivant va illustrer le principe.

```
db.maCollection.find(critères, projection)
```

Les critères correspondent aux champs contenus dans les documents que nous recherchons (en général ce qui se trouve après la clause WHERE en langage SQL). La projection, elle, va correspondre aux champs que nous souhaitons retourner dans notre résultat (en général ce qui se trouve après la clause SELECT en SQL).

Cette expression permet d'afficher tous les documents contenus dans la collection maCollection.

```
db.maCollection.find( { } )
```

4.4.2 Critères de recherche

Si on connaît l'identifiant, on effectue la recherche ainsi :

```
db.films.find({"_id" : "2"})
```

Sur le même modèle, on peut interroger n'importe quel attribut :

```
db.films.find({"titre" : "Chronique des années de braise"})
```

L'exemple ci-dessus concerne les attributs atomiques (une seule valeur), mais comment faire pour interroger des objets ou des tableaux imbriqués ? On utilise dans ce cas des chemins, un peu à la XPath, mais avec une syntaxe plus "orienté-objet". Voici comment on recherche les films de Mohamed Lakhdar :

```
db.films.find ({"réalisateur.prénom" : "Mohamed Lakhdar"})
```

Conformément aux principes du semi-structuré, on accepte sans protester la référence à des attributs ou des chemins qui n'existent pas. Parce qu'il n'y a pas de schéma, pas de contrainte sur la structure des objets. La requête suivante ne ramène rien, mais ne génère pas d'erreur.

```
db.films.find ({"acteur.prénom" : " Mohamed Lakhdar "})
```

Important : Contrairement à une base relationnelle, une base semi-structurée ne proteste pas quand on fait une faute de frappe sur des noms d'attributs.

4.4.3 Projections

Jusqu'à présent, les requêtes ramènent l'intégralité des objets satisfaisant les critères de recherche. On peut aussi faire des projections, en passant un second argument à la fonction find().

```
db.films.find ({"acteurs.prénom" : "Mohamed Lakhdar"}, {"titre" : true, "acteurs" : 'j'})
```

Le second argument est un objet JSON dont les attributs sont ceux à conserver dans le résultat. La valeur des attributs dans cet objet-projection ne prend que deux interprétations. Toute valeur autre que 0 ou null indique que l'attribut doit être conservé. Si on choisit au contraire d'indiquer les attributs à exclure, on leur donne la valeur 0 ou null. Par exemple, la requête suivante retourne les films sans le genre et sans le pays :

```
db.films.find ({"acteurs.prénom" : " Mohamed Lakhdar "}, {"genre" : null, "pays" : 0})
```

4.4.4 Jointures

La jointure, au sens de : associer des objets distincts, provenant en général de plusieurs collections, pour appliquer des critères de recherche croisés, n'existe pas en MongoDB. C'est une limitation très importante du point de vue de la gestion de données. On peut considérer qu'elle est cohérente avec une approche documentaire dans laquelle les documents sont supposés indépendants les uns des autres, avec une description interne suffisamment riche pour que toute recherche porte sur le contenu du document lui-même. Cela étant, on peut imaginer toutes sortes de situations où une jointure est nécessaire dans une application de traitement de données.

Le serveur ne sachant pas effectuer de jointures, on en est réduit à les faire côté client, comme illustré sur la figure. Cela revient essentiellement à appliquer l'algorithme de jointures par boucle imbriquées en stockant des données temporaires dans des structures de données sur le client, et en effectuant des échanges réseaux entre le client et le serveur.

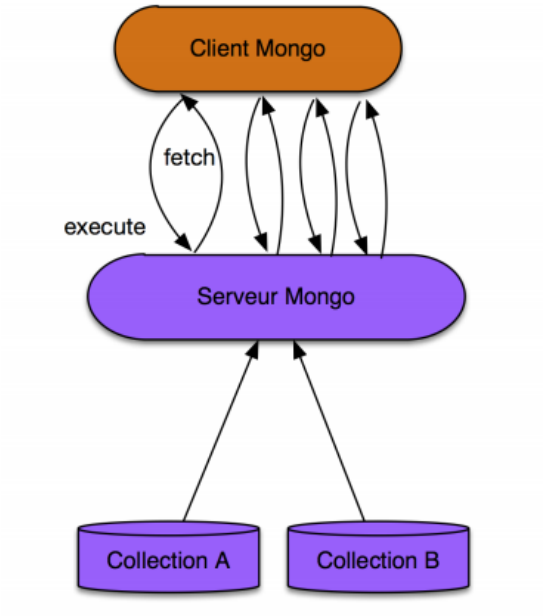

Figure III.5 : La jointure s'effectue coté client [28]

Comme l'interpréteur mongo permet de programmer en Javascript, nous pouvons en fait illustrer la méthode assez simplement. Considérons la requête : "Donnez tous les films dont le réalisateur est Mohamed Lakhdar Hamina ".

Note : Nous travaillons sur la base filmsref dans laquelle un film ne contient que la référence au réalisateur, ce qui évite les redondances, mais complique la reconstitution de l'information.

La première étape dans la jointure côté client consiste à chercher l'artiste Mohamed Lakhdar Hamina et à le stocker dans l'espace mémoire du client (dans une variable, pour dire les choses simplement).

```
hamina = db.artists.find ({"nom" : "Hamina", "prénom" : "Mohamed Lakhdar" })
```

On dispose maintenant d'un objet Hamina. Une seconde requête va récupérer les films dirigés par cet artiste.

```
db.films.find ({"réalisateur._id" : hamina['_id']}, {"titre :1" })
```

4.5 Scalabilité

4.5.1 Définition

Un système est scalable si ses performances sont proportionnelles aux ressources qui lui sont allouées. Les ressources sont les composants matériels que l'on peut ajouter au système. Il s'agit essentiellement des serveurs, mais on peut aussi prendre en compte des dispositifs liés au réseau comme les routeurs.

La notion de performance est la plus flexible. Voici les deux acceptions principales que nous allons étudier.

– débit : c'est le nombre d'unités d'information (documents) que nous pouvons traiter par unité de temps.

– latence : c'est le temps mis pour accéder à une unité d'information (document), en lecture et/ou en écriture.

Au lieu de mesurer le débit en documents/seconde, on regarde souvent le nombre d'octets (par exemple, 10 MO/s).

Exemple : On veut compter le nombre de documents dans le système. Chaque serveur exécute pour cela une opération qui consiste à charger en mémoire centrale les données stockées sur des disques magnétiques. On mesure le débit de cette opération en MO/s.

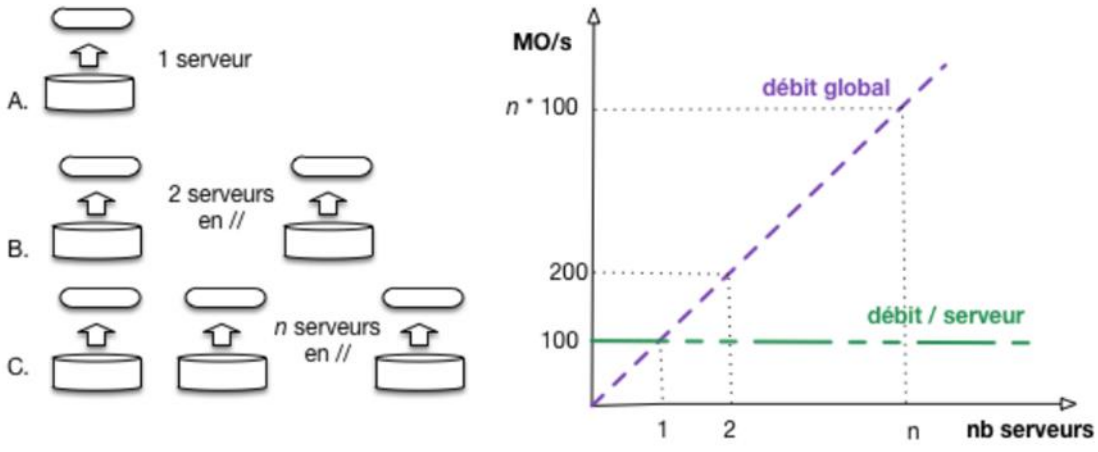

Figure III.6 : Mesure de scalabilité [28]

Avec un seul serveur, on constate un débit de 100 MO/s. Avec deux serveurs, on peut répartir les données équitablement et effectuer les lectures en parallèle : on obtient un débit de 200 MO/s. Il n'est pas difficile d'extrapoler et de considérer que si on a *n* serveurs, le débit sera de *n* * 100 MO/s.

En pratique, il est difficile d'augmenter les performances du réseau proportionnellement aux ressources, et les transferts sont un des facteurs qui peuvent limiter la scalabilité en pratique.

4.6 Réplication et reprise sur panne

4.6.1 Intérêt de la réplication

Bien entendu, on pourrait penser à la solution traditionnelle consistant à effectuer des sauvegardes régulières, et on peut considérer la réplication comme une sorte de sauvegarde continue. Les systèmes NoSQL vont nettement plus loin, et utilisent la réplication pour atteindre plusieurs objectifs.

– **Disponibilité :** La réplication permet d'assurer la disponibilité constante du système. En cas de panne d'un serveur, d'un nœud ou d'un disque, la tâche effectuée par le composant défectueux peut être immédiatement prise en charge par un autre composant. Cette technique de reprise sur panne immédiate et automatique est un atout essentiel pour assurer la stabilité d'un système pouvant comprendre des milliers de nœuds, sans avoir à engloutir un budget monstrueux dans la surveillance et la maintenance.

– Scalabilité (lecture) : Si une donnée est disponible sur plusieurs machines, il devient possible de distribuer les requêtes (en lecture) sur ces machines. C'est le scénario typique pour la scalabilité des applications Web par exemple.

– **Scalabilité (écriture) :** Enfin, on peut penser à distribuer aussi les requêtes en écriture, mais là on se retrouve face à de délicats problèmes potentiels d'écritures concurrentes et de réconciliation.

Le niveau de réplication dépend notamment du budget qu'on est prêt à allouer à la sécurité des données. On peut considérer que 3 copies constituent un bon niveau de sécurité. Une stratégie possible est par exemple de placer un document sur un serveur dans une baie, une copie dans une autre baie pour qu'elle reste accessible en cas de coupure réseau, et une troisième dans un autre centre de données pour assurer la survie d'au moins une copie en cas d'accident grave (incendie, tremblement de terre).

4.6.2 Techniques de réplication

Il y a deux techniques de réplication :

1. Réplication synchrone.
2. Réplication asynchrone.

Les deux techniques utilisent une écriture en mémoire vive RAM, et fichier journal (log) : Au lieu d'effectuer des écritures répétées sur le disque sans ordre prédéfini (accès dits "aléatoires") qui imposent à chaque fois un déplacement de la tête de lecture et donc une latence conséquente, on écrit séquentiellement dans un fichier de journalisation (log) et on place également la donnée en mémoire RAM.

À terme, le contenu de la mémoire RAM, marqué comme contenant des données modifiées, sera écrit sur le disque dans les fichiers de la base de données (opération de flush()). Cela permet de grouper les opérations d'écritures et donc de revenir à des entrées/sorties séquentielles sur le disque, aussi bien dans le fichier journal que dans la base principale. Que se passe-t-il en cas de panne avant l'opération de flush() ? Dans ce cas les données modifiées n'ont pas été écrites dans la base, mais le journal (log) est, lui, préservé. La reprise sur panne consiste à ré-effectuer les opérations enregistrées dans le log.

Le scénario d'une réplication synchrone est alors illustré par la figure III.6 (Réplication avec écritures synchrones). Quand le client reçoit finalement l'acquittement, il peut être sûr que trois copies de d sont effectivement enregistrées de manière durable dans le système. Cela nécessite une chaîne comprenant 8 messages, tout obstacle le long du chemin (un serveur temporairement surchargé par exemple) risquant d'allonger considérablement le temps d'attente.

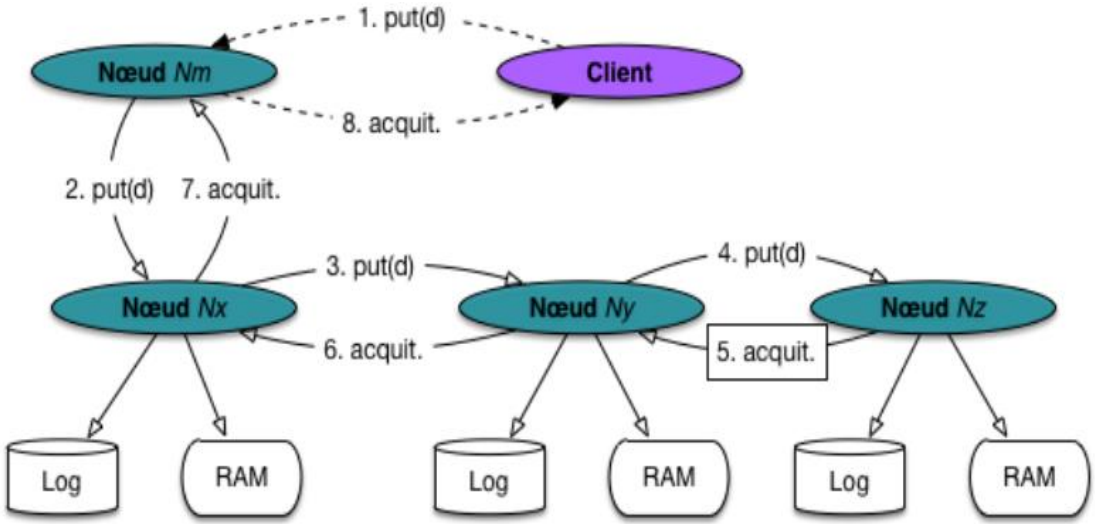

Figure III.7 : Réplication avec écritures synchrones [28]

La seconde technique utilise des écritures asynchrones, dans ce cas le serveur Nx qui reçoit la requête va effectuer l'écriture et envoyer immédiatement l'acquittement au client, lui rendant ainsi la main et permettant la poursuite de son exécution. Après l'acquittement, Nx commence l'envoi des messages pour la réplication, là encore en mode asynchrone (voir figure III.7 Réplication avec écritures asynchrones).

Dans ce scénario, beaucoup plus rapide pour le client, deux phénomènes apparaissent :

– le client reçoit un acquittement alors que la réplication n'est pas complète ; il n'y a donc pas à ce stade de garantie complète de sécurité ;

– le client poursuit son exécution alors que toutes les copies de d ne sont pas encore mises à jour ; il se peut alors qu'une lecture renvoie une des copies obsolètes de d.

Il y a donc un risque pour la cohérence des données.

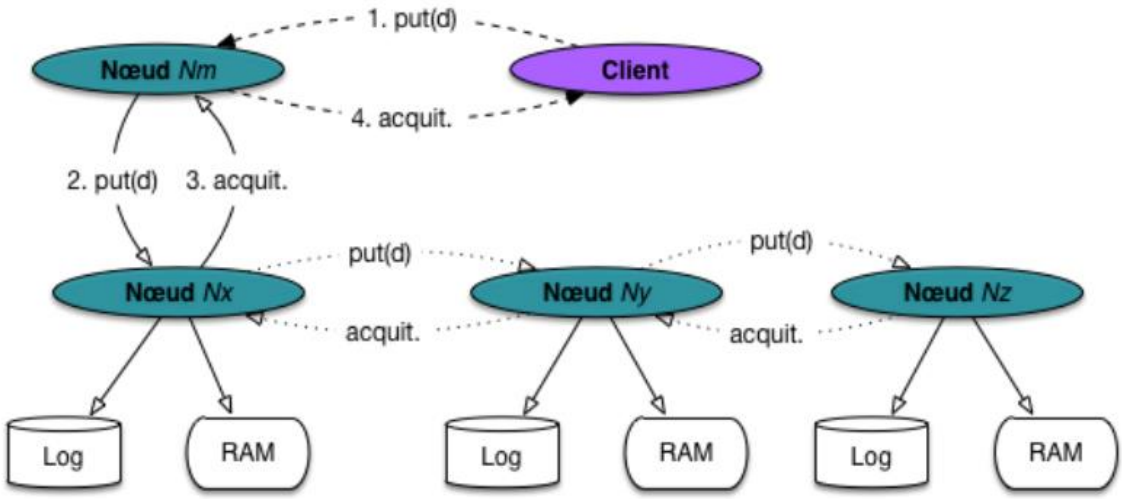

Figure III.8 : Réplication avec écritures asynchrones [28]

4.7 Cohérence des données

La cohérence est la capacité d'un système de gestion de données à refléter fidèlement les opérations d'une application. Un système est cohérent si toute opération (validée) est immédiatement visible et permanente. Si je fais une écriture de d suivie d'une lecture, je

dois constater les modifications effectuées ; si je refais une lecture un peu plus tard, ces modifications doivent toujours être présentes.

La cohérence dans les systèmes répartis (NoSQL) dépend de deux facteurs : la topologie du système (maître-esclave ou multi-nœuds) et le caractère asynchrone ou non des écritures. Trois combinaisons sont possibles en pratique, illustrées par la figure III.8 Réplication et cohérence des données.

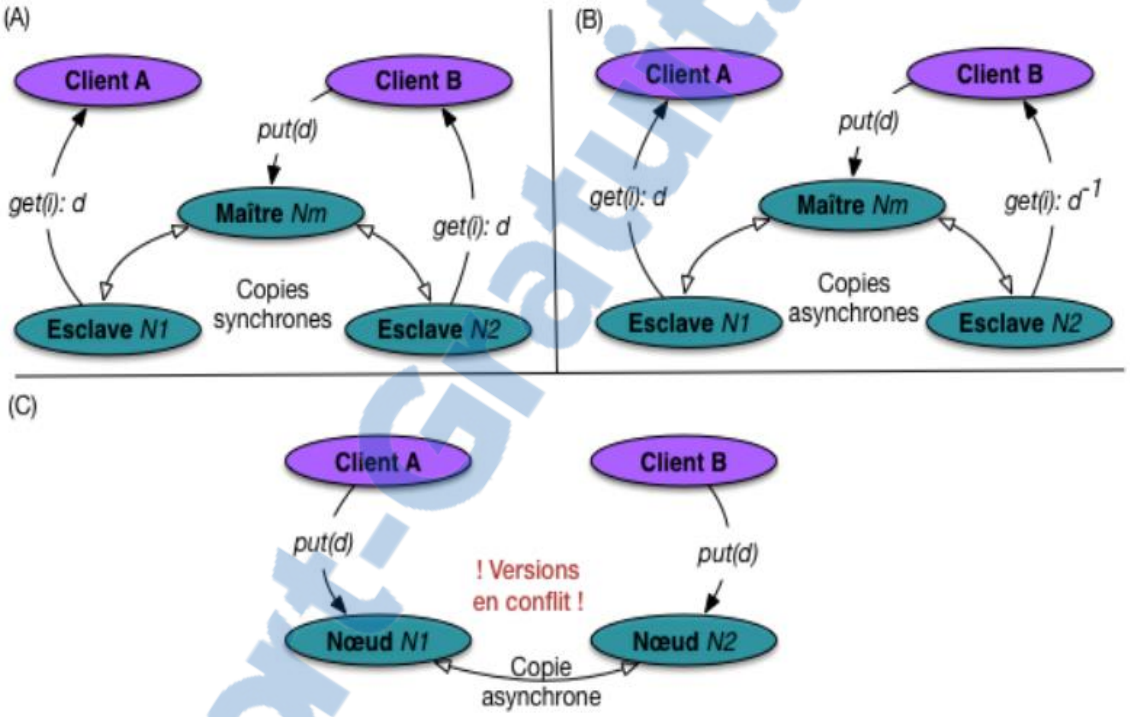

Figure III.9 : Réplication et cohérence des données [28]

Premier cas (A, en haut à gauche) : la topologie est de type maître/esclave, et les écritures synchrones. Toutes les écritures se font par une requête adressée au nœud maître qui se charge de les distribuer aux nœuds esclaves. L'acquittement n'est envoyé au client que quand toutes les copies sont en phase. Ce cas assure la « cohérence forte », car toute lecture du document, quel que soit le nœud sur lequel elle est effectuée, renvoie la même version, celle qui vient d'être mise à jour. Cette cohérence se fait au prix de l'attente que la synchronisation soit complète, et ce à chaque écriture.

Dans le second cas (B) la topologie est toujours de type maître/esclave, mais les écritures sont asynchrones. La cohérence n'est plus forte : il est possible d'écrire en

s'adressant au nœud maître, et de lire sur un nœud esclave. Si la lecture s'effectue avant la synchronisation, il est possible que la version du document retournée soit non pas d mais $d-1$, celle qui précède la mise en jour. L'application client est alors confrontée à la situation, rare mais perturbante, d'une écriture sans effet apparent, au moins immédiat. C'est le mode d'opération le plus courant des systèmes NoSQL, qui autorisent donc un décalage potentiel entre l'écriture et la lecture. Ce décalage est temporaire et toutes les versions vont être synchronisées "à terme" (délai non précisé). On parle donc de cohérence à terme (eventual consistency en anglais).

Le dernier cas (C), correspond à une topologie multi nœuds, en mode asynchrone. Les écritures peuvent se faire sur n'importe quel nœud, ce qui améliore la scalabilité du système. L'inconvénient est que deux écritures concurrentes du même document peuvent s'effectuer en parallèle sur deux nœuds distincts. Au moment où la synchronisation s'effectue, le système va découvrir (au mieux) que les deux versions sont en conflit. Le conflit est reporté à l'application qui doit effectuer une réconciliation (il n'existe pas de mode automatique de réconciliation).

En résumé, trois niveaux de cohérence peuvent se exister dans les systèmes NoSQL :

- **cohérence forte** : toutes les copies sont toujours en phase, le prix à payer étant un délai pour chaque écriture.
- **cohérence faible** : les copies ne sont pas forcément en phase, et rien ne garantit qu'elles le seront, cette situation, trop problématique.
- **cohérence à terme** : c'est le niveau de cohérence typique des systèmes NoSQL : les copies ne sont pas immédiatement en phase, mais le système garantit qu'elles le seront "à terme".

4.8 Réplication dans MongoDB

MongoDB présente un cas très représentatif de gestion de la réplication et des reprises sur panne.

4.8.1 Replica set

Une grappe de serveurs partageant des copies d'un même ensemble de documents est appelée un replicat set (RS) dans MongoDB. Dans un RS, un des nœuds joue le rôle de maître (on l'appelle primary) ; les autres (esclaves) sont appelés secondaries.

Un RS contient typiquement trois nœuds, un maître et deux esclaves. C'est un niveau de réplication suffisant pour assurer une sécurité presque totale des données. Dans une grappe MongoDB, on peut trouver plusieurs replica sets, chacun contenant un sous-ensemble d'une très grande collection.

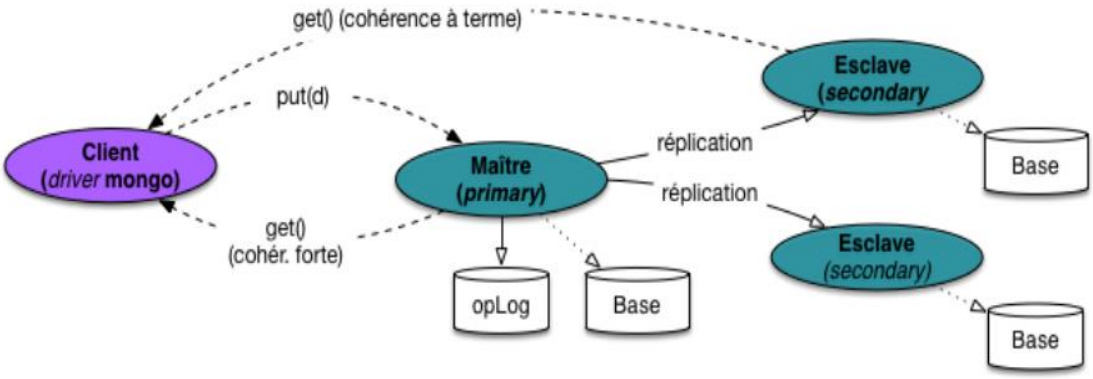

Figure III.10 : Un réplica set dans MongoDB [28]

Le maître est ici chargé du stockage de la donnée principale. L'écriture dans la base est paresseuse, et un journal des transactions est maintenu par le maître (c'est une collection spéciale nommée opLog). La réplication vers les deux esclaves se fait en mode asynchrone.

Deux niveaux de cohérence sont proposés par MongoDB : La cohérence forte est obtenue en imposant au client d'effectuer toujours les lectures via le maître. Dans un tel mode, les esclaves ne servent pas à répartir la charge, mais jouent le rôle restreint d'une sauvegarde/réplication continue, avec remplacement automatique du maître si celui-ci subit une panne.

Important : La cohérence forte est le mode par défaut dans MongoDB. La cohérence à terme est obtenue en autorisant les clients (autrement dit, très concrètement, le driver MongoDB intégré à une application) à effectuer des lectures sur les esclaves.

4.9 Partitionnement

Le fait de disposer des mêmes données sur plusieurs serveurs par réplication ouvre également la voie à la distribution de la charge (en recherche, en insertion) et donc à la scalabilité.

4.9.1 Principes généraux

On considère une collection constituée de documents dotés d'un identifiant. Dans ce chapitre, on va essentiellement voir une collection comme un ensemble de paires *(i, d)*, où *i* est un identifiant et *d* le document associé.

Le principe du partitionnement s'énonce assez simplement : la collection est divisée en fragments formant une partition de l'ensemble des documents. Dans notre cas *S* est une collection, les éléments sont des documents, et les fragments sont des sous-ensembles de documents.

Note : On trouvera souvent la dénomination shard pour désigner un fragment, et sharding pour désigner le partitionnement.

4.9.2 Clé de partitionnement

Un partitionnement s'effectue toujours en fonction d'une clé, soit un ou plusieurs attributs dont la valeur sert de critère à l'affectation d'un document à un fragment. Un bon partitionnement répartit les documents en fragments de taille comparable. Idéalement, on choisira comme clé de partitionnement l'identifiant unique des documents.

4.9.3 Structures

Il existe deux grandes approches pour déterminer une partition en fonction d'une clé : par intervalle et par hachage.

– Premier cas (par intervalle), on obtient un ensemble d'intervalles disjoints couvrant le domaine de valeurs de la clé, à chaque intervalle correspond un fragment.

–Second cas (par hachage), une fonction appliquée à la clé détermine le fragment d'affectation.

Elles déterminent la construction des structures de données représentant le partitionnement. Ces structures sont toujours au nombre de deux.

– la structure de routage établit la correspondance entre la valeur d'une clé et le fragment qui lui est associé (ou, très précisément, l'espace de stockage de ce fragment). – la structure de stockage est un ensemble d'espaces de stockages séparés, contenant chacun un fragment. Sans la structure de routage, rien ne fonctionne. Elle se doit de fournir un support très efficace à l'identification du fragment correspondant à une clé, et on cherche en général à faire en sorte qu'elle soit suffisamment compacte pour tenir en mémoire RAM. Les fragments sont, eux, nécessairement stockés séquentiellement sur disque.

Un partitionnement doit être dynamique : en fonction de l'évolution de la taille de la collection et du nombre de ressources allouées à la structure pour que le nombre de fragments doit pouvoir évoluer. C'est indispensable pour optimiser l'utilisation de l'espace disponible et obtenir les meilleures performances malgré que ce soit la propriété la plus difficile à satisfaire techniquement.

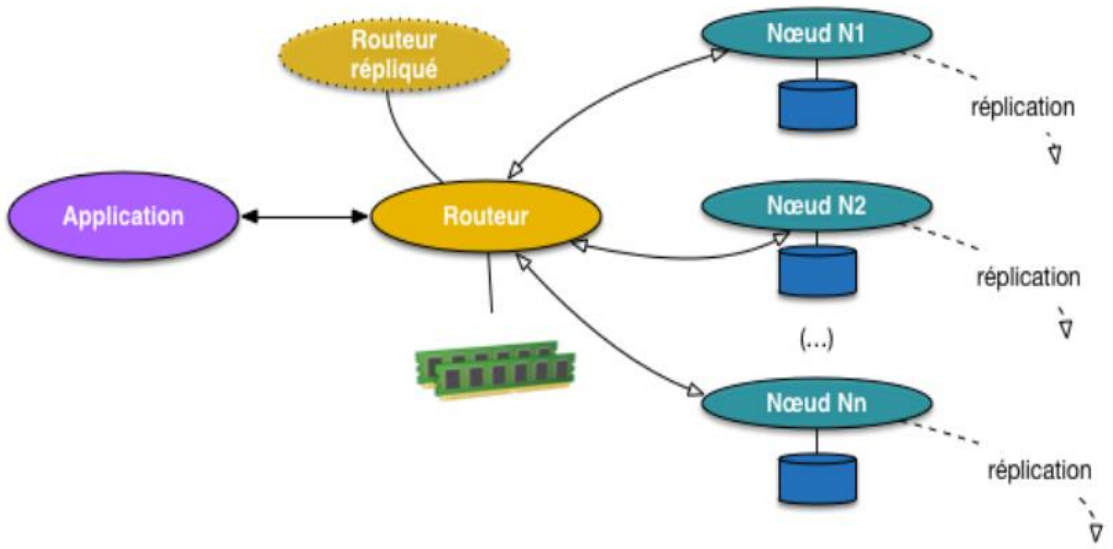

Figure III.11 : Partitionnement et systèmes distribués [28]

Un nœud particulier, le routeur, maintient la structure de routage, reçoit les requêtes de l'application et les redirige vers les nœuds en charge du stockage. Ces derniers stockent les fragments. Cette organisation s'additionne à celle gérant la réplication. Le routeur par exemple doit être synchronisé avec au moins un nœud-copie apte à le suppléer en cas de défaillance, de même, chaque nœud de stockage gère la réplication des fragments dont il a la charge et en informe le routeur pour que ce dernier puisse rediriger les requêtes en cas de besoin.

Il est nécessaire de souligner que le vocabulaire varie constamment :

– le routeur est dénommé, selon les cas, Master, Balancer, Router/Config server, ...

– les fragments sont désignés par des termes comme chunk, shard, tablet, region, bucket,...

Les méthodes de partitionnement, par intervalle ou par hachage, sont représentées par des systèmes de gestion de données importants :

– par intervalle : HBase/BigTable, MongoDB, ...

– par hachage : Dynamo/S3/Voldemort, Riak, REDIS, memCached, ElasticSearch, ...

PARTIE 2 : Implémentation et mise en œuvre de la migration

1. TABLES ORACLE DE LA BIOMETRIE

La base de données de la biométrie contient beaucoup de données avec 76 tables. Cependant, les données qui représentent le cœur du métier de la biométrie sont seulement 42 tables centrales (qui commencent par *ENT**) avec un volume de 4 GO.

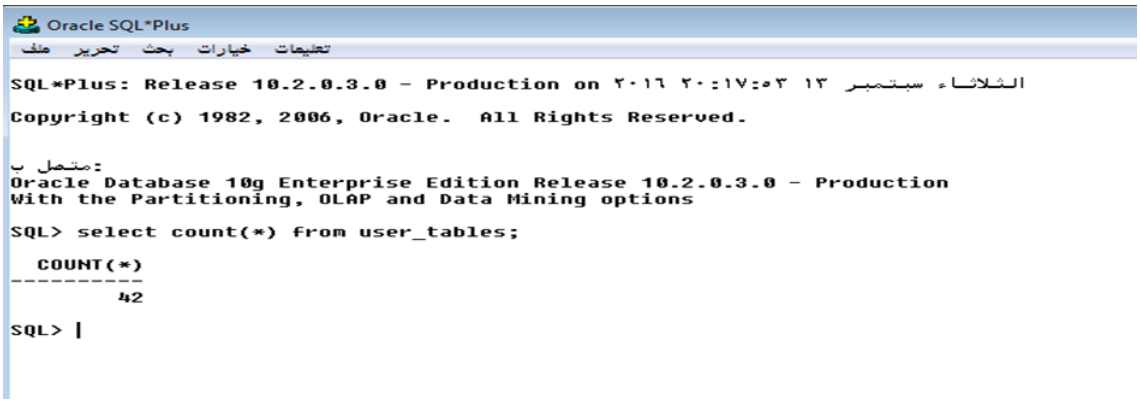

Figure III.12 : Nombre de tables de la biométrie

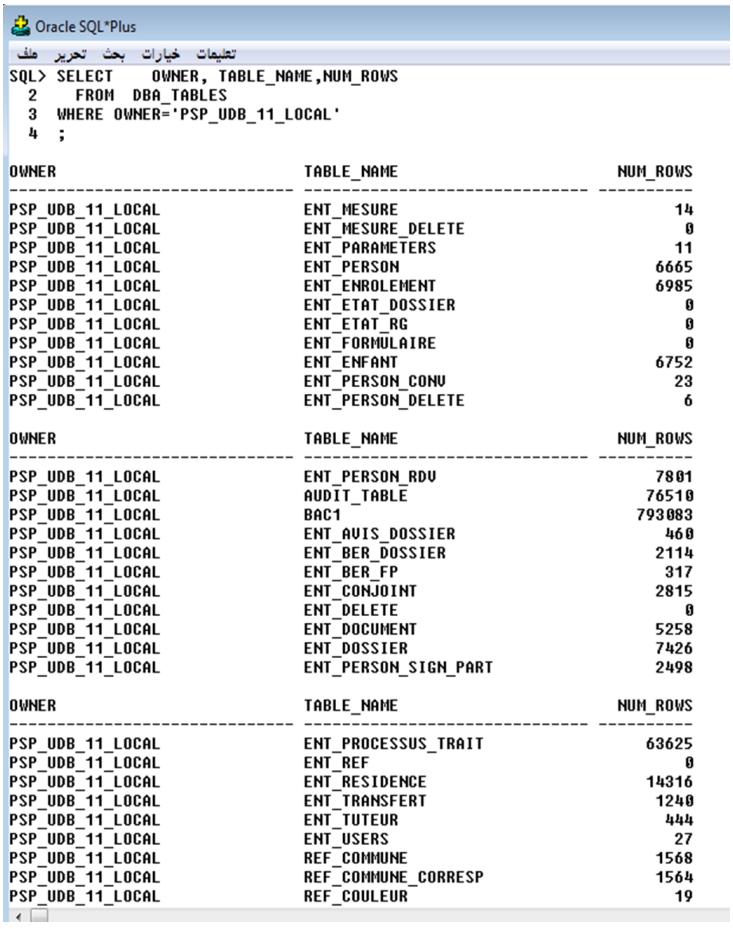

Figure III.13 : Les tables de la biométrie

2. ENVIRONNEMENT DE TRAVAIL

Dans cette section, nous allons recenser les outils Soft et Hard adoptés pour réaliser l'application de migration de base de données des passeports biométriques sous Oracle vers une nouvelle base de données biométriques sous MongoDB.

2.1 Environnement hard

- **premiere machine:**

Hôte : Dell inspiron N7110

Microprocesseur : intel ® Core TM i5-2430M

RAM : 6 GO

- **deuxieme machine:**

Hôte : HP ProBook 4540s

Microprocesseur : intel ® Core TM i7-3632 QM

RAM : 8 GO

2.2 Langage de programmation

Pascal comme langage de programmation, (dont le nom vient du mathématicien français **Blaise Pascal**) a été inventé par **Niklaus Wirth** dans les années 1970. Il permet de créer des logiciels compatibles avec des nombreux systèmes d'exploitation.

2.3 Outil de développement

DELPHI version 7.0, est un environnement de développement par borland Software corporation.

2.4 Système de gestion de base de données

Nous avons utilisé deux types de système de gestion de bases de données. Pour le relationnel on a opté pour le ORACLE et pour le NoSQL c'est MongoDB.

2.4.1 ORACLE

Nous avons utilisé oracle entreprise manager, version 10g qui fait partie des logiciels de gestion de base de données relationnelles les plus utilisés au monde et le meilleur classé.

2.4.2 MongoDB

Pour la solution NoSQL, nous avons opté pour la version 3.2.8 de MongoDB comme un modèle orienté document.

2.5 Format de données communiquées

Le format de données communiquées c'est JSON, qui est un format de données textuel, générique, Il permet de représenter des informations structurées. Le principal avantage de l'utilisation de JSON, est sa simplicité de mise en œuvre.

La démarche que nous allons suivre pour aboutir notre projet est la suivante :

1. installation d'Oracle et l'importation de la base biométrique située au serveur de la daïra.
2. Génération des fichiers JSON contenants les enregistrements des tables.
3. Installation de MongoDB et l'importation des fichiers JSON.

3. INSTALLATION ET CONFIGURATION DE MONGODB

Dans cette partie, nous entamons l'installation de MongoDB. Nous avons téléchargé la dernière version de MongoDB. Voici les étapes d'installation que nous avons suivie :

3.1. Préparation et installation

nous décompressons l'exécutable de MongoDB par la commande suivante :
tar -xvf mongodb-linux-x86_64-ubuntu1604-3.2.8.tgz (voir Figure III.14) et par la suite nous créons maintenant le répertoire */data/db*

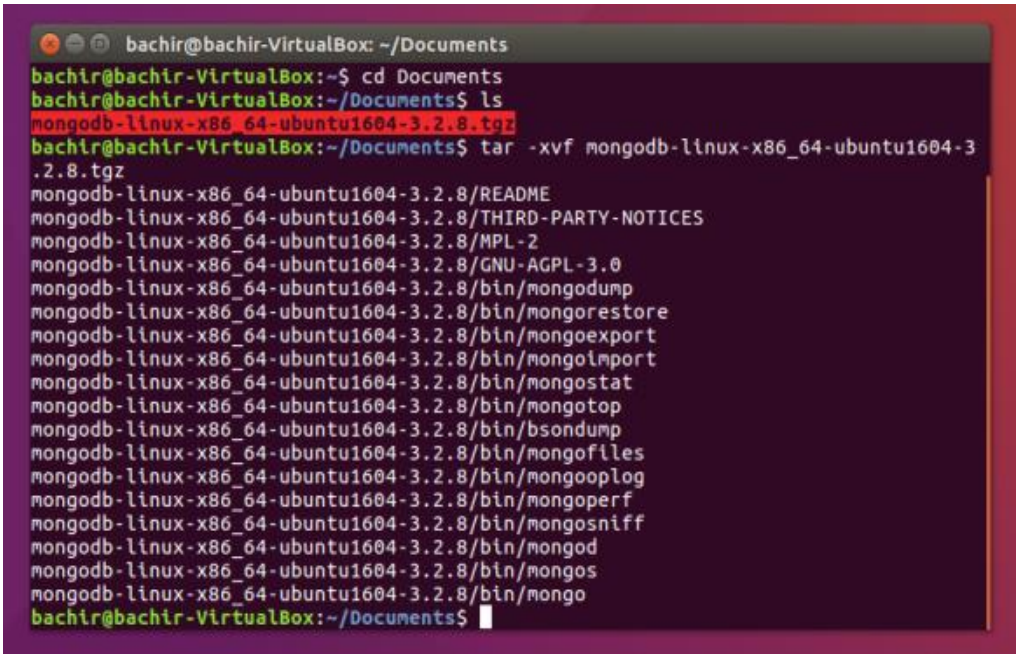

Figure III.14 : Décompression du l'exécutable de MongoDB

Dans le répertoire décompressé */home/ bin* nous remarquons qu'il existe trois (03) services importants : *mongod*, *mongo* et *mongos* (voir Figure III.15).

Le 1er service *mongod* est responsable de créer et gérer la base de données MongoDB.

Le 2ème service *mongo* est l'interpréteur de la ligne de commande MongoDB, utilisé pour manipuler la base de données MongoDB et ses objets.

Le 3ème service *mongos* est un dispatcheur (*Load Bancer*) pour la répartition dans un milieu distribué, nous le verrons dans la préparation de l'environnement distribué.

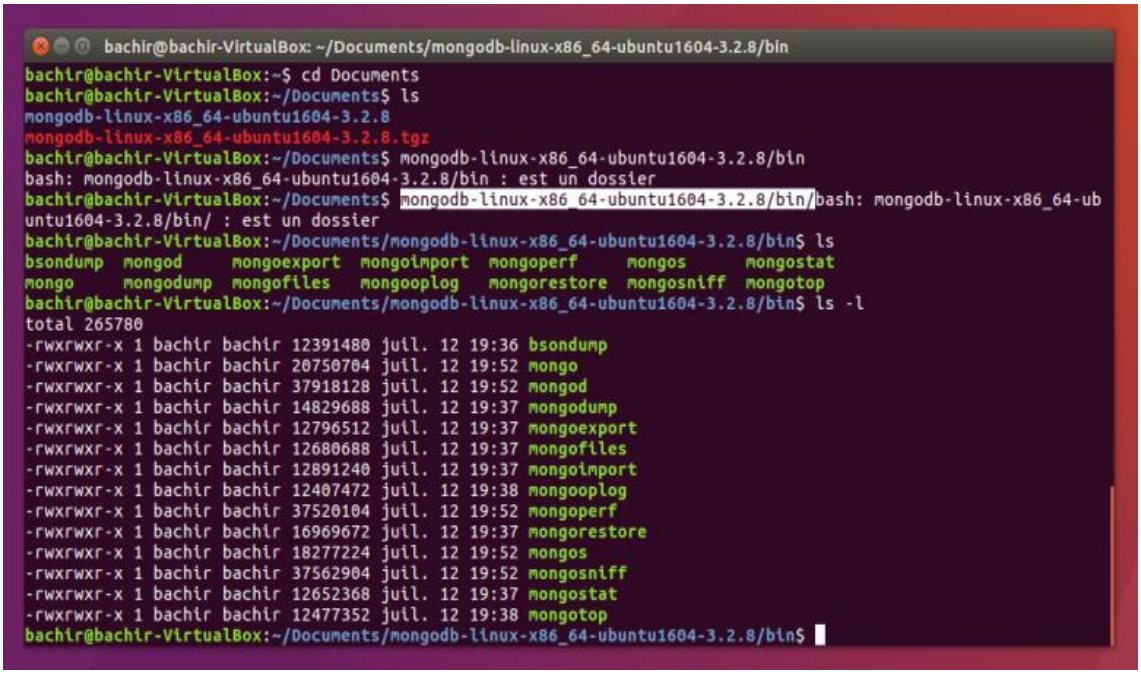

Figure III.15 : Services exécutables de MongoDB

3.2. Lancement des services MongoDB sous Ubuntu 16.04

Pour lancer MongoDB, nous ouvrons deux fenêtres du terminal Shell Linux .Dans la 1ère fenêtre, nous lançons le 1er service *mongo* en précisant le répertoire */data/db* comme emplacement de la base de données (voir Figure III.16). Nous remarquons que le service est lancé avec le N° de port par défaut 27017. Cette fenêtre doit être toujours en exécution pour assurer la gestion et les connexions à la base.

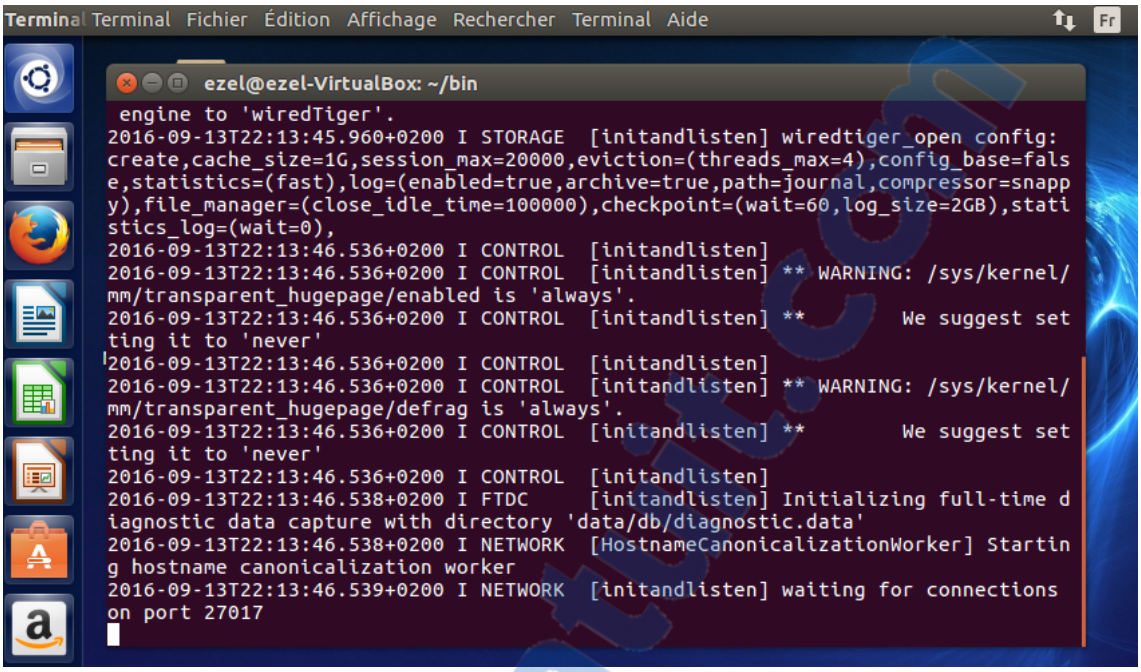

Figure III.16 : Lancement du Services mongod

Dans la 2ème fenêtre, nous lançons le 2ème service (l'interpréteur) *mongo* pour se connecter et manipuler les bases de données (Figure III.17).

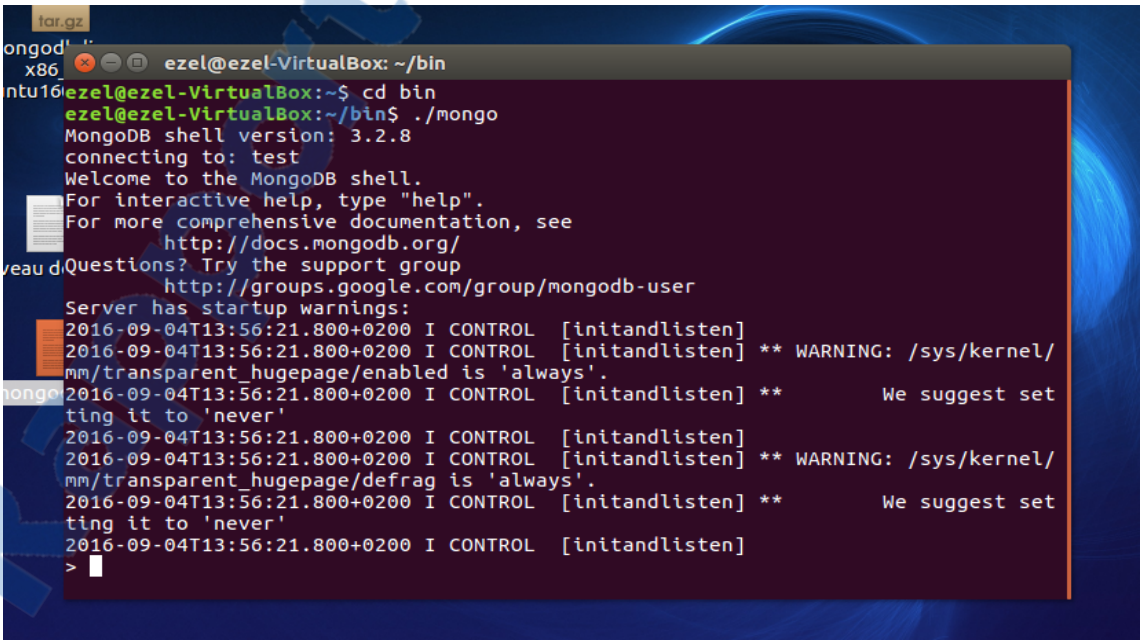

Figure III.17 : L'interpréteur mongo

Une fois connecté, nous remarquons qu'une base nommée *local* est déjà créée automatiquement. Nous pouvons taper des commandes MongoDB telles que : *db* pour afficher le nom de la base actuelle (par défaut c'est *test*), et *show dbs* pour visualiser les bases existantes. Nous créons maintenant notre base de données appelée biom par la commande use puis affichons la liste des bases. La base biom va abriter les données de la biométrie une fois migrées vers MongoDB comme c'est montré ci-dessous :

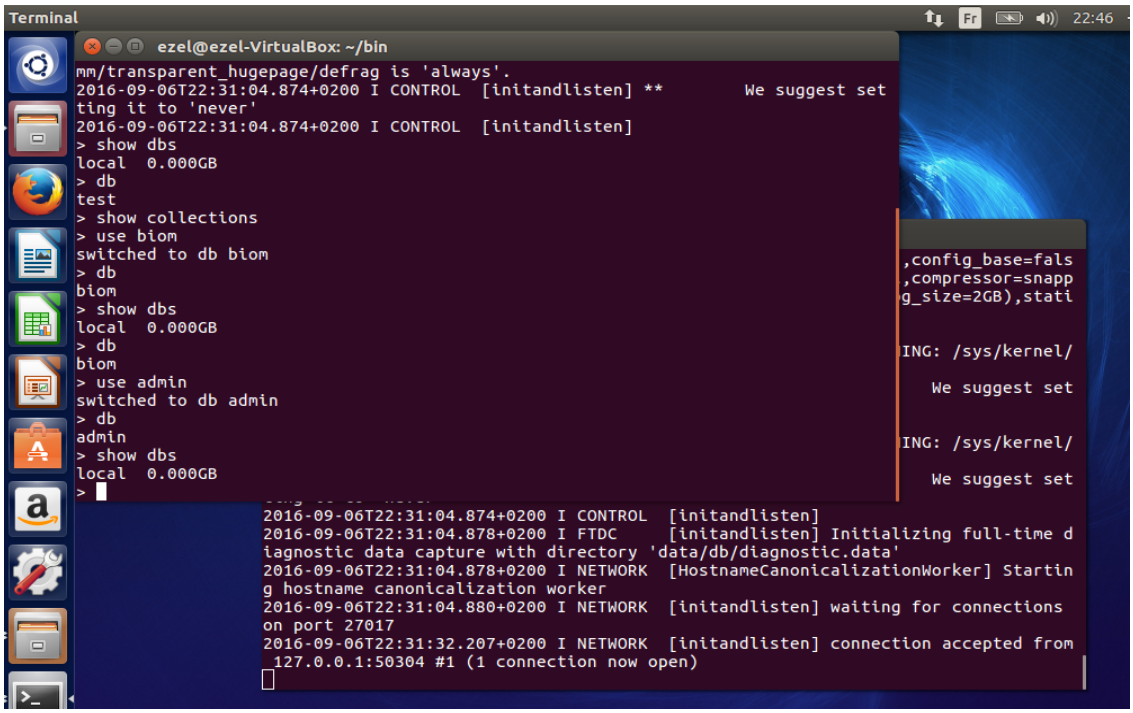

Figure III.18 : Vérification des bases par défaut

3.3. Lancement des services MongoDB sous Windows 7

Il faut commencer par télécharger MongoDB disponibles à l'adresse suivante : http://www.mongodb.org

Une fois le téléchargement est terminé il est recommandé de dézipper l'archive et de placer le contenu dans un répertoire nommé **Mongodb** à la racine du disque dur : **«c:\bin»**. Pour stocker les informations, MongoDB utilise le répertoire par default c:\bin\data\db. Il faut donc créer ce répertoire à partir de l'explorateur Windows.

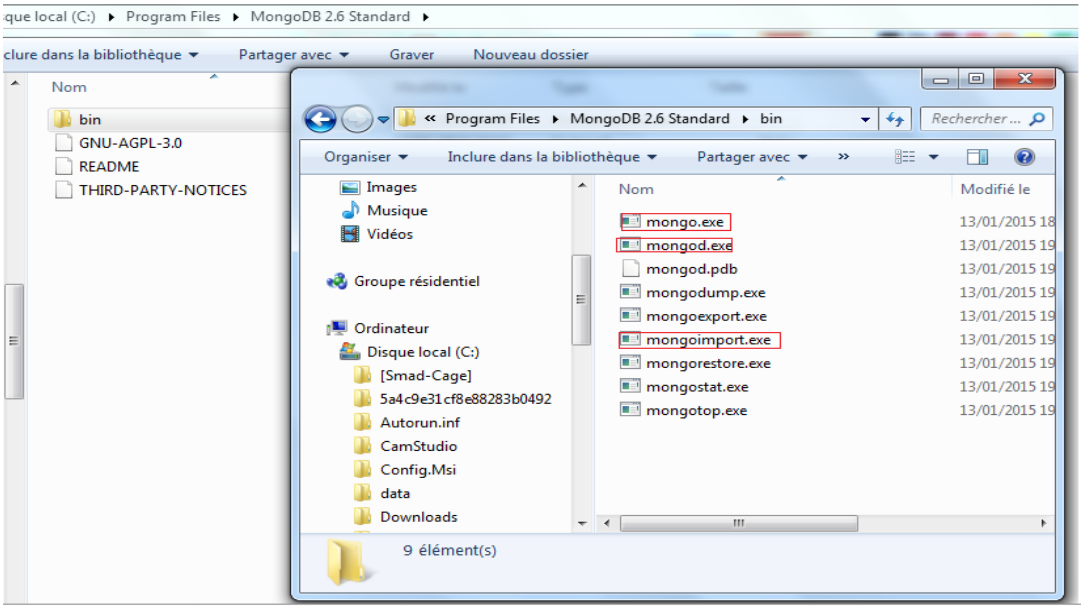

Figure III.19 : Liste des services MongoDB

Pour faire les vérifications, on lance **mongod.exe et mongo.exe à** partir du répertoire **c:\bin**.

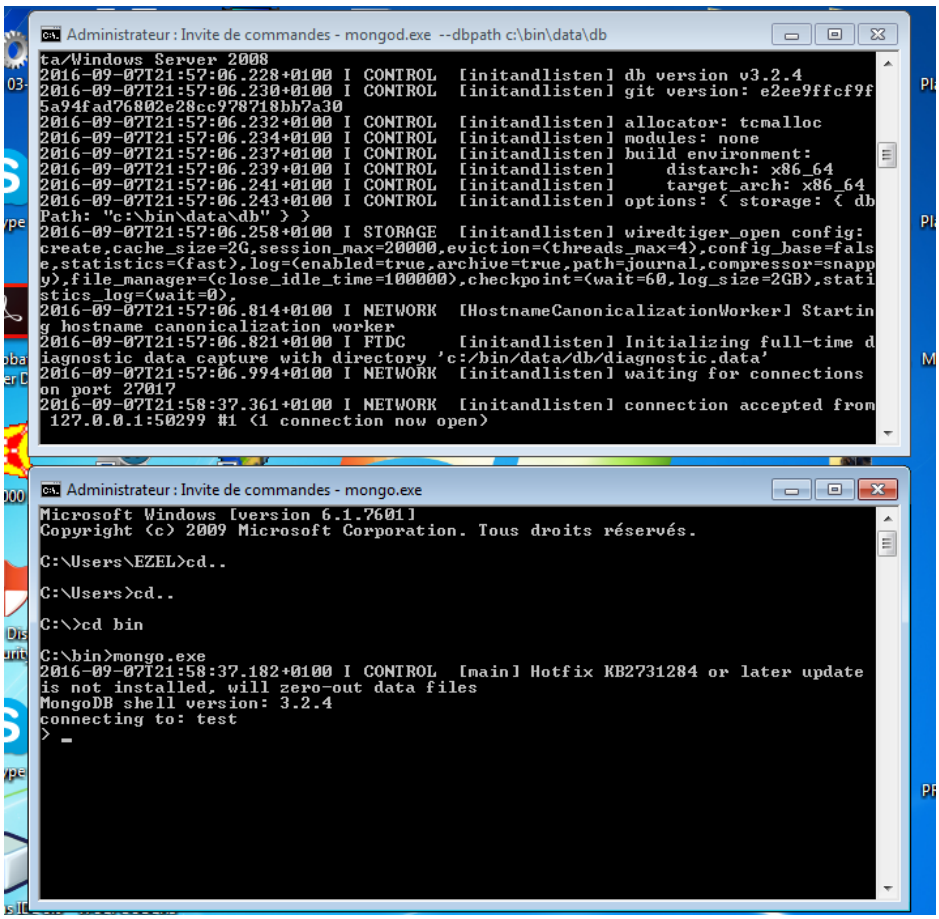

Figure III.20 : Lancement des services MongoDB

- On va exécuter la commande suivante : `mongod --install --rest –master –logpath=C:\bin\log\mongolog.txt`
- Lancer l'éditeur de registres

- Chercher HKEY_LOCAL_MACHINE >> SYSTEM >> CurrentControlSet >> services>>MongoDB
- Cliquer sur ImagePath, changer sa valeur avec cette valeur :

C:\bin\mongod.exe --rest --master -
logpath=C:\bin\log\mongolog.txt --
dbpath=C:\bin\data –service

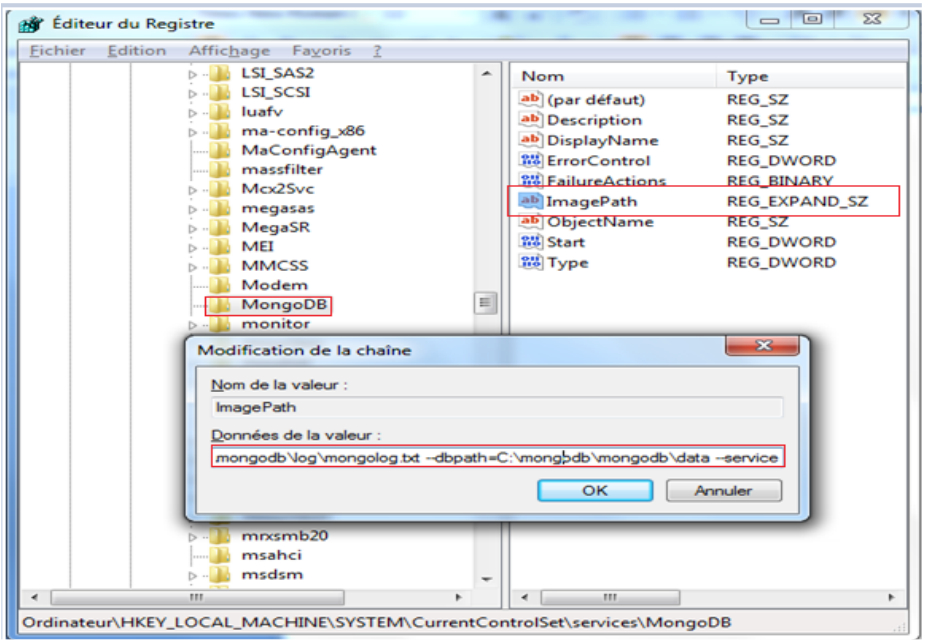

Figure III.21 : Configuration des services MongoDB

-Lancer maintenant le service MongoDB

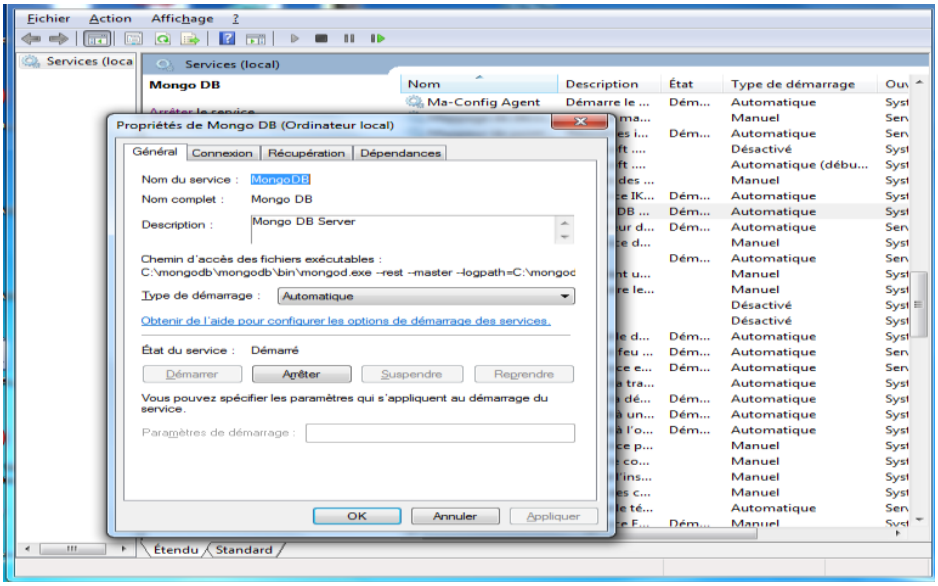

Figure III.22 : démarrage des services de MongoDB

4. INSTALLATION ET CONFIGURATION DE L'ENVIRONNEMENT DISTRIBUE

Dans cette partie, nous allons installer un environnement distribué en se basant sur la notion du Sharding MongoDB, pour notre environnement nous implémenter le Sharding MongoDB conformément au schéma de la figure suivante

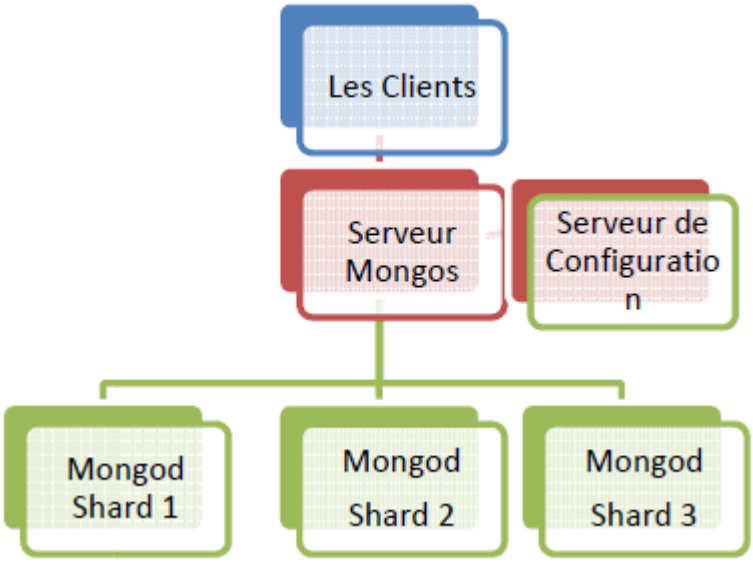

Figure III.23 : Principe sharding

Donc, nous avons besoin de 5 serveurs (nœuds) :

- 1 serveur *Mongos* qui gère la répartition (Sharding) et distribue les données dans les 3 serveurs Shards (*Mongod*).
- 1 serveur de configuration utilisé par Mongos pour gérer la configuration des différents Shards et réplications.
- 3 serveurs Shards, chacun exécute le service *Mongod* et contient des partitions des données distribuées par *Mongos*.

Tous les clients se connectent au serveur Mongos responsable du Sharding. A noter que dans un environnement de production réel, il faut avoir 3 serveurs de configuration redondants (sauvegardes), et peut être d'autres serveurs *Mongos* pour l'équilibrage de charge vis à vis les connexions des clients. Aussi pour un Shard, il faut avoir au moins 3 réplications.

Dans notre cas, nous allons installer trois (03) Shards et un 4ème serveur qui joue deux rôles *Mongos* et serveur de configuration. Ces quatre (04) serveurs doivent être

connectés entre eux. Une autre remarque importante et intéressante dans le Sharding, est que les nœuds peuvent être de systèmes hétérogènes, où chaque nœud peut avoir son propre système installé. Nous utilisons cette propriété intéressante dans notre cas d'étude (02 shards sous linux et 01 shard et 02 serveurs sous Windows 7).

4.1. Configuration des Shards MongoDB

Commençons par installer et configurer les 3 serveurs Shards.

Le 1er Shard sera notre machine qu'on vient de configurer. Dans ce Shard1, nous arrêtons par la touche CTRL-C le service mongod que nous avons lancé avant .Puis, nous le relançons en spécifiant l'argument shardsvr.

La commande. /*mongod --shardsvr --dbpath /data/db* indique que ce *mongod* est un Shard et fait partie d'un environnement distribué. Nous remarquons dans la Figure III.24 que l'option Sharding dans un Cluster est activé, et que le nouveau port est maintenant 27018.

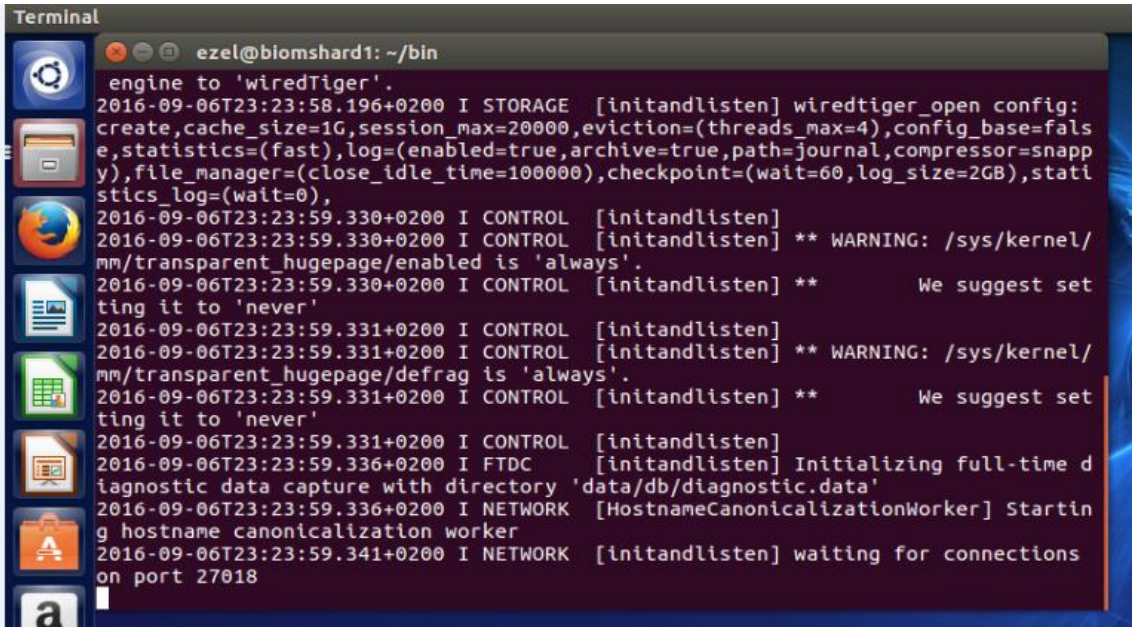

Figure III.24 : Lancement du premier Shard sous ubuntu 16.04

De la même façon, nous configurons le Sharding pour le 2ème Shard. Le système d'exploitation de cette 2ème machine est ubuntu 16.04.

Nous renommons cette machine Shard2, et nous créons le compte *mongodb*, le répertoire */data/db*, et nous installons MongoDB de la même façon qu'avec *Shard1*.

Enfin, le 3ème Shard est configuré au même titre que les 2 Shards précédents mais sous Windows 7.

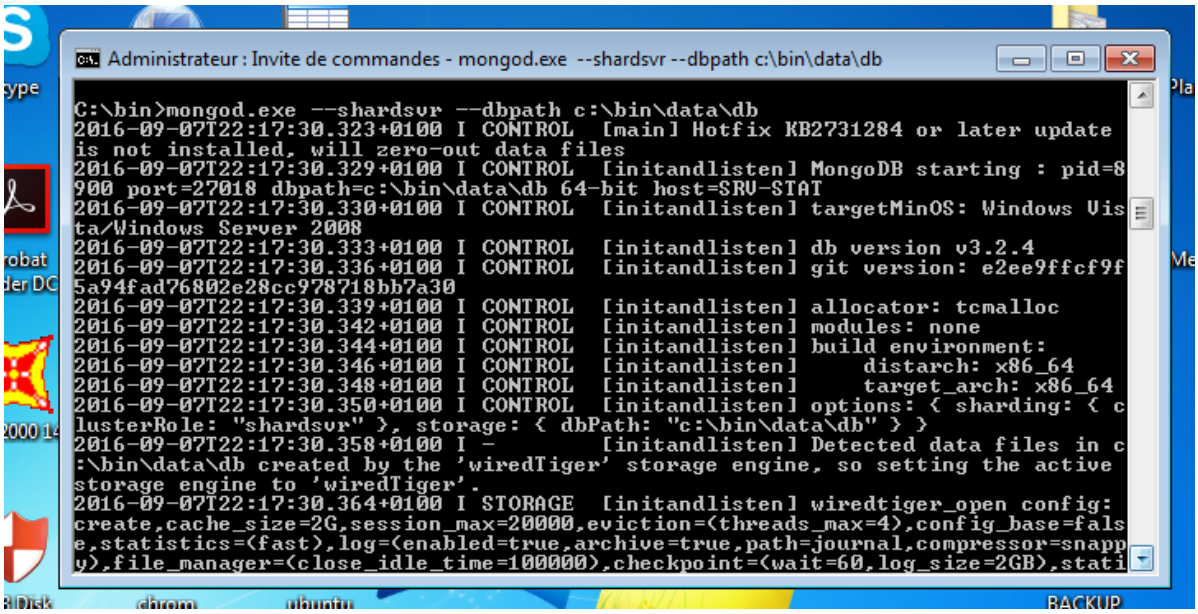

Figure III.25 : Lancement du premier Shard sous Windows 7

4.2 Configuration du Serveur de Config MongoDB

Trois services sont utilisés à savoir : mongod, configsvr, et mongos.

Nous passons maintenant au 4ème serveur pour le configurer comme étant un serveur de configuration pour le Sharding. Afin d'étaler l'hétérogénéité de notre environnement distribué, nous avons choisi Windows 7 64 bits comme système pour ce serveur. Dans cette machine nous créons le répertoire c:*bin**mongodb_config*, qui va contenir les fichiers de configuration, et nous installons MongoDB sous Windows 64 bits à télécharger de site officiel de MongoDB.

Ensuite, nous exécutons la commande *mongod* en spécifiant l'argument *configsvr* pour dire qu'il s'agit d'un serveur de configuration du Sharding suite à la commande suivante :

mongod.exe --configsvr --dbpath c:\ bin\dataconfig

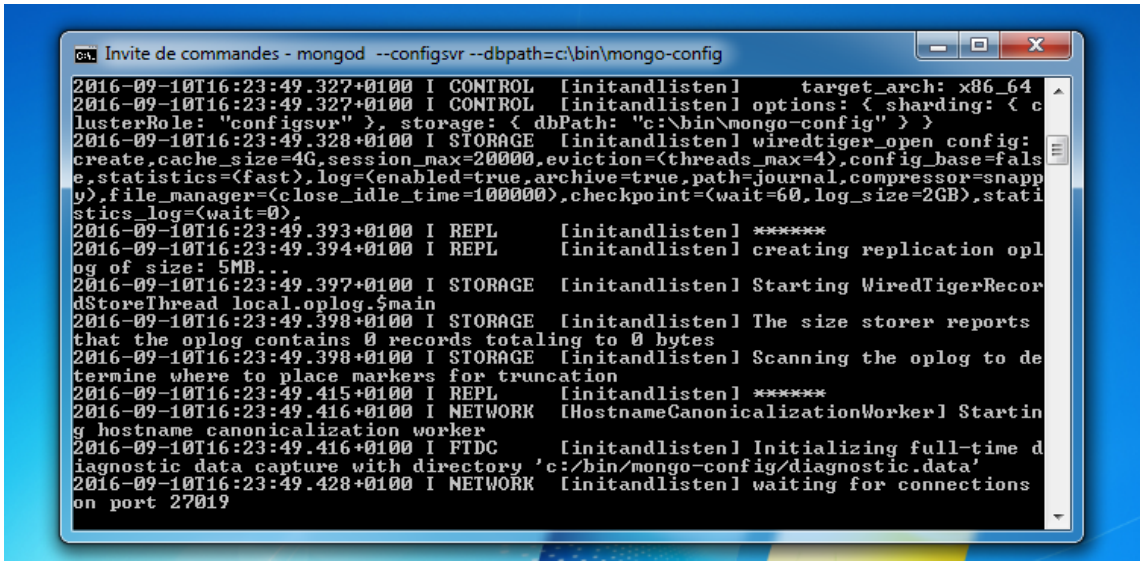

Figure III.26 : Lancement du serveur de configuration du Sharding

4.3. Configuration du Serveur Mongos

Le service *Mongos* est très important car il prend en charge toutes les opérations de Sharding, il répartie et collecte les données sur les Shards en se servant des informations stockées dans le serveur de configuration. De plus Mongos représente l'interface avec les clients. Dans notre environnement, et vu qu'il manque de machines, nous allons configurer *Mongos* dans le serveur de configuration. Ceci est possible puisque ce serveur n'est pas trop surchargé.

Nous allons suivre les étapes ci-dessous pour configurer le service *Mongos*.

1ère étape : Lancement de Mongos et lien avec le serveur Config

On exécute la commande *mongos* en spécifiant l'argument *configdb* et l'*@ip* et le *port* du serveur de configuration. Ceci permet de lier le serveur *Mongos* au serveur de configuration.

Pour notre cas, le serveur de Config est le même, l'*@ip* et le port sont 192.168.0.1:27019 comme il est montré dans la Figure suivante :

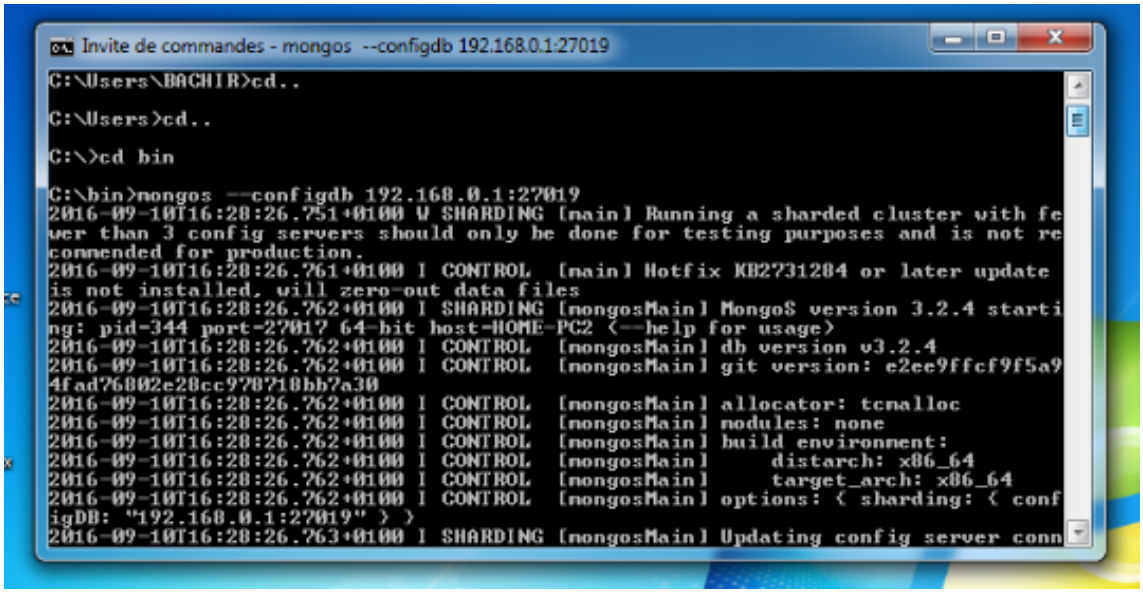

Figure III.27 : Lancement du serveur Mongos de configuration du Sharding

Nous remarquons qu'un service *Balancer* est lancé pour une éventuelle répartition par la suite.

2ème étape: Ajout des trois Shards à la configuration

Nous lançons l'interpréteur *mongo* et se connectons à la base de données *admin* qui est déjà créée par défaut lors de l'installation. Une fois connecté, nous remarquons que l'invite affiche maintenant *"mongos>"* (Figure III.28). Nous lançons la commande ***db.runCommand({addshard:"@ip:port"})*** pour chacun des 3 *shards* en remplaçant l'@ip et le port par celui des *Shards*. Ces commandes permettent de lier les 3 *Shards* à la configuration du Sharding.L'ensemble des ces commandes est affiché dans la Figure *3.13*. Les adresses *ip* utilisées sont celles des *Shard1*, *Shard2* et *Shard3* respectivement.

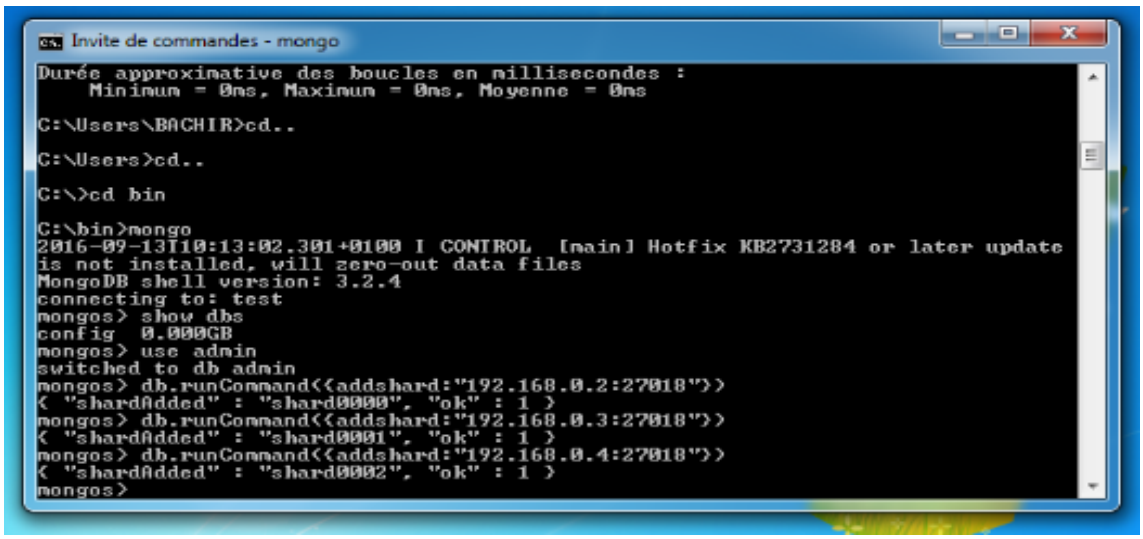

Figure III.28 : Ajout des trois Shards à la configuration du Sharding

Nous remarquons qu'après l'ajout des *Shards* au *Mongos*, la base de données *biom* (qui existait juste dans le *Shard1*) est maintenant accessible dans *Mongos*. Ceci confirme la gestion centralisée à partir de *Mongos*.

Pour vérification, nous pouvons basculer vers la base de données *config* et afficher des informations sur les *Shards* en utilisant des commandes de la Figure suivante.

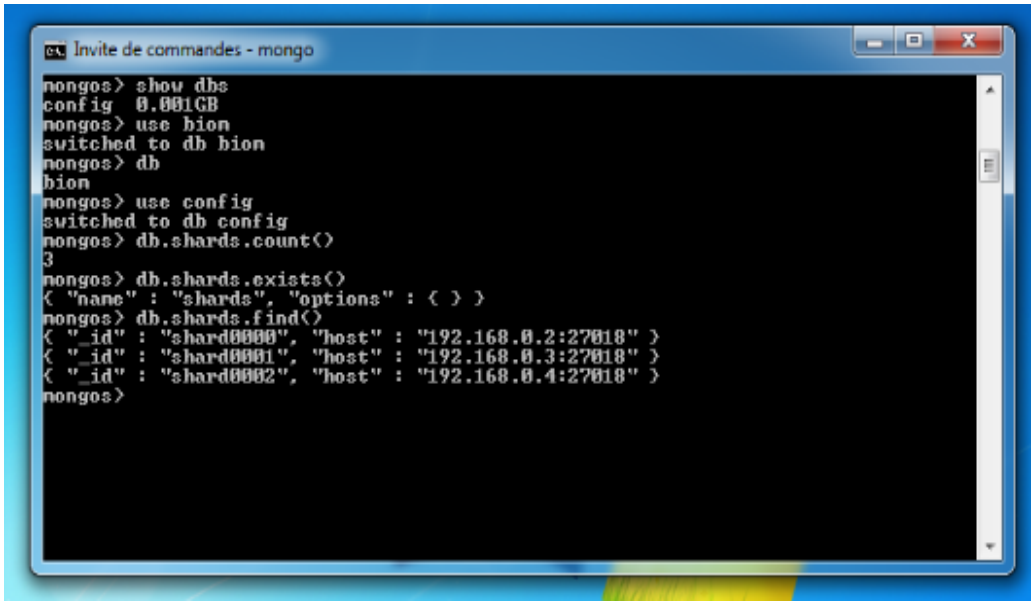

Figure III.29 : Activation réussi du Sharding pour la base biom

3ème étape : Activer le Sharding pour la base de données

L'option du Sharding n'est pas activée par défaut pour toutes les bases, il faut donc l'activer pour notre base biom suivant les images suivantes :

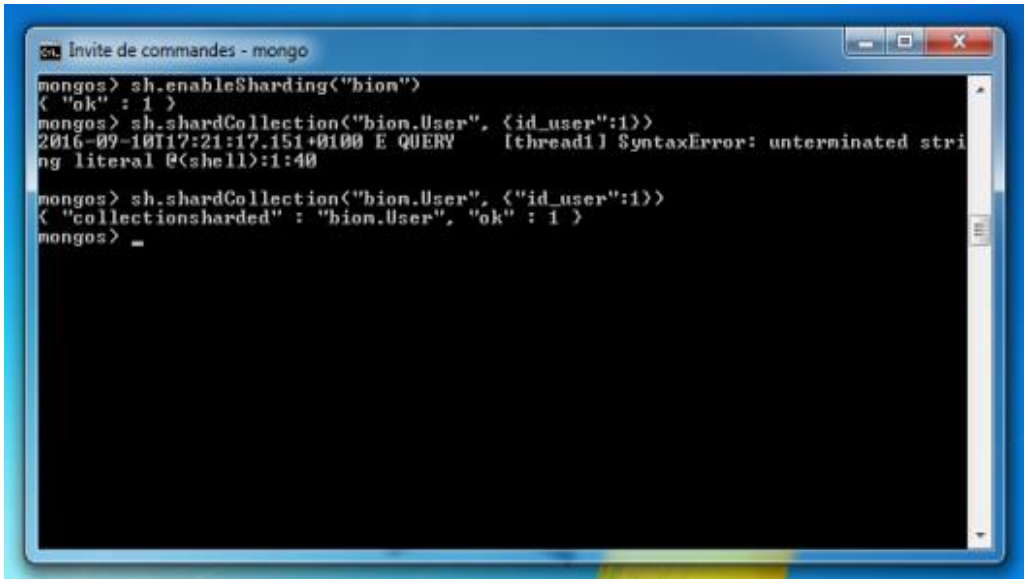

Figure III.30 : Activation du Sharding pour la base biom

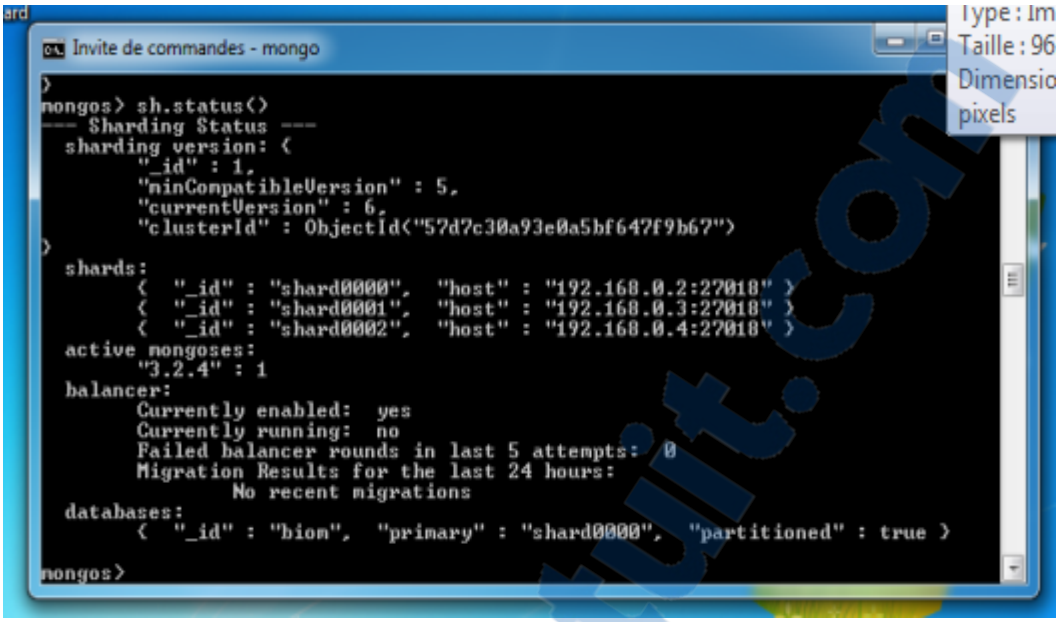

Figure III.31 : Activation réussi du Sharding pour la base biom

Voilà donc notre environnement Sharding distribué est configuré et prêt pour l'exploitation. Nous l'utiliserons une fois que les données sont migrées vers MongoDB.

5. Migration des données Oracle vers mongodb

Nous avons discuté dans la première partie de ce chapitre la migration d'une base de données relationnelle oracle vers la base NoSQL MongoDB et nous avons opté d'utilisé une approche en 3 étapes : passage par les fichiers Json puis l'import par l'outil mongoimport et conversion en BSON, comme il est schématisé dans la figure suivante :

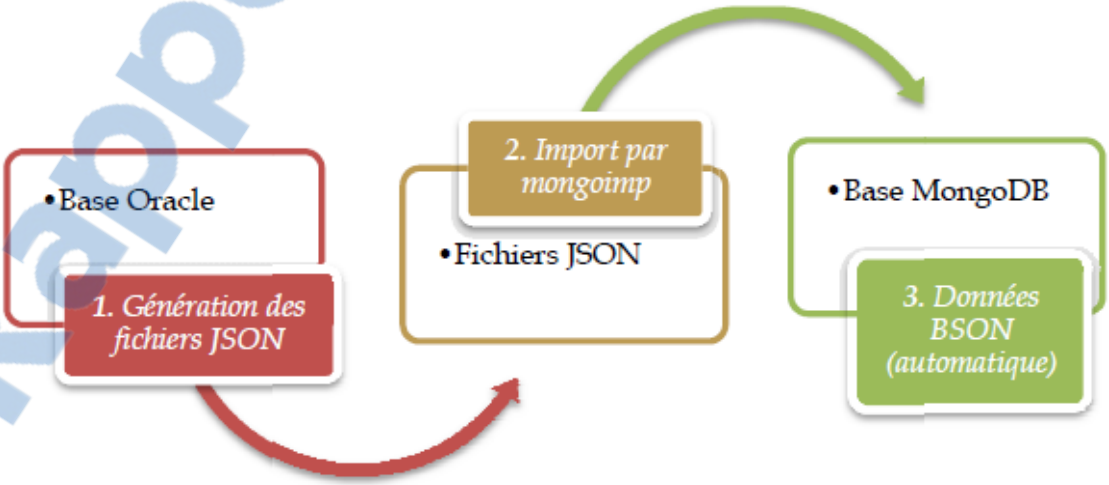

Figure III.32 : Schéma de l'approche de migration d'Oracle vers MongoDB

Nous détaillerons l'implémentation de ces trois étapes dans ce qui suit.

5.1. Génération de fichier JSON (Application *oracle-json*)

Pour réaliser cette 1ère étape, nous avons développé une application appelée *oracle-json sous* DELPHI-7 Cette application permet d'extraire les données d'une table Oracle (ou de plusieurs tables), et de générer automatiquement les données en format JSON vers des fichiers choisis. Une fois l'application est lancée nous pouvons se connecter à une base Oracle en spécifiant ses paramètres de connexion (voir la Figure III.33).

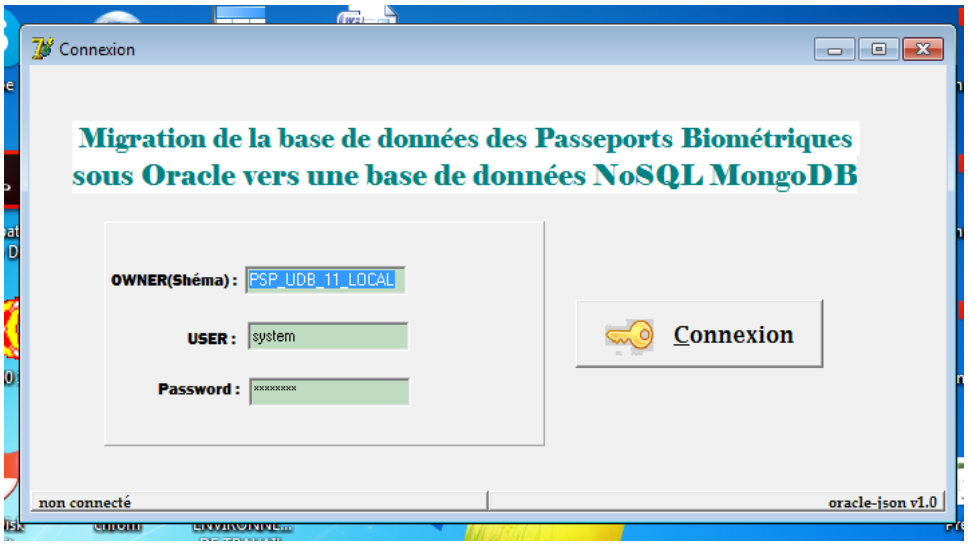

Figure III.33 : Interface de l'application Oracle-json

Une fois connecté, la fenêtre affiche la liste des tables oracle et son nombre total. Il suffit de choisir une ou plusieurs tables par les boutons affichés en les glissant dans la fenêtre des listes sélectionnées ou pour visualiser les données de certaine table suivant les figures suivantes :

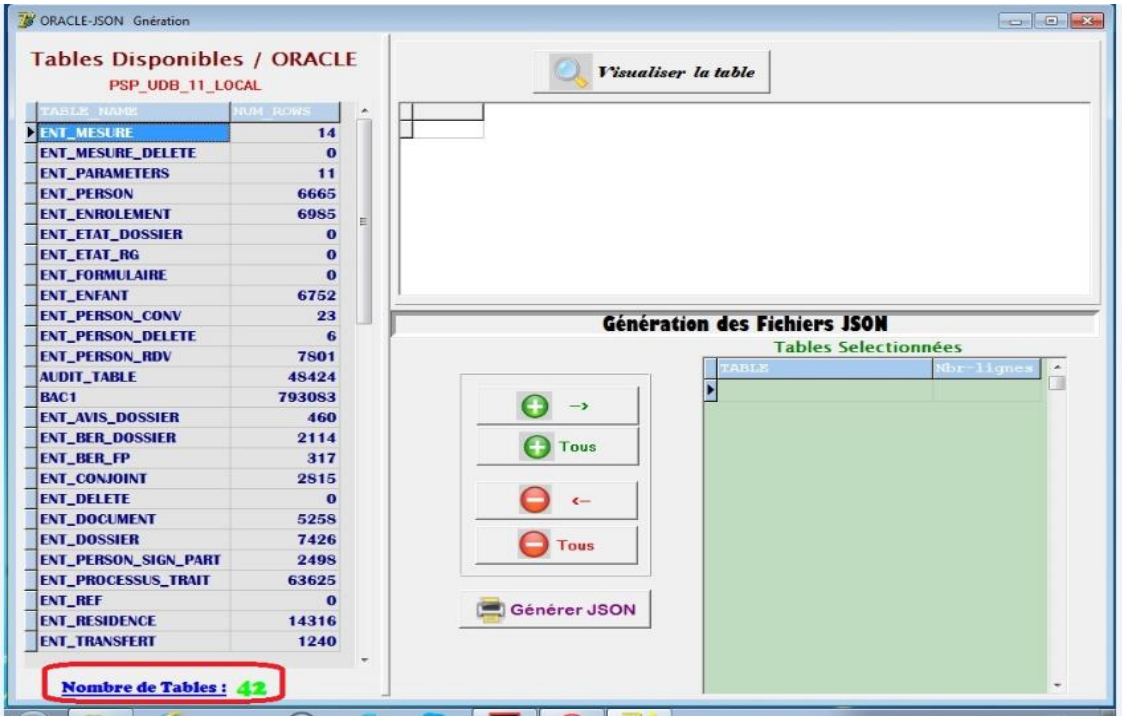

Figure III.34 : Sélection des tables pour générer les fichiers JSON ou visualiser

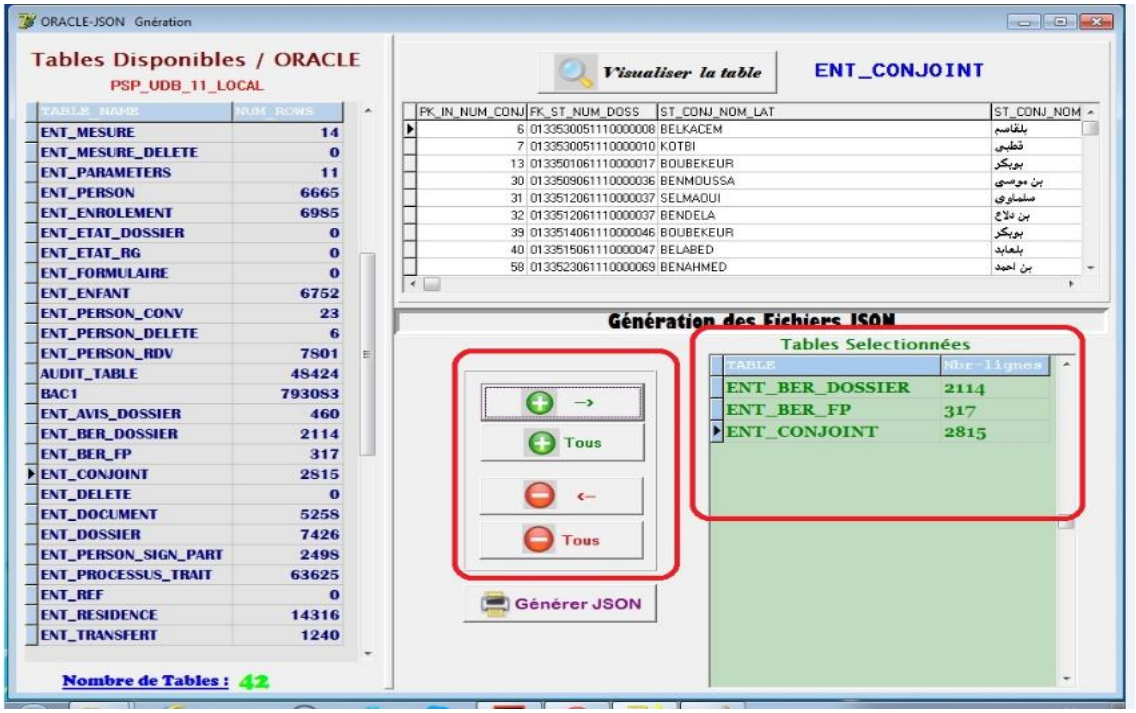

Figure III.35 : Sélection des tables pour la génération

Il suffit ensuite de cliquer sur le bouton « Générer JSON » pour générer de(s) fichier(s) de données JSON et lancer l'opération de génération, Le nom de(s) fichier(s) est par défaut le nom de(s) table(s).

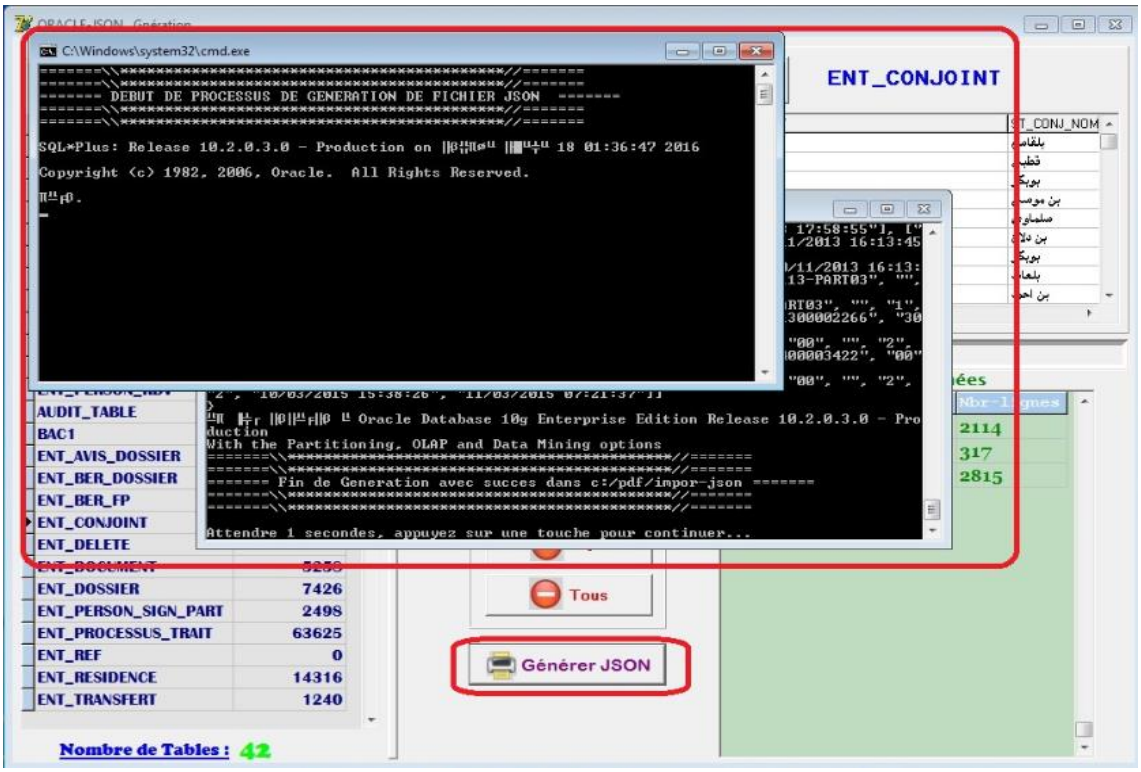

Figure III.36 : génération en cours des fichiers json.

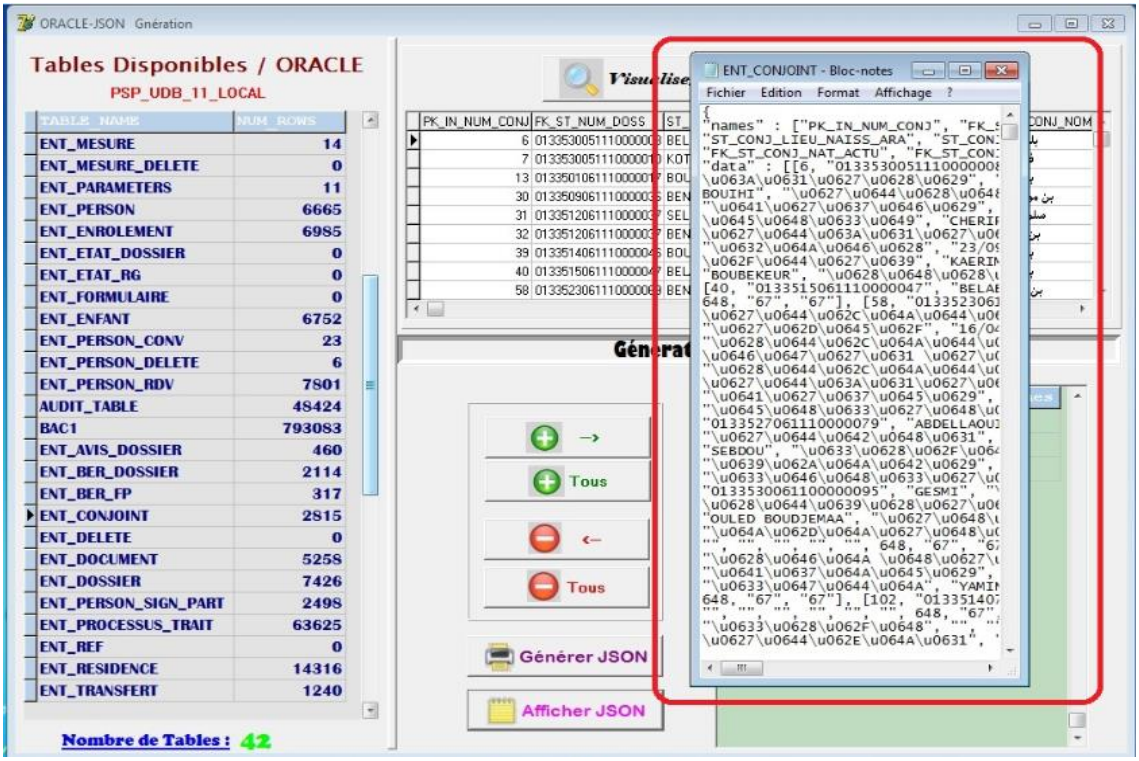

Figure III.37 : Un extrait d'un fichier JSON généré par oracle-json

5.2. Import des données JSON vers MongoDB

Une fois que les fichiers JSON générés, il suffit d'utiliser l'outil *mongoimport* qui se trouve dans le répertoire *bin* de l'installation MongoDB. Cet outil charge un seul fichier JSON dans une collection MongoDB.

La syntaxe est : ***./mongoimport -db BDD -c Collection --file fichier.json***

Par exemple, pour le charger dans notre base du fichier *ENT_USERS.json* :

./mongoimport -db biom -c ENT_USERS --file ENT_USERS.json

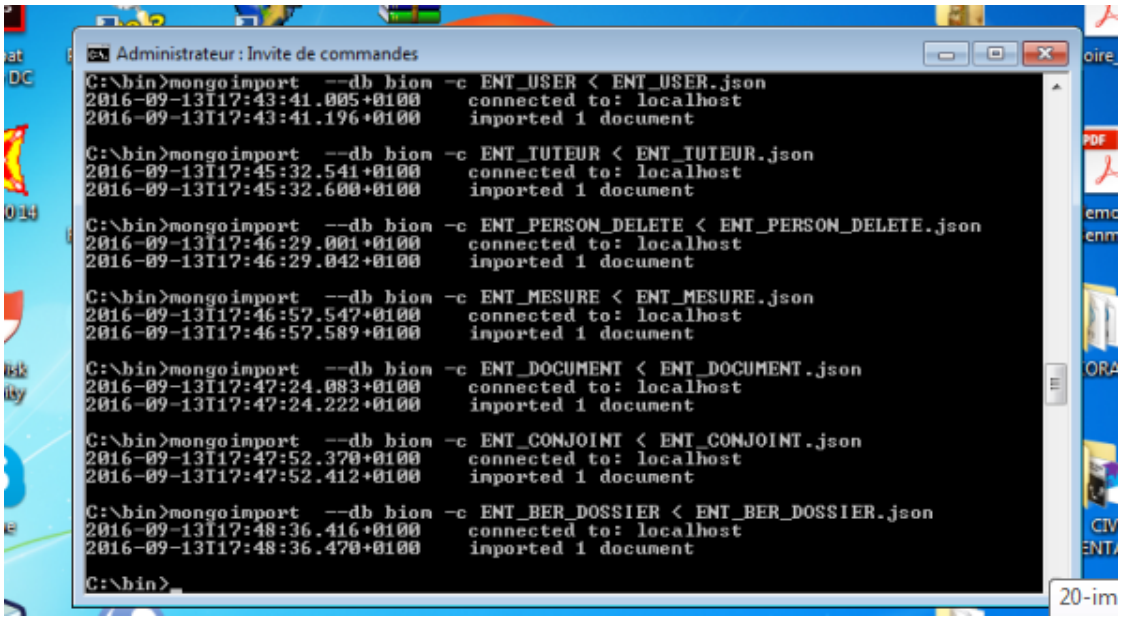

Figure III.38 : importation des fichiers json

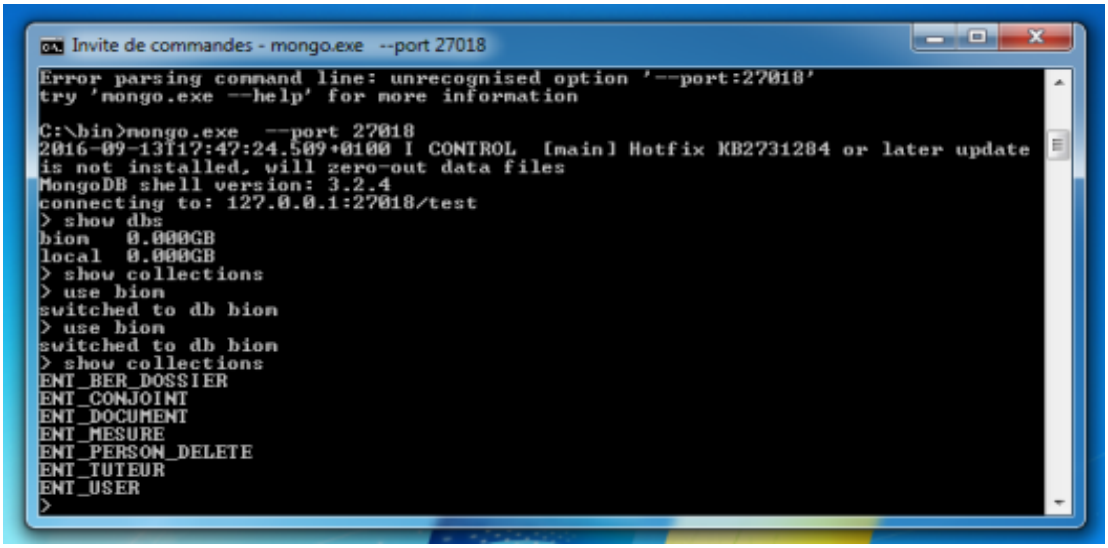

Figure III.39 : les tables importer sur shard 1

6. Exploitation dans l'environnement distribué

Notre base de données MongoDB *biom est* maintenant prête à l'exploitation. Nous pouvons exploiter le Sharding MongoDB sur des collections, tester les performances et analyser les résultats obtenus.

6.1. Test et exploitation du Sharding

Nous commençons par tester le Sharding pour des collections MongoDB. Nous allons voir un exemple appliqué sur une collection *ENT_USERS qui* contient un seul attribut CODE_*USER.* Etant connecté par l'interpréteur *mongo* à la 3ème machine qui exécute le service *Mongos* nous pouvons lancer le Sharding de cette collection en choisissant une clé de Sharding. Pour rappel la clé du Sharding d'une collection permet de distribuer les données de cette collection dans les noeuds en se basant sur les valeurs de cette clé.

Pour notre collection *ENT_USERS,* nous définissons l'attribut *CODE_ USER.* comme clé du Sharding par la commande *sh.shardCollection().*

Une fois l'opération réussie, nous allons insérer un ensemble de *3000* données dans la collection *ENT_USERS* en utilisant une instruction répétitive de type boucle Java Script *for()*comme il est montré dans la Figure.

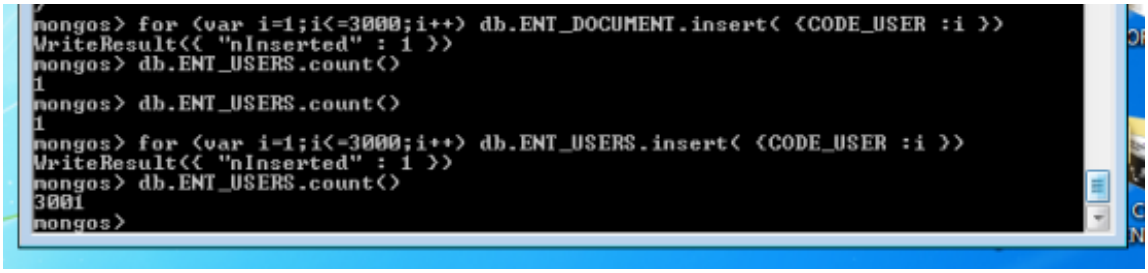

Figure III.40 : Insertion des données dans la collection ENT_USERS

Les 3000 instances (ou *shunk*s) sont insérées dans la collection, nous devons maintenant s'assurer que la distribution a été bien faite dans les trois (03) nœuds *shard1*, *shard2* et *shard3*. Nous utilisons la commande *getShardDistribution()* de la collection.

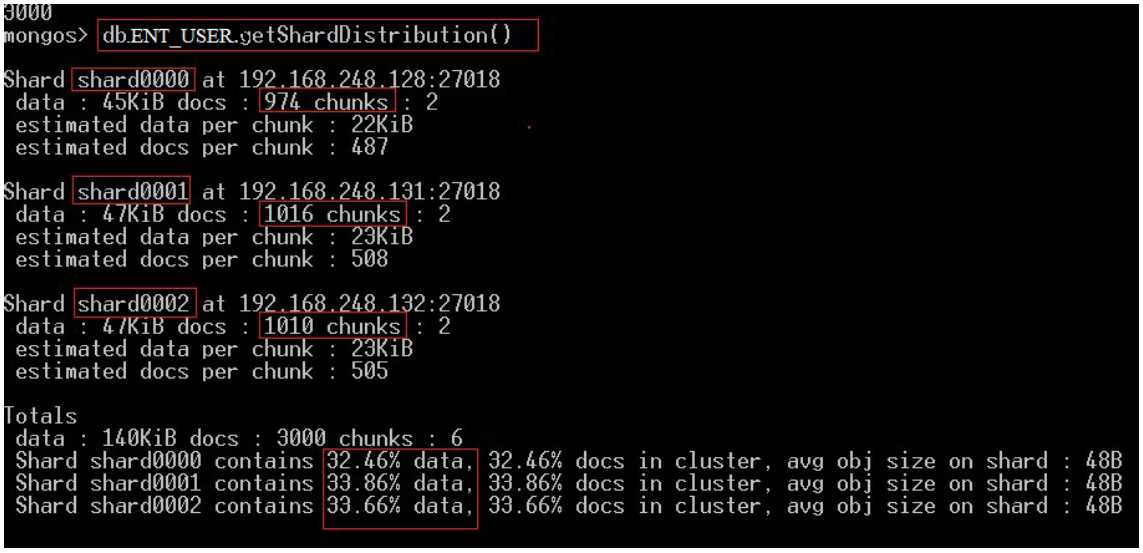

Figure III.41 : Vérification du Sharding sur la collection ENT_USER

Nous remarquons que les données ont été bien distribuées entre les 3 nœuds de notre environnement configuré : le 1er *shard* contient 974 chunks (32.46%), le 2ème *shard* contient 1016 chunks (33.86%), et le 3ème *shard* contient 1010 chunks (33.66%).

6.2 Résultats de comparaison performance

Dans cette section, nous présentons quelques résultats importants obtenus en comparant les performances entre oracle et MongoDB pour notre base de données migrée, parmi les facteurs de comparaison nous avons choisis l'élasticité et le temps d'exécution.

Elasticité :

Le gain principal de performance obtenu est bien l'élasticité qui correspond à la possibilité d'utiliser plusieurs nœuds pour distribuer les données, et partager les traitements entre eux.

La Figure III.42 montre la comparaison entre les deux cas de déploiement de notre base de données dans MongoDB et dans Oracle pour le stockage et le traitement.

Le graphe de cette figure montre que pour MongoDB, les 3 nœuds partagent les données et participent aux traitements des requêtes. Par contre et pour oracle, seul un nœud prend en charge le stockage et le traitement, et les deux autres restent inexploités.

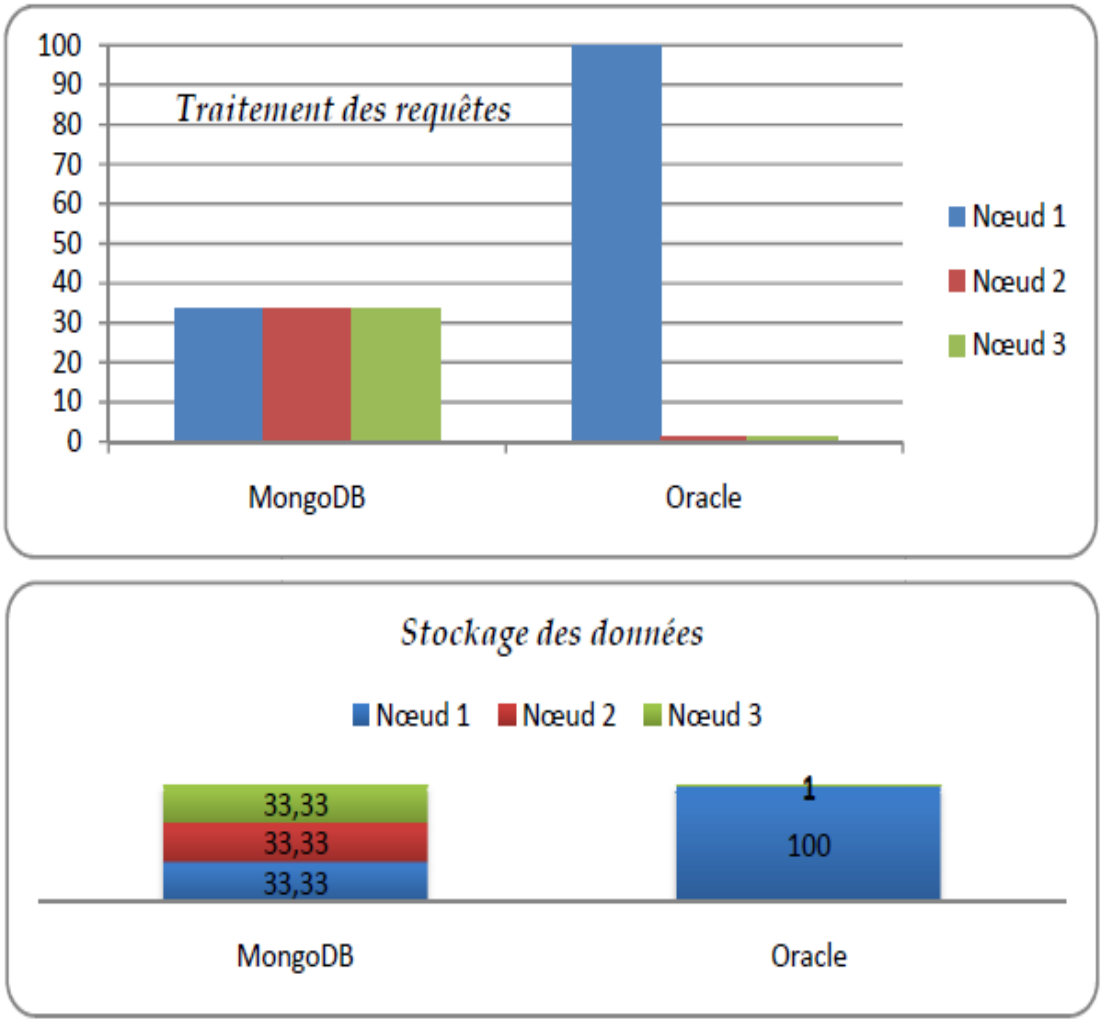

Figure III.42 : Comparaison d'élasticité entre MongoDB et Oracle entre les 3 nœuds

Temps d'exécution :

Le temps d'exécution des requêtes dépend fortement de la nature de ces requêtes.

Certains requêtes pour le relationnel peuvent être plus rapide que celle NoSQL et vice versa. Donc, la comparaison de temps d'exécution entre MongoDB et Oracle est strictement relative à la nature des requêtes et la façon dont sont modélisées les collections incluses.

7. CONCLUSION

Dans ce chapitre, nous avons présenté le domaine d'étude, la solution NoSQL adoptée ainsi que l'explication de l'approche adoptée pour la migration des données du système relationnel Oracle vers le système NoSQL MongoDB.

Aprés avoir présenté les tables concernées, nous avons installé MongoDB et créé la base de données cible.

Ensuite, nous avons installé et configuré la plateforme distribuée avec les nœuds nécessaires avec le Sharding de MongoDB. Dans cet environnement, nous avons réalisé notre expérimentation en utilisant notre application OracleJson.

Des tests d'exploitation ont été accomplis pour mettre en place la répartition des données entre les nœuds et comparer les performances des résultats obtenus.

Les résultats de l'étude de performance montrent que la solution utilisée est très intéressante et même elle donne, dans plusieurs cas, des performances plus intéressantes par rapport à celle d'Oracle, soit en temps de réponse ou soit en équilibrage de charges entre les sites.

Conclusion générale

Actuellement, le NoSQL est une technologie qui émerge en puissance, il est mis en œuvre dans des environnements manipulant de grandes masses de données tels que Google, Yahoo, Twitter, Facebook, etc. Les moteurs de recherche sont les premiers utilisateurs de ces technologies puisqu'ils ont besoin d'une grande puissance de stockage et de traitement de ces volumes de données, de même pour les réseaux sociaux qui gère une très grosse montée en charge dû au grand nombre d'utilisateurs et de requêtes simultanées.

Ce travail est motivé par des carences des systèmes de bases de données relationnels. Dans un contexte centralisé, il est reproché à ces systèmes leur faible niveau de flexibilité face à la gestion des objets hétérogènes (les attributs d'un objet doivent être déclarés et connus d'avance), ce qui n'offre pas un grand facteur d'évolutivité. L'autre limite majeure est l'utilisation des jointures qui n'est pas toujours évidente et optimale en cas de forte volumétrie. Dans un contexte distribué, il est difficile d'appliquer des propriétés ACID, notamment la cohérence, ce qui ne permet pas l'élasticité de ces bases dans un tel contexte.

L'objectif de ce mémoire étant la proposition d'une approche de migration de la base de données relationnelle des passeports biométriques, du ministère de l'intérieur et des collectivités locales (MICL), sous Oracle vers une base de données NoSQL orientée document MongoDB dans un environnement distribué. L'exportation des données relationnelles vers un format reconnu par les deux systèmes était la clé de réussite de notre solution. L'importation des données à partir des fichiers JSON vers MongoDB était la deuxième phase du processus de migration. La configuration de MongoDB dans un environnement complètement distribué s'est appuyée sur la réplication et le sharding.

Enfin, il s'agira là, d'une démarche puissante qui aidera non seulement, à faire connaître la technologie NoSQL, mais aussi, à vulgariser l'utilisation du NoSQL qui reste malgré tout peu connu. On peut estimer que l'objectif tracé au préalable a été

atteint, néanmoins plusieurs axes de recherches peuvent être développés dans le futur, comme la migration des métadonnées, qui n'a pas été traité dans ce travail, aussi les différents liens entre les tables de la base de données relationnelles. Cette migration peut être déployée dans le futur dans les autres applications opérationnelles du MICL qui sont sensibles à un passage à l'échelle très rapidement comme par exemple l'application de la gestion d'état civil en se limitant pas aux extraits de naissance mais aux différents actes administratifs délivrés par la commune et la gestion des cartes grises puisque notre parc automobile est en hausse permanente.

REFERENCES BIBLIOGRAPHIQUES

[1] Olivier Losson, Introduction aux Systèmes de Gestion de Bases de Données Relationnelles, cours Master Sciences et Technologies, Université Lille1.

[2] Michel Grech, Bases de données relationnelles & SQL, Formation de Bases de Données, Infotique, PARIS.

[3] Georges Gardarin, Bases de données, Editions Eyrolles, 5e tirage 2003.

[4] Christian Soutou, apprendre SQL avec MySQL, Editions Eyrolles, 2006.

[5] R. F.errere, initiation aux bases de données modélisation et langage SQL Master 2 Modélisation Statistique, Laboratoire de Mathématiques CNRS - Université de Franche-Comté, 2015/2016.

[6] Philippe Rigaux, Cours en ligne de bases de données, 13 juin 2001, disponible sur : http://sql.bdpedia.fr/

[7] Laurent Audibert, Base de Données et Langage SQL, Institut Universitaire de Technologie de Villetaneuse, département informatique, 1re année.

[8] Xavier Maletras, Le NoSQL- Cassandra, Thèse Professionnelle, université paris, 27/05/2012.

[9] Kouedi Emmanuel, Approche de migration d'une base de données relationnelle vers une base de données NoSQL orientée colonne, Mémoire master informatique, Université de Yaoundé I, mai 2012.

[10] Soufiane Rital, Not only SQL Introduction, cours pour Telecom Paris Tech, Institut Telecom CNRS LTCI, paris, 2012.

[11] Adriano Girolamo Piazza, NoSQL Etat de l'art et benchmark, Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES, Haute École de Gestion de Genève, 9 octobre 2013.

[12] Matteo Di Maglie, Adoption d'une solution NoSQL dans l'entreprise, Travail de Bachelor réalisé en vue de l'obtention du Bachelor, Haute École de Gestion de Genève, 12 septembre 2012.

[13] Meyer Léonard, l'avenir du NoSQL, 2014. [Online] disponible sur : http://www.leonardmeyer.com/wp-content/uploads/2014/06/avenirDuNoSQL.pdf

[14] Mathieu ROGER, Base NOSQL, Synthèse d'étude et projets d'intergiciels, l'Unité de Formation et de Recherche en Informatique, Mathématiques et Mathématiques Appliquées de Grenoble, France.

[15] http://davidmasclet.gisgraphy.com/post/2010/06/09/10-minutes-pour-comprendre-NoSQL Date d'accès : 14/08/2016

[16] Hadrien Furrer, SQL, NoSQL, NewSQL stratégie de choix, Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES, Haute École de Gestion de Genève, 16 Février 2015.

[17] http://www.croes.org/gerald/blog/qu-est-ce-que-rest/447/
Date d'accès : 18/08/2016

[18] http://michaelmorello.blogspot.com/2012_06_01_archive.html
Date d'accès : 20/08/2016

[19] http://blog.xebia.fr/2010/04/30/nosql-europe-bases-de-donnees-orientees-documents-et-mongodb/
Date d'accès : 23/08/2016

[20] Rudi Bruchez, Les bases de données NoSQL et le big data, éditions eyrolles, 2eme édition, 2015.

[21] http://www.marginweb.com/blog/20121110/quand-et-pourquoi-utiliser-une-base-de-donnees-nosql
Date d'accès : 25/08/2016

[22] http://www.infoq.com/news/2012/02/NoSQL-Adoption-Is-on-the-Rise/
Date d'accès : 26/08/2016

[23] Jack Vaughan, NoSQL : le choix difficile de la bonne base, News and Site Editor.
Date d'accès : 29/08/2016

[24] http://www.lemagit.fr/conseil/Quel-SGBD-NoSQL-pour-vos-besoins-IT-Criteres-de-choix
Date d'accès : 29/08/2016

[25] Dahmani Djilali, migration d'une base de données relationnelle vers une base de données nosql dans le cloud, Mémoire Pour l'Obtention du Diplôme de Post Graduation Spécialisée, université d'Oran, 2015.

[26] http://www.silicon.fr/base-donnees-nosql-impose-sgbdr-93305.html
Date d'accès : 31/08/2016

[27] https://blog.arismore.fr/arismore-choisit-mongodb-pour-la-performance-et-la-simplicite/
Date d'accès : 01/09/2016
[28] Philippe Rigaux, Bases de données documentaires et distribuées, Cours en ligne, 15 Février 2016, Disponible sur : http://b3d.bdpedia.fr/

Liste des figures

Liste des tableaux

Résumé

Une base de données classique ne peut pas prendre en charge l'évolution et la montée en charge des données dans une architecture relationnelle. Ces données ne peuvent pas être distribuées dynamiquement et efficacement entre les nœuds. Ce problème d'élasticité provient de la rigidité du modèle relationnel.

Face à cette problématique, nous proposons une approche permettant de faire migrer une base de données relationnelle de la gestion des passeports biométriques sous Oracle vers un système NoSQL MongoDB. Une application a été développée et déployée pour la migration, et des scénarios de test ont été pris pour confirmer l'élasticité des données et comparer les performances entre les deux systèmes sous Oracle et sous MongoDB.

Mots clés : Relationnel, NoSQL, Oracle, MongoDB, Sharding, Migration, JSON.

Abstract

A typical database can not support the evolution and scalability of data in a relational architecture. These data can not be dynamically and efficiently distributed between the nodes. This problem of elasticity comes from the rigidity of the relational model.

Faced with this problem, we propose an approach for migrating a relational database management biometric passports under Oracle to a NoSQL system MongoDB. An application has been developed and deployed for migration, and test scenarios were taken to confirm the elasticity data and compare performance between the two systems under Oracle and MongoDB

Keywords : Relational, NoSQL, Oracle, MongoDB, Sharding, Migration, JSON.

ملخص:

لا يمكن لقاعدة بيانات تقليدية أن تدعم تطور وتدرجية البيانات في هندسة علائقية. هده البيانات لا يمكن أن توزع بشكل حيوي وفعال بين العقد. مشكل المرونة هذا ناتج عن صلابة النموذج العلائقي. في مواجهة هذه المشكلة نقترح منهجية لترحيل قاعدة البيانات العلائقية الخاصة بجوازات السفر البيومترية من النوع اوراكل إلى نظام نوأسكل منقودبي. تم تطوير تطبيق من اجل نقل البيانات، تم اخذ حالات اختبار لتأكيد مرونة البيانات ومقارنة الأداء بين اوراكل ومنقودبي.

الكلمات المفتاحية:

علائقي ، نوأسكل، أوراكل، مونقودبي، التقسيم، ترحيل، جيزون.