RÉPUBLIQUE DU SÉNÉGAL

Un peuple - un but - une foi

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR, DE LA RECHERCHE ET DE L'INNOVATON

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

L'excellence ma référence

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

Mention : Management des systèmes d'information Automatisés

Département : Économie Gestion

UFR : Sciences Économiques et Sociales

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

SUJET :

Systèmes de Gestion des Bases de Données Relationnelles et NoSQL : Généralité, Etude Comparative et Techniques de migration.

¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

Présenté par :

Pape Mamadou SOW

Sous la direction de :

Dr Edouard Ngor SARR

Sous la supervision de :

Prof Ousmane SALL

Membres du jury :

Prof Abdou Aziz NIANG (**Président**)

Dr Bala Moussa BIAYE (**Examinateur**)

Dr Lamine FATY (**Examinateur**)

Année universitaire : 2024–2025

Soutenu publiquement le 05/07/2025

Dédicace

Je dédie ce mémoire à toute ma famille, pour son amour, son soutien indéfectible et ses encouragements constants.

Remerciements

Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce mémoire :

- A mon directeur de mémoire Docteur Edouard Ngor SARR, pour sa disponibilité, ses conseils avisés, sa rigueur et son accompagnement tout au long de ce travail. Son encadrement m'a été d'un soutien précieux, tant sur le plan académique que méthodologique ;
- Au Professeur Ousmane SALL pour le suivi et l'évaluation de mon travail. Merci à vous cher professeur ;
- Aux membres du jury qui ont accepté d'évaluer ce modeste travail ;
- Au corps professoral de la filière Management informatisé des organisations (MIO), pour la qualité des enseignements dispensés durant ma formation, qui ont nourri ma réflexion et consolidé mes compétences ;
- A mes amis et camarades de classe, pour les échanges, l'entraide et les moments de partage qui ont enrichi cette aventure universitaire ;
- Une pensée toute particulière et remplie d'émotion va à la mémoire de mon grand-père Bougouma NDAO, dont les valeurs, le courage et la sagesse continuent de m'inspirer. Ce travail est aussi un hommage à son souvenir.

À toutes ces personnes, j'adresse ma reconnaissance la plus sincère.

Résumé

Ce mémoire s'intéresse à la problématique de la migration entre deux types majeurs de systèmes de gestion de bases de données : les systèmes relationnels (SGBDR) et les bases de données NoSQL. Ces deux paradigmes, bien que complémentaires, reposent sur des modèles conceptuels et techniques très différents. L'objectif principal de ce travail est, après une étude comparative de ces deux systèmes, de concevoir un outil de migration bidirectionnelle fiable, flexible et reproductible. En s'appuyant sur des scripts en Python, nous avons développé une solution permettant de transférer des données entre ces systèmes tout en conservant leur intégrité, leur structure et leur logique métier. Une étude de cas a été menée afin de tester et évaluer les performances de cette migration dans des conditions réelles. Le mémoire s'appuie sur une analyse théorique approfondie, un état de l'art comparatif, une phase de mise en œuvre pratique et une évaluation des résultats. Ce travail vise à apporter des réponses concrètes aux professionnels, étudiants et chercheurs confrontés aux défis de transition entre modèles relationnels et NoSQL.

Mots-clés : SGBDR, NoSQL, MySQL, MongoDB, Migration, Python

Abstract

This master's thesis focuses on the challenge of migrating between two major types of database management systems: relational database management systems (RDBMS) and NoSQL. These two paradigms, though complementary, are built upon vastly different conceptual and technical models. The primary objective of this work, following a comparative study of these two systems, is to design a reliable, flexible, and reproducible bidirectional migration tool**.** By leveraging Python scripts, we developed a solution to transfer data between these systems while preserving its integrity, structure, and business logic. A case study was conducted to test and evaluate the performance of this migration under real-world conditions. This thesis is based on an in-depth theoretical analysis, a comparative state-of-the-art review, a practical implementation phase, and an evaluation of the results. This work aims to provide concrete answers for professionals, students, and researchers facing the challenges of transitioning between relational and NoSQL models.

Keywords : SGBDR, NoSQL, MySQL, MongoDB, Migration, Python

Sommaire

Dédicace i
Remerciements ii
Résumé iii
Abstract iv
Sommaire v
Liste des figures vi
Liste des tableaux viii
Sigles et abréviations ix
Introduction générale 1
Chapitre 1 : Généralité sur les SGBDR 4
Chapitre 2 : Généralité sur les NoSQL 29
Chapitre 3 : Migration SGBDR et NoSQL 51
Chapitre 4 : Etude de cas MySQL et MongoDB 70
Conclusion 88
Références 90
Table des matières 94
Annexe 1 : Code de connexion MySQL avec Python 98
Annexe 2 : Code de connexion MongoDB avec Python 99
Annexe 3 : Code de la fonction pour lister les tables MySQL 100
Annexe 4 : Code pour la migration d'une table vers MongoDB 101
Annexe 5 : Code pour la migration de toutes les tables vers MongoDB 103
Annexe 6 : Code pour la migration personnalisée 104
Annexe 7 : Code du fichier main 104
Annexe 8 : Code de connexion pour la deuxième phase de migration 105
Annexe 9 : Code pour lister les collections 106
Annexe 10 : Code de migration d'une collection vers MySQL 107
Annexe 11 : Code de migration de toutes les collections vers MySQL 108
Annexe 12 : Code pour une migration personnalisée vers MySQL 109
Annexe 14 : Code de l'interface 110

Liste des figures

Figure 1. La relation entre le langage SQL et le SGBD lors d'une action sur l'application ... 7
Figure 2. Théorème CAP ... 33
Figure 3. Aperçu des données dans une base orientée colonne [42] ... 36
Figure 4. Aperçu des données dans une base orientée document [32] ... 36
Figure 5. Aperçu des données dans une base de données type clé-valeur [21] ... 38
Figure 6. Structure de NoSQL orienté graph ... 39
Figure 7. Modèle base de donnée NoSQL de type graphe [32] ... 39
Figure 8. Processus de migration de données ... 53
Figure 9. Conversion physique directe [34] ... 54
Figure 10. Conversion conceptuelle [34] ... 54
Figure 11. Utilisation de Wrappers [36] ... 55
Figure 12. Migration par framework ... 56
Figure 13. Technique par utilisation d'outils spécialisés ... 58
Figure 14. Migration par transformation directe des requête ... 59
Figure 15. Technique par ETL ... 61
Figure 16. Migration par conversion de schémas ... 62
Figure 17. Technique par migration schémas et mappage ... 64
Figure 18. Migration par algorithme automatisé ... 65
Figure 19. Migration par API et scripts ... 67
Figure 20. Structuration de la base de données MySQL ... 75
Figure 21. Structuration des tables clients et commandes ... 75
Figure 22. Structuration des tables ligne_commandes et produits ... 76
Figure 23. Structuration de la base de données MongoDB ... 76
Figure 24. Structuration des collections departements et employes ... 76
Figure 25. Structuration des collections projets et salles ... 77
Figure 26. La page d'accueil de l'interface MySQL vers MongoDB ... 78
Figure 27. Option de lister les tables MySQL ... 79
Figure 28. Migration de la table « clients » vers MongoDB ... 79
Figure 29. Aperçu de la nouvelle collection « clients » dans MongoDB ... 80
Figure 30. Images d'une migration complète MySQL vers MongoDB ... 80
Figure 31. Affichage de toutes les tables migrées dans MongoDB ... 81

Figure 32. Migration personnalisée (colonne). 81
Figure 33. Affichage de la colonne migrée dans MongoDB 82
Figure 34. Page d'accueil de la section MongoDB vers MySQL 82
Figure 35. Les collections existantes dans « entreprise » de Mongo 83
Figure 36. Migration de la collection « salles » vers MySQL 83
Figure 37. Aperçu de la nouvelle table « salles » dans MySQL 84
Figure 38. Migration complète MongoDB vers MySQL 84
Figure 39. Aperçu des nouvelles tables en provenance de MongoDB 85
Figure 40. Migration des champs « nom » et « budget » vers MySQL 86
Figure 41. La nouvelle table « projets » dans MySQL 86

Liste des tableaux

Tableau 1. Exemple de relations, colonnes et ligne 9
Tableau 2. Tableau récapitulatif de l'algèbre relationnelle 14
Tableau 3. Tableau récapitulatif des cas d'usage du SGBDR 18
Tableau 4. Tableau récapitulatif de ces catégories d'instructions 21
Tableau 5. Points forts et limites du SGBDR 27
Tableau 6. Exemple de données stockées [32] 35
Tableau 7. Tableau récapitulatif des modèles avec des exemples de base de données 40
Tableau 8. Tableau récapitulatif de l'étude comparative 50

Sigles et abréviations

- **SGBDR :** Système de gestion des bases de données relationnelles
- **BDD :** Base de données
- **NoSQL**: Not Only Structured Query Language
- **SQL** : Structured Query Language
- **IBM :** IBM (International Business Machines Corporation) est une entreprise pionnière dans le développement des systèmes de gestion de bases de données.
- **DEC :** Digital Equipment Corporation, une société informatique qui a contribué au développement des SGBD.
- **SYBASE :** Sybase a été fondée en 1984 et a développé le système de gestion de bases de données Adaptive Server Enterprise (ASE).
- **E. F. Codd :** Edgar Frank Codd est le mathématicien et informaticien britannique qui a introduit le concept de base de données relationnelle en 1970.
- **CAP :** le théorème CAP (Consistance, Disponibilité, Tolérance au Partitionnement).
- **HDFS:** Hadoop Distributed File System
- **HBASE:** Hadoop Database
- **ETL** : Extract–Transform–Load.
- **BASE**: Basically Available, Soft-state et Eventual Consistency
- **WAMP** : Acronyme de APACHE, MYSQL et PHP
- **ERP** : Enterprise Resource Planning
- **ORM** : Object-Relational Mapping
- **RGPD :** Règlement général sur la protection des données
- **SDD** : Sénégal Dem Dikk

Introduction générale

Une base de données est un système structuré permettant de stocker, d'organiser, de gérer et de mettre à jour efficacement les informations. Elle joue un rôle central dans la gestion des données au sein des organisations. Avant l'émergence des bases modernes, les systèmes traditionnels étaient centrés sur les traitements, chaque application gérant ses propres fichiers de données. Cette approche entraînait une forte dépendance entre programmes et données, une duplication fréquente des informations, des formats hétérogènes et des conflits lors des accès simultanés. Ces systèmes, fondés sur des modèles hiérarchiques ou en réseau, étaient rigides et peu adaptés à des besoins variés [1]. C'est pour pallier ces limites que les bases de données relationnelles ont vu le jour dans les années 1970. Reposant sur le langage SQL et une organisation en tables interconnectées, elles ont rapidement été adoptées pour leur souplesse, leur normalisation et leur efficacité. Pendant de longues années, les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR) ont dominé le paysage technologique grâce à leur robustesse, leur fiabilité et le respect strict des règles d'intégrité. Ils se sont imposés dans des domaines exigeants tels que la finance, la santé ou le commerce, où la structuration rigoureuse des données et la gestion des relations complexes sont essentielles [2]. Toutefois, l'avènement des big data et des applications modernes, traitant des volumes importants de données non ou semi-structurées (comme des documents, des vidéos ou des journaux d'événements), a révélé les limites des SGBDR, notamment en matière de scalabilité et d'adaptabilité.

Pour répondre à ces nouveaux besoins, les bases de données NoSQL se sont imposées comme des alternatives modernes, mieux adaptées à la gestion de données massives dans des environnements distribués. Là où les SGBDR excellent dans les contextes nécessitant cohérence et structure, les NoSQL privilégient la performance et la flexibilité, parfois au détriment de la standardisation et de la cohérence forte. Cette évolution des besoins conduit naturellement à s'interroger sur les stratégies de transition entre ces deux paradigmes. Si la migration de SGBDR vers NoSQL est fréquente, l'inverse, le retour vers les bases relationnelles devient également pertinent, notamment pour bénéficier de la rigueur structurelle et des garanties d'intégrité qu'offrent les SGBDR.

Ainsi, un défi majeur se pose : **comment faire migrer efficacement des données entre ces deux types de systèmes**, **dans un sens comme dans l'autre, tout en préservant la cohérence, l'intégrité, la disponibilité et la pertinence métier des données ?**

Cette problématique soulève des enjeux techniques complexes tels que la transformation des structures, la gestion des types de données, la conservation des contraintes fonctionnelles, et la reproductibilité des migrations. Pour y répondre, ce mémoire adopte une approche mixte, c'est-à-dire une démarche combinant une étude comparative conceptuelle et théorique des paradigmes relationnel et NoSQL, avec une mise en œuvre pratique et méthodologique de la migration dans un contexte réel. Cette approche permet d'articuler les apports de la recherche et les contraintes du terrain, afin d'offrir une vision à la fois globale et opérationnelle du sujet.
Les objectifs poursuivis dans ce travail sont les suivants :

- **Analyser les paradigmes relationnel et NoSQL** : approfondir la compréhension des fondements, des architectures et des logiques sous-jacentes des deux types de bases, en mettant en évidence leurs complémentarités et spécificités ;
- **Identifier les cas d'usage typiques des systèmes SGBDR et NoSQL**, afin de mieux comprendre les contextes dans lesquels chacun s'illustre, en lien avec les exigences métiers, les types de données, et les contraintes de performance ;
- **Explorer les problématiques de transformation structurelle** : étudier les défis techniques liés à la conversion des modèles relationnels vers des modèles NoSQL (ex. : documents, colonnes, graphes, etc.), mais aussi les enjeux inverses dans le cadre d'un retour vers un modèle relationnel ;
- **Élaborer une méthodologie structurée et reproductible de migration de données,** en tenant compte des phases de préparation, de mapping, de transfert, de validation et de contrôle qualité ;
- **Mettre en œuvre cette méthodologie à travers une étude de cas réelle**, afin d'illustrer concrètement les étapes du processus, les choix techniques effectués, les outils mobilisés, ainsi que les bénéfices et les limites constatés ;

Ce travail revêt un intérêt :

- **Pour les entreprises** : Ce mémoire leur offre un guide pratique pour évaluer leurs besoins en matière de gestion de données et effectuer des transitions technologiques en toute sécurité ;
- **Pour moi** : Ce travail constitue une opportunité de renforcer mes compétences dans le domaine des bases de données, une expertise essentielle dans le monde technologique actuel. La compréhension approfondie des systèmes SGBDR et NoSQL, ainsi que la maîtrise des méthodologies de migration bidirectionnelle, me permettront d'enrichir

mon profil technique et d'élargir mes perspectives de carrière, notamment dans l'administration des bases de données ou la gestion des systèmes d'information.

- **Pour les chercheurs** : Cette étude enrichit la littérature existante avec une analyse comparative actualisée et une méthodologie structurée pour la migration des données entre paradigmes ;
- **Pour les étudiants et praticiens** : Ce travail sert de base pour comprendre les défis et opportunités associés à l'adoption ou au retour des bases de données NoSQL ou relationnelles.

Ainsi, ce mémoire s'organisera en quatre chapitres :

- **Chapitre 1 : Généralité sur les SGBDR** : Dans ce chapitre, nous présenterons les bases de données relationnelles, leur mode de fonctionnement, leur cas d'usage, leur système interrogation, certains types de base de données relationnels qui existent (les plus populaires), les points forts de ce système et ses limites ;
- **Chapitre 2 : Généralité sur les NoSQL** : Dans ce chapitre, nous présenterons les bases de données NoSQL, leur mode de fonctionnement, leur cas d'usage, leurs systèmes d'interrogations, les types de base de données NoSQL qui existent, les points forts de ces systèmes et leur limite ;
- **Chapitre 3 : Etude comparative** : Nous étalerons une étude comparative entre ces deux systèmes à base : de leur situation d'usage, de la volumétrie, leur type de scalabilité et en terme de performance suivi d'une analyse comparative et une partie de discussion ;
- **Chapitre 4 : Migration SGBDR et NoSQL** : Ce dernier chapitre est consacré à la présentation des types de migrations de données qui existent (les formes), en faire un choix pour notre étude de cas dans laquelle on fera une migration MySQL vers MongoDB et dans le sens inverse. Pour conclure ce chapitre on discutera de cette pratique.

Chapitre 1 : Généralité sur les SGBDR

1.1.Bases de données et les SGBD

Une base de données [1]est un système informatique qui permet de stocker, organiser, consulter et modifier des données de manière efficace. On peut le définir aussi comme étant un ensemble organisé d'informations, centralisées ou réparties, conçues pour répondre aux besoins d'une ou plusieurs applications. Ces données peuvent être consultées et modifiées par un groupe d'utilisateurs. En d'autres termes, une base de données regroupe des informations complètes, structurées, durables et sans redondance, portant sur un sujet spécifique [2].

Celle-ci joue un rôle très important dans les systèmes modernes. En effet, les avancées en matière de matériel informatique et de logiciels ont permis de développer des bases de données toujours plus performantes, capables de traiter de grandes quantités de données en un temps réduit tout en gardant la qualité. L'ensemble de ces données sont gérées par un système solide. Ainsi, un SGBD est un ensemble de logiciels système conçu pour gérer de grandes quantités de données, souvent stockées sur des mémoires secondaires telles que les disques magnétiques. Ces systèmes permettent aux utilisateurs d'insérer, modifier, rechercher et partager efficacement des données, même dans des bases atteignant plusieurs milliards d'octets [3]. Dans un SGBD, les utilisateurs peuvent interroger les données sur des critères simples (par exemple, les produits à moins de 100 francs) ou basés sur des relations complexes (comme les produits commandés par un client spécifique). De plus, plusieurs utilisateurs peuvent interagir simultanément avec les données, tout en ayant l'illusion qu'ils travaillent de manière exclusive grâce à la transparence du partage. A noter que le SGBD protège les données contre les incidents et garantit des performances optimales [4]. Celui-ci peut être relationnel (SGBDR) ou non relationnel (NoSQL).

1.2. Les SGBDR

Un SGBDR est un ensemble de programmes qui permettent d'interroger une base de données relationnelle [5]. Les SGBDR tiennent à atteindre des objectifs suivants :

- **L'administration simplifiée des données :** Un SGBD fournit des outils permettant de définir, décrire et modifier les structures des données ainsi que leurs représentations externes. Ces outils facilitent leur évolution en fonction des besoins des applications, rendant la gestion des données plus flexible et centralisée [3] ;

[1] **Donnée** : La **donnée** peut être définie comme une représentation élémentaire d'une information ou d'un fait brut, souvent utilisée comme matière première pour générer de l'information ou des connaissances après traitement.

- **L'efficacité des accès :** Le SGBD optimise les performances des requêtes en assurant un débit élevé (nombre de transactions par seconde) et des temps de réponse rapides. Cela inclut une gestion efficace des ressources matérielles et logicielles pour éviter les conflits entre utilisateurs et garantir une expérience fluide [4] ;
- **Le contrôle de la redondance :** Contrairement aux systèmes classiques où chaque application possède ses propres fichiers, le SGBD intègre les données en un fichier partagé. Cela réduit la duplication physique, économise de l'espace mémoire, et évite les mises à jour multiples, améliorant ainsi la cohérence et l'efficacité [3] ;
- **La cohérence des données :** Le SGBD applique des contraintes d'intégrité pour garantir que les données respectent des règles prédéfinies (par exemple, des plages de valeurs valides pour certains champs). Cela prévient les erreurs et assure une exactitude constante des informations [7] ;
- **Le partage simultané des données :** Les données peuvent être utilisées simultanément par plusieurs applications, sans conflit ni attente excessive. Chaque utilisateur a l'impression de travailler de manière exclusive, même lorsque d'autres modifient les données en temps réel [3] ;
- **La sécurité et fiabilité des données** : En effet, il prend en charge la protection contre les accès non autorisés, c'est-à-dire, des mécanismes de contrôle permettent de gérer les droits d'accès aux données pour prévenir les actions malveillantes. Mais aussi, la récupération après panne, c'est-à-dire, en cas d'incident matériel ou logiciel, le SGBD restaure les données à partir de sauvegardes. Il garantit également l'intégrité des transactions interrompues, par exemple en annulant les modifications incomplètes [3].

En résumé, les systèmes de gestion de base de données relationnel se distinguent de par leur capacité à centraliser et optimiser la gestion des données, tout en garantissant leur cohérence, leur partage sécurisé et leur accès rapide. Ils jouent un rôle clé dans les systèmes modernes en offrant des outils pour une administration efficace, une réduction des redondances, et une sécurité accrue face aux pannes ou aux accès non autorisés.

Exemple illustratif : Imaginez qu'un utilisateur arrive dans un supermarché. Il scanne un aliment présent pour connaître ses caractéristiques nutritionnelles. Que va faire notre application ? L'application va traduire cette recherche en SQL et l'envoyer au SGBD, qui va récupérer l'aliment en question dans le stockage de la base de données, pour ensuite le

redonner à l'application. L'utilisateur retrouvera ainsi son aliment avec toutes ses caractéristiques.

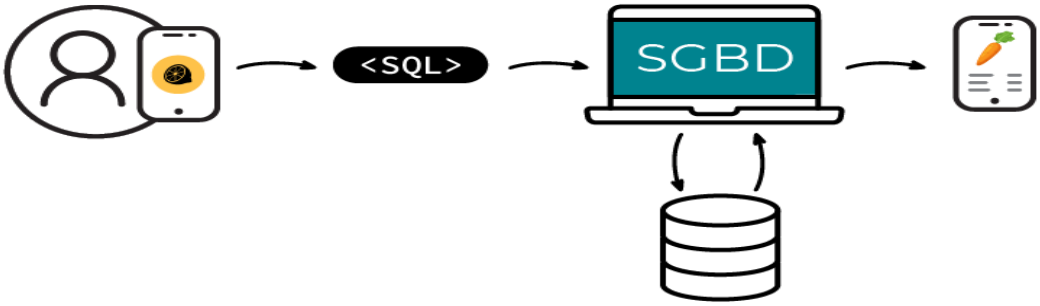

Figure 1. La relation entre le langage SQL et le SGBD lors d'une action sur l'application[2]

Les SGBDR est caractérisé par le modèle relationnel et l'algèbre relationnel.

1.2.1 Le modèle relationnel

Le modèle relationnel est une méthode, basée sur des principes mathématiques, pour organiser et lier diverses informations. Il permet de structurer les données et de définir les connexions entre-elles [5]. Le modèle relationnel, introduit par E. F. Codd en 1970, constitue une étape révolutionnaire dans l'histoire des bases de données. Son objectif initial était de fournir une approche ensembliste simple pour organiser et manipuler les ensembles d'enregistrements, tant au niveau de leur description que de leur gestion [6]. Aujourd'hui, ce modèle est à la base de nombreux systèmes de gestion de bases de données relationnelles (SGBDR) et a contribué à façonner les architectures modernes, notamment celles qui permettent l'accès aux données depuis des stations de travail. Le succès du modèle relationnel a dépassé les attentes, atteignant plusieurs objectifs fondamentaux fixés par son créateur, tout en s'adaptant aux évolutions technologiques.

Les objectifs initiaux du modèle relationnel étaient :

- **L'indépendance des données et des programmes** : Le premier objectif du modèle relationnel était d'offrir une séparation claire entre les programmes d'applications et la représentation interne des données. Cela inclut notamment la gestion des ordres d'implantation des données dans les fichiers et la prise en charge des index et des chemins d'accès. Cette indépendance a permis de simplifier le développement et la

[2] https://openclassrooms.com/fr/courses/6971126-implementez-vos-bases-de-donnees-relationnelles-avec-sql/7139618-decouvrez-le-systeme-de-gestion-de-base-de-donnees-sgbd

maintenance des applications, en les rendant moins dépendantes des modifications internes au stockage des données [4] ;

- **La cohérence et réduction de la redondance des données :** Le modèle relationnel a également été conçu pour résoudre des problèmes majeurs rencontrés avec les modèles hiérarchiques et en réseau que sont la cohérence des données qui est garantie par des règles d'intégrité intégrées au modèle et la réduction de la redondance est obtenue grâce à la structure relationnelle, où les données sont organisées sous forme de tables à deux dimensions. Cette simplicité rend les informations plus faciles à gérer et à comprendre [6].

A partir de là, d'autres objectifs non prévus au début sont atteints :

- **Développement de langages non procéduraux** : Le modèle relationnel a permis la création de langages de manipulation de données basés sur des théories mathématiques solides, notamment : l'algèbre relationnelle qui offre une manipulation intuitive des données à travers des opérateurs simples et formels, semblables aux opérateurs arithmétiques utilisés pour les nombres et les langages assertionnels qui permettent aux utilisateurs de spécifier les données souhaitées sans avoir à détailler la manière de les obtenir. Cette abstraction simplifie l'interaction avec la base de données, tout en réduisant la complexité des requêtes [7] ;
- **Extensibilité** : Bien que non prévu initialement, le modèle relationnel s'est avéré extensible. Il est capable de gérer non seulement des données tabulaires simples, mais aussi des données plus complexes, adaptées aux besoins modernes des applications. Cette flexibilité a permis son adoption dans divers domaines, des systèmes transactionnels aux applications analytiques [6] ;
- **Établissement d'un standard universel :** Le modèle relationnel est devenu une norme mondiale pour la description et la manipulation des bases de données. Les systèmes basés sur ce modèle sont aujourd'hui omniprésents et sont au cœur des infrastructures informatiques modernes [4].

Ainsi, le modèle relationnel a non seulement répondu aux défis initiaux de la gestion des données, mais il a également ouvert la voie à de nouvelles possibilités en matière de manipulation et de standardisation des bases de données. Sa simplicité et sa robustesse continuent de jouer un rôle clé dans l'évolution des systèmes d'information et des technologies de données. A noter qu'aussi y a des notions liées au modèle relationnel. On désignera celles-

ci tous éléments faisant son existence et son fonctionnement que sont : une table (ou relation), la notion de ligne (ou enregistrement) et colonne (ou attribut), la notion de relation entre les données, clé primaire et clé étrangère.

1.2.1.1 Une table, Colonne et ligne

Une table, également appelée relation, est une structure bidimensionnelle utilisée pour organiser et stocker les données dans un modèle relationnel. Elle est composée de [8] :

- **Colonnes** : aussi appelées attributs, qui représentent les différentes propriétés ou catégories d'informations stockées dans la table ;
- **Lignes** : appelées également enregistrements, n-uplets ou tuples, qui contiennent des ensembles de valeurs correspondant à chaque attribut.

Chaque intersection entre une colonne et une ligne contient une donnée spécifique. Une table peut contenir un nombre illimité de lignes, mais chaque ligne doit être unique et correspond à un enregistrement distinct. Par ailleurs, un champ désigne une cellule spécifique, c'est-à-dire la valeur individuelle à l'intersection d'un enregistrement (ligne) et d'un attribut (colonne). Pour résumé, une table est un ensemble structuré de données où chaque enregistrement renseigne les différentes valeurs associées à ses champs respectifs [9].
Le tableau ci-dessous montre une représentation d'une table de base de données relationnelle avec les colonnes et lignes.

Tableau 1. Exemple de relations, colonnes et ligne

Colonne=attribut

Id	Prénom	Nom
201800177	Pape M	SOW
201800188	Rassoul	FALL

Enregistrement=ligne

Valeur de l'attribut= champ

1.2.1.2 Clé candidate et clé primaire

Une clé candidate est un ensemble minimal d'attributs qui permettent d'identifier de manière unique chaque tuple (ligne) dans une relation (table) [5]. Cela signifie que :

- Les valeurs d'une clé candidate doivent être distinctes pour chaque tuple de la table ;
- Une table peut avoir une ou plusieurs clés candidates, et chaque clé candidate est un candidat potentiel pour devenir une clé primaire ;

Le caractère minimal signifie qu'aucun attribut supplémentaire n'est nécessaire pour garantir cette unicité [2]. Toute clé primaire est une clé candidate, mais toutes les clés candidates ne deviennent pas forcément une clé primaire. Le choix de cette dernière dépend des besoins du système et des contraintes opérationnelles, telles que la simplicité ou la rapidité des accès.

Tableau 2. Clé Primaire et clé candidate

Clé primaire

Clé candidate

Id	Prénom	Nom	email
201800177	Pape M	SOW	sowpape@gmail.com
201800188	Rassoul	FALL	rf@gmail.com

1.2.1.3 Clé étrangère

Une clé étrangère est un attribut ou un ensemble d'attributs d'une relation qui fait référence à la clé primaire d'une autre relation. Elle établit un lien entre les deux tables en assurant l'intégrité des données, car chaque valeur de la clé étrangère doit obligatoirement exister dans la clé primaire correspondante. Plus explicitement, elle est définie dans une relation (appelée ici R2) et fait référence à la clé primaire d'une autre relation (appelée R1). Les valeurs de la clé étrangère dans R2 doivent correspondre aux valeurs existantes de la clé primaire dans R1, ou être nulles si la relation le permet [3].

Tableau 3. Clé étrangère

Clé primaire

Matricule	Prénom	Nom	email
201800177	Pape M	SOW	sowpape@gmail.com
201800188	Rassoul	FALL	rf@gmail.com

Table étudiant

ISBN	TitreLivre	filière	**Matricule**
A12890	Sous l'orage	MIO	201800177
A78280	Etranger	MIO	201800188

Clé étrangère

Table locLivre

1.2.1.4 Les contraintes d'intégrités dans les bases de données

Les contraintes d'intégrité sont des règles essentielles pour assurer la cohérence et la validité des données dans une base de données. Elles définissent les valeurs permises pour certaines

données en fonction de leur rôle ou de leur relation avec d'autres données. Voici les principales contraintes [3] :

- **Contrainte d'unicité de clé** : Cette contrainte garantit que deux tuples d'une relation ne peuvent pas avoir la même valeur pour la clé primaire, assurant ainsi une identification unique de chaque enregistrement ;
- **Contrainte référentielle** : Elle impose qu'une clé étrangère dans une relation (R1) corresponde toujours à une valeur existante de la clé primaire d'une autre relation (R2), assurant la cohérence entre les relations. Par exemple, pour chaque emprunt (relation Emprunt), la valeur de l'attribut Cote doit exister comme clé primaire dans la relation Livres ;
- **Contrainte de domaine** : Cette contrainte détermine que les valeurs d'un attribut doivent appartenir à un domaine ou à une plage prédéfinie. Par exemple, un salaire doit être compris entre 5 000 et 30 000, ou une couleur doit appartenir à l'ensemble {Bleu, Blanc, Rouge} ;
- **Valeurs nulles et clés** : Une valeur nulle (NULL) est utilisée lorsqu'une donnée est inconnue au moment de l'insertion. Cependant, les attributs participant à une clé primaire ne peuvent jamais être nulls, car cela compromettrait l'unicité et l'identification des enregistrements ;
- **Contrainte d'entité** : Cette contrainte exige que toute relation possède une clé primaire et que les attributs de cette clé soient non nuls, garantissant une identification claire et cohérente de chaque enregistrement.

Ces règles sont cruciales pour maintenir l'intégrité logique des bases de données et sont généralement vérifiées et appliquées automatiquement par les SGBD.

1.2.2 L'algèbre relationnel

L'algèbre relationnelle, créée par E. F. Codd, est un cadre formel permettant de manipuler des relations dans le modèle relationnel. Comparable à l'arithmétique pour les entiers, elle définit un ensemble d'opérations élémentaires pour transformer et interroger les données [1].

Codd a initialement introduit huit (8) opérations de base, dont certaines sont dérivables des autres. Ces opérations élémentaires permettent de manipuler les données efficacement, en facilitant des tâches comme la projection, la sélection, les jointures ou les regroupements. En tant que fondement du langage SQL utilisé aujourd'hui, l'algèbre relationnelle reste un pilier central dans le domaine des bases de données relationnelles [2].

Ces opérations sont la projection, la sélection, l'union, l'intersection, la différence, le produit cartésien, la jointure, la division cartésienne et les fonctions d'agrégation.

1.2.2.1 La projection

La projection est une opération en algèbre relationnelle qui extrait une nouvelle relation à partir d'une relation initiale en sélectionnant uniquement certains attributs spécifiés. Elle élimine les autres attributs ainsi que les doublons, garantissant ainsi l'unicité des tuples dans la relation résultante. La syntaxe est R2 = ∏ attributs (R1) [3].

1.2.2.2 La sélection

La sélection est une opération qui, à partir d'une relation initiale (RELATION 1), produit une nouvelle relation (RELATION 2) ayant le même schéma mais contenant uniquement les tuples qui satisfont une condition donnée. Cette condition peut inclure des opérateurs de comparaison (>, <, =, ≠, /) et des opérateurs logiques (et/ou). La syntaxe est R2 = σcondition (R1) [7].

1.2.2.3 L'union

L'union est une opération qui combine deux relations ayant le même schéma (RELATION 1 et RELATION 2) pour produire une nouvelle relation (RELATION 3) contenant tous les tuples présents dans l'une ou l'autre des relations, ou dans les deux. Les doublons sont éliminés, garantissant que chaque tuple dans la relation résultante est unique [10]. La syntaxe est RELATION 3 = RELATION 1 ∪ RELATION 2

1.2.2.4 L'intersection

L'intersection est une opération qui, à partir de deux relations ayant le même schéma (RELATION 1 et RELATION 2) produit une nouvelle relation (RELATION 3) contenant uniquement les tuples communs aux deux relations [6]. La syntaxe est RELATION 3 = RELATION 1 ∩ RELATION 2

1.2.2.5 La différence

La différence est une opération, qui, à partir deux relations ayant le même schéma (RELATION 1 et RELATION 2) produit une nouvelle relation (RELATION3) contenant uniquement les tuples présents dans RELATION1 et absent dans RELATION2 [11]. La syntaxe est RELATION 3 = RELATION 1 – RELATION 2

1.2.2.6 Produit cartésien

Le produit cartésien est une opération qui, à partir de deux relation (RELATION 1 et RELATION 2) ayant le même schéma, produit une nouvelle relation (RELATION 3) ayant pour schéma la concaténation des schémas des deux relations opérandes. Les tuples de la

relation résultante sont toutes les combinaisons possibles des tuples des deux relations opérandes [7]. Pour être plus explicite, pour chaque tuple de RELATION 1, on crée une combinaison avec chaque tuple de RELATION 2, ce qui génère toutes les combinaisons possibles des tuples des deux relations d'où la naissance de la RELATION 3**.** La syntaxe est RELATION 3 = RELATION 1 X RELATION 2

1.2.2.7 La jointure

La jointure est une opération fondamentale en algèbre relationnelle, utilisée pour combiner deux relations en une seule, selon un critère spécifique appelé condition de jointure. Elle peut être considérée comme une extension du produit cartésien, où seuls les tuples satisfaisant la condition de comparaison entre les attributs des deux relations sont conservés dans le résultat.
La jointure combine deux relations en comparant des valeurs spécifiques de leurs attributs, elle réduit le nombre de tuples résultants en éliminant ceux qui ne satisfont pas la condition de jointure. Le schéma de la relation résultante est une concaténation des schémas des deux relations opérandes [12]. La syntaxe est RELATION3=RELATION1 ⋈ condition RELATION2

1.2.2.8 La division cartésienne

La division est une opération en algèbre relationnelle qui s'applique à deux relations R1 et R2, où le schéma de R2 est strictement inclus dans celui de R1. L'opération produit une nouvelle relation (R3) contenant toutes les occurrences de R1 qui sont associées à toutes les occurrences de R2 [12]. La syntaxe est R3=R1÷R2

1.2.2.9 Les fonctions d'agrégation

Les fonctions d'agrégation permettent de manipuler et résumer des ensembles de données au sein d'une relation, en regroupant plusieurs enregistrements pour effectuer des calculs spécifiques [7]. Voici les principales fonctions d'agrégation :

- **AVG ()** : Calcule la moyenne d'un ensemble de valeurs ;
- **COUNT ()** : Compte le nombre de lignes concernées par une requête ;
- **MAX ()** : Récupère la plus haute valeur dans un ensemble ;
- **MIN ()** : Récupère la plus petite valeur dans un ensemble ;
- **SUM ()** : Calcule la somme de plusieurs lignes pour une colonne donnée.

Tableau 4. Tableau récapitulatif de l'algèbre relationnelle

Opération	Symbole	Fonction	Résultat attendu
Projection	π	Sélectionne certaines colonnes (attributs) d'une relation.	Relation avec les colonnes choisies, sans doublons.
Sélection	σ	Filtre les lignes selon une condition logique.	Relation avec les lignes qui satisfont la condition.
Union	∪	Combine deux relations ayant le même schéma.	Ensemble de tous les tuples des deux relations (sans doublons).
Intersection	∩	Récupère les tuples communs à deux relations.	Tuples présents dans les deux relations.
Différence	−	Récupère les tuples présents dans une relation mais pas dans l'autre.	Tuples de la première relation absents de la seconde.
Produit cartésien	×	Associe chaque tuple d'une relation avec chaque tuple de l'autre.	Toutes les combinaisons possibles de tuples des deux relations.
Jointure	⋈	Combine deux relations selon une condition sur des attributs communs.	Relation fusionnée avec les tuples qui satisfont la condition.
Division	÷	Récupère les tuples associés à toutes les valeurs d'une autre relation.	Tuples qui ont une relation avec tous les tuples de l'autre.
Fonctions d'agrégation	AVG(), COUNT(), etc.	Effectuent des calculs statistiques sur une colonne (somme, moyenne, max, etc.).	Une ou plusieurs valeurs synthétiques (ex. total, moyenne, etc.).

1.3 Mode de Fonctionnement (ACID)

Les systèmes de gestion de bases de données relationnelles assurent la fiabilité des transactions à travers les propriétés ACID que sont : L'atomicité, la cohérence, l'isolation et la durabilité.

1.3.1 Atomicité

Une transaction est indivisible. Elle doit être entièrement réalisée ou totalement annulée. Si une seule opération échoue, toutes les modifications précédentes de cette transaction sont annulées, préservant ainsi l'intégrité de la base de donnée [8]. Par exemple, considérons une transaction qui transfère de l'argent d'un compte A à un compte B. La transaction implique 2 opérations que sont :

- Débiter le compte A de 100.000 FCFA ;
- Créditer le B de 100.000 FCFA.

Si le débit du compte A réussi et que le crédit du compte B échoue, l'atomicité garantit que le débit sera annulé et qu'aucun changement ne sera effectué dans la base de données.

1.3.2 Cohérence

Une transaction doit maintenir la cohérence de la base de données. Les modifications effectuées doivent respecter toutes les contraintes d'intégrité définies. En cas d'échec, le système revient au dernier état cohérent connu [8]. Par exemple**,** supposons qu'il y ait une contrainte intégrée qui stipule qu'un compte ne peut pas avoir un solde négatif. Donc, si on tente de débiter le compte A et qu'après le débit, le compte aurait un solde négatif, la transaction sera directement annulée. Cette action va ainsi maintenir la cohérence de la base de données.

1.3.3 Isolation

Chaque transaction s'exécute comme si elle était seule dans le système. Les modifications en cours dans une transaction ne sont pas visibles pour les autres jusqu'à ce qu'elles soient validées. Cela évite les conflits entre transactions simultanées [8]. Par exemple, si deux transactions tentent de transférer de l'argent simultanément entre ces mêmes comptes (compte A et compte B), l'isolement garantit que l'une des transactions sera exécutée entièrement avant que l'autre ne commence. Cela évite des situations où les deux transactions pourraient lire des données intermédiaires et produire des résultats incorrects.

1.3.4 Durabilité

Une fois qu'une transaction est validée, ses effets sont permanents, même en cas de panne du système. Cette garantie repose sur l'utilisation de journaux [3]qui enregistrent toutes les transactions pour permettre leur récupération [8]. Par exemple, si la transaction de transfert d'argent est validée et que le système tombe en panne juste après, la durabilité garantit que le transfert sera enregistré de manière permanente dans la base de données. À la reprise, le système retrouvera l'état de la base de données tel qu'il était après la transaction.

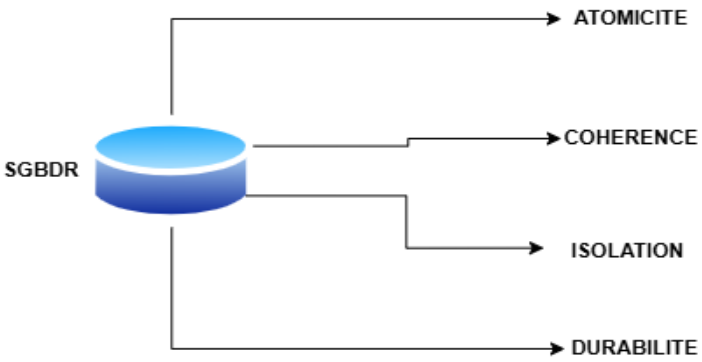

Figure 2. Mode de fonctionnement SGBDR : Règle ACID

[3] **Journal en base de données** : Fichier enregistrant de manière séquentielle toutes les modifications effectuées sur une base de données, utilisé pour garantir l'intégrité et la récupération des données en cas de panne.

1.4 Les Usages des SGBDR

Les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR) constituent aujourd'hui un pilier fondamental des systèmes d'information. Depuis leur apparition dans les années 1970, et particulièrement avec la formalisation du modèle relationnel par Edgar F. Codd, leur utilisation n'a cessé de croître dans les environnements professionnels et institutionnels [6]. Leur succès repose principalement sur leur capacité à structurer les données sous forme de tables interconnectées, à garantir l'intégrité des données et à fournir un accès multi-utilisateur sécurisé.

1.4.1 Utilisation dans les systèmes de gestion traditionnels

Les SGBDR sont massivement déployés dans les applications de gestion courante des entreprises. Ils servent de socle à des systèmes tels que les ERP (Enterprise Resource Planning), les logiciels de comptabilité, de facturation, de paie, de gestion de stock, ou encore de suivi de la relation client (CRM). Dans ces contextes, les SGBDR permettent de modéliser les données de manière cohérente et de garantir une intégrité référentielle entre les différentes entités (clients, commandes, produits, fournisseurs, etc.). Par exemple, une entreprise de distribution peut utiliser un SGBDR pour suivre l'état de ses inventaires en temps réel, en relation avec chaque vente à un client, et générer automatiquement des rapports de performance mensuels [10].

1.4.2 Intégration dans les applications web dynamiques

Avec l'émergence du web dynamique, les SGBDR ont été au cœur de la transformation numérique. Les sites web interactifs et les plateformes de services en ligne tels que les sites de e-commerce (ex : Jumia, Amazon), les portails institutionnels, ou les plateformes de réservation (ex : Air Sénégal, Air France, SDD), s'appuient tous sur des bases de données relationnelles pour gérer les profils utilisateurs, les transactions, les historiques d'achat, les avis, etc. Dans ce contexte, les SGBDR permettent un accès simultané aux données, une gestion des droits utilisateurs, une rapidité dans l'exécution des requêtes, ainsi qu'une grande souplesse d'intégration avec les technologies [10].

1.4.3 Utilisation dans les systèmes décisionnels

Les SGBDR jouent également un rôle essentiel dans les systèmes décisionnels. Intégrés dans des entrepôts de données, ils servent de fondation à l'analyse de données à grande échelle. Grâce à des technologies complémentaires telles que l'analyse multidimensionnelle (OLAP) ou la fouille de données (data mining), les entreprises peuvent extraire des indicateurs de

performance, identifier des tendances, détecter des anomalies ou segmenter leur clientèle de manière fine. Par exemple, une entreprise de télécommunication peut utiliser un entrepôt de données relationnelles pour analyser le comportement de ses abonnés, prévoir les risques de résilience et adapter ses campagnes de fidélisation [11].

1.4.4 Support dans les systèmes critiques et distribués

Les SGBDR sont largement utilisés dans les environnements à haute exigence de disponibilité, de cohérence et de sécurité, comme les banques, les assurances, les systèmes de santé ou les administrations publiques. Ils sont intégrés dans des architectures distribuées, parfois en interaction avec des systèmes NoSQL ou des bases orientées graphiques, selon les besoins spécifiques. Grâce à leurs mécanismes transactionnels (ACID), les SGBDR assurent que chaque opération s'exécute de manière fiable, même en cas de défaillance du système. Dans les systèmes bancaires, par exemple, ils sont utilisés pour enregistrer les opérations de débit/crédit, les mouvements de comptes en temps réel, et détecter les opérations suspectes suivre à l'aide de requêtes complexes [12].

Voici un tableau récapitulatif :

Tableau 5. Tableau récapitulatif des cas d'usage du SGBDR

Contexte d'utilisation	Type d'applications concernées	Exemples concrets
Gestion administrative	Comptabilité, facturation, gestion des ressources humaines, CRM	Logiciels de paie, ERP (ex. SAP, Odoo), gestion client
Web dynamique et e-commerce	Portails web, sites transactionnels, services en ligne	Amazon, Booking.com, portails gouvernementaux
Systèmes décisionnels	Entrepôts de données, reporting, analyse multidimensionnelle (OLAP), data mining	Tableaux de bord, segmentation client, prévision des ventes
Applications critiques	Traitement sécurisé, transactions atomiques, données sensibles	Banques, hôpitaux, compagnies d'assurance
Systèmes distribués	Architectures client-serveur, cloud, interconnexion de bases	Bases interconnectées via API, synchronisation de bases multi sites
Secteur bancaire et financier	Gestion des comptes, transactions, opérations sécurisées	Suivi en temps réel des soldes et opérations
Transport et logistique	Gestion des flux, réservations, optimisation des trajets	Suivi colis, réservation en ligne, billetterie
Santé	Systèmes d'information hospitaliers, Dossier Patient Informatisé (DPI)	Prescriptions électroniques, historique médical
Éducation	Scolarité, emploi du temps, suivi académique	Bases d'étudiants, bulletins, gestion des examens
Commerce et distribution	Gestion des produits, commandes, ventes et stocks	Inventaire, caisse, facturation automatisée
Administration publique	Démarches administratives, bases légales et citoyennes	Recensements, fiscalité, état civil

1.5 Interrogation des SGBDR (le SQL)

Le SQL (Structured Query Language) est le langage standard universel pour interagir avec les bases de données. Il est largement adopté par la quasi-totalité des systèmes de gestion de bases de données (SGBD), des plus simples comme Access aux solutions d'entreprise robustes comme Oracle. Son succès repose principalement sur sa simplicité et sa capacité à permettre aux utilisateurs de formuler des requêtes en se basant sur la logique de la base de données (le

schéma conceptuel), laissant au SGBD le soin d'optimiser et d'exécuter ces requêtes. Le SQL est un langage déclaratif et ensembliste : il opère sur des ensembles de données (les tables ou relations) et produit également des tables en résultat [12]. Le SQL possède 5 catégories d'instructions : LDD, LMD, LCD et LCT

1.5.1 Langage de définition de données

Le Langage de Définition de Données (LDD), également connu sous son acronyme anglais DDL (Data Definition Language), est la partie du SQL spécifiquement conçue pour la gestion structurelle d'une base de données relationnelle. Il permet aux utilisateurs de créer, modifier et supprimer les objets fondamentaux de la base, tels que les tables, les vues et les index. Au-delà de la simple manipulation d'objets, le LDD est crucial pour définir la nature des données stockées, notamment leurs types (numérique, textuel, date, booléen, etc.), et pour imposer des règles de validité et de cohérence par l'ajout de contraintes d'intégrité (par exemple, clés primaires, clés étrangères, ou règles de validation de valeurs). Enfin, il inclut des commandes pour l'administration de la sécurité, permettant d'accorder ou de révoquer des privilèges d'accès aux données pour des utilisateurs spécifiques et, dans certains systèmes, d'activer des mécanismes d'audit pour suivre les actions sur la base. Les instructions du LDD [12] sont :

- CREATE : Cette instruction permet de créer des objets dans la base de données tels que des tables, des vues, des index, des utilisateurs, … ;
- ALTER : Cette instruction permet de modifier la structure d'un objet, elle agit sur les colonnes (ajout, modification ou suppression) ;
- DROP : Elle permet de supprimer des objets de la base de données tels que les tables, vues ou index. Cette action est irréversible.
- RENAME : Elle permet de renommer un objet existant dans la base de données, à l'exemple d'une table ou d'une colonne.
- TRUNCATE : Elle permet de vider entièrement une table.

1.5.2 Le Langage de Manipulation de Données

Le langage de manipulation de données (LMD), ou Data Manipulation Language (DML) en anglais, constitue la suite de commandes SQL dédiée à l'interaction directe avec les informations stockées au sein d'une base de données.

Ce sous-langage est essentiel pour réaliser les opérations suivantes [13] :

- **INSERT** : Cette instruction permet l'ajout de nouvelles données ;
- **UPDATE** : Elle permet la modification des données existantes ;

- **DELETE** : Elle permet la suppression d'enregistrements ;
- **SELECT** : L'instruction permet d'interroger la base de données pour visualiser et extraire son contenu, offrant des capacités de filtrage, de tri et de jointure ;
- **EXPLAIN** et **PLAN :** aident à comprendre et optimiser la performance des requêtes ;
- **LOCK TABLE :** Elle est utilisée pour gérer l'accès concurrentiel aux données.

NB : Il est impératif de noter que toutes les modifications effectuées via le LMD ne deviennent permanentes qu'après leur validation explicite dans le cadre d'une transaction, garantissant ainsi l'intégrité et la cohérence de la base de données [13].

1.5.3 Le langage de contrôle de données

Le Langage de contrôle de données (LCD), ou Data Control Language (DCL) en anglais, est un sous-ensemble essentiel de SQL qui se concentre sur la gestion des droits et des permissions d'accès aux objets et aux données au sein d'une base de données. Ses commandes principales sont **GRANT** et **REVOKE**, elles permettent respectivement d'accorder et de retirer des privilèges spécifiques à des utilisateurs ou des rôles, assurant ainsi la sécurité et l'intégrité du système en définissant qui peut accéder à quelles informations et quelles opérations il est autorisé à effectuer [13].

1.5.4 Le langage de contrôle de transactions

Le Langage de Contrôle de Transaction (LCT) ou Transactionnal Control Language (TCL) en anglais, est un composant fondamental de SQL qui régule les modifications effectuées par le Langage de Manipulation de Données (DML). Son rôle principal est de gérer les propriétés des transactions, en permettant de valider de manière permanente les changements apportés à la base de données, ou à l'inverse, d'annuler toutes les modifications effectuées depuis le début de la transaction ou le dernier point de sauvegarde, assurant ainsi la cohérence et l'intégrité des données face aux erreurs ou aux pannes.

Les instructions du TCL sont : COMMIT, SAVEPOINT, ROLLBACK, SET TRANSACTION [14].

Tableau 6. Tableau récapitulatif de ces catégories d'instructions

Type de langage SQL	Nom anglais (acronyme)	Fonction principale	Commandes principales	Fonctionnalités complémentaires
Langage de Définition de Données	Data Definition Language (DDL)	Créer, modifier ou supprimer la structure des objets de la base	CREATE, ALTER DROP, RENAME, TRUNCATE	Définition des types, contraintes (clés primaires/étrangères), sécurité (AUDIT, NOAUDIT), analyse (ANALYZE)
Langage de Manipulation de Données	Data Manipulation Language (DML)	Manipuler les données existantes dans la base	SELECT, INSERT, UPDATE, DELETE	EXPLAIN, PLAN (analyse des requêtes), LOCK TABLE (concurrence), nécessite COMMIT pour rendre les changements permanents
Langage de Contrôle de Transaction	Transactionnal Control Language (TCL)	Gérer les transactions et garantir la cohérence des modifications	COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION	Gestion de l'annulation ou de la validation des modifications
Langage de Contrôle de données	Data Control Language (DCL)	Gérer les permissions et la sécurité d'accès aux données	GRANT, REVOKE	Attribution ou retrait de droits aux utilisateurs ou rôles

1.6 SGBDR les plus répandus

Il existe plusieurs types de système de gestions de base de données. On peut citer MySQL, Oracle database, SQLite, Microsoft Server. On peut en citer aussi d'autres systèmes puissants comme PostgreSQL, IBM db2, etc. Ainsi, pour la présentation on va prendre 4 SGBD parmi les plus répandus :

- **MySQL** : MySQL est un système de gestion de bases de données SQL open source populaire, développé, distribué et pris en charge par Oracle Corporation. MySQL gère un ensemble structuré de données. Une base de données MySQL vous permet d'ajouter,

d'accéder et de traiter les données qu'elle contient. MySQL stocke les données dans des tables distinctes. Les structures de la base de données sont organisées en fichiers physiques optimisés pour la vitesse. Le modèle logique, avec des objets tels que des bases de données, des tables, des vues, des lignes et des colonnes, offre un environnement de programmation flexible. La partie SQL de « MySQL » signifie « Structured Query Language », le langage standardisé le plus couramment utilisé pour accéder aux bases de données. Le logiciel MySQL utilise la licence GPL (GNU General Public) et est un logiciel open source [15] ;

- **Oracle database** : Fourni par Oracle Corporation, Oracle Database est un système de gestion de base de données relationnelle , il a été développé par Larry Ellison, accompagné entre autres, de Bob Miner et Ed Oates. En 2025, il a été classé le premier des 424 meilleurs SGBDR, devant MySQL et Microsoft SQL Server d'après une source[4] ;
- **SQLite** : SQLite est une bibliothèque embarquée qui implémente un moteur de base de données relationnelle SQL, léger, autonome, sans configuration ni serveur. Contrairement aux bases classiques fonctionnant en mode client-serveur, SQLite s'exécute directement au sein de l'application et lit ou écrit les données dans un simple fichier sur disque. Son code est en domaine public, ce qui permet une utilisation libre à des fins personnelles ou commerciales. Grâce à sa petite taille (souvent inférieure à 500 Ko) et sa simplicité, SQLite est largement utilisé dans les systèmes embarqués comme les smartphones, les tablettes, les lecteurs MP3 ou les applications web mobiles. Il prend en charge les transactions ACID, fonctionne avec de nombreux langages de programmation (C, Java, Python, C#, etc.) et est compatible avec différents systèmes d'exploitation, notamment Android, iOS ou Symbian. Réputé pour sa fiabilité, SQLite subit de nombreux tests avant chaque version et dispose d'un suivi rigoureux des bugs. Sa légèreté, sa portabilité et sa robustesse en font une solution idéale pour les applications nécessitant un stockage local efficace et peu gourmand en ressources [16] ;
- **Microsoft SQL Server** : Microsoft SQL Server est le système de gestion de base de données relationnelle (SGBDR) développé par Microsoft depuis 1989, dont la version la plus récente est SQL Server 2022, sortie en novembre 2022. Il s'agit d'un SGBDR commercial, fonctionnant sous Windows et Linux, avec un fort support du langage SQL

[4] https://db-engines.com/en/ranking.

et des fonctionnalités avancées telles que les schémas de données, les index secondaires, le support XML, les déclencheurs, la cohérence immédiate, ainsi que le partitionnement horizontal des tables. SQL Server se classe régulièrement parmi les meilleurs SGBD au monde (actuellement 3e selon DB-Engines), grâce à ses performances, sa polyvalence et son intégration dans l'écosystème Microsoft.[5]

1.7 Points forts

Les SGBDR présentent de nombreux avantages qui justifient leur adoption généralisée :

- **Fiabilité et cohérence des données :** Les SGBDR sont intrinsèquement conçus autour du modèle ACID, un ensemble de propriétés fondamentales garantissant la fiabilité des transactions. L'Atomicité assure que chaque transaction, qu'il s'agisse d'une simple mise à jour ou d'une série complexe d'opérations, est traitée comme une unité logique indivisible. Cela signifie que toutes les opérations au sein d'une transaction doivent être menées à terme avec succès, ou bien aucune d'entre elles ne l'est, évitant ainsi des états de données inconsistants ou partiels en cas d'échec. La Cohérence maintient l'intégrité de la base de données en garantissant que toute transaction valide fait passer le système d'un état cohérent à un autre, en respectant toutes les règles métier et contraintes d'intégrité prédéfinies. L'Isolation est cruciale dans les environnements multi-utilisateurs, car elle garantit que l'exécution concurrente de multiples transactions ne produit pas d'interférences indésirables, chaque transaction étant isolée des autres comme si elle était la seule à s'exécuter. Enfin, la Durabilité assure que, une fois qu'une transaction est validée et enregistrée, ses modifications sont permanentes et survivent à toute défaillance du système, qu'il s'agisse d'une panne de courant ou d'un redémarrage inattendu. Ces propriétés ACID confèrent aux SGBDR une robustesse et une fiabilité inégalées pour les applications nécessitant une intégrité des données sans compromis [16] ;
- **Structure et clarté du modèle relationnel** : Le modèle relationnel, sur lequel reposent les SGBDR, propose une organisation des données hautement structurée sous forme de tables, composées de lignes et de colonnes, avec des relations bien définies entre elles. Cette approche intrinsèquement logique et ordonnée facilite grandement la compréhension de l'architecture des données et simplifie leur gestion quotidienne. En

[5] https://db-engines.com/en/system/Microsoft+SQL+Server

imposant un schéma strict et prédéfini, les SGBDR garantissent une cohérence structurelle et une uniformité des données, ce qui est particulièrement avantageux pour les applications où la structure des informations est stable et clairement définie. Cette rigidité de schéma, bien que parfois perçue comme une contrainte, est en réalité une force pour maintenir une intégrité des données élevée et pour assurer une vue cohérente des informations immédiatement après leur manipulation [18] ;

- **Langage de requête standardisé (SQL) :** Le Structured Query Language (SQL) est l'un des piliers des SGBDR et représente un langage standardisé et bien documenté pour l'interaction avec les bases de données relationnelles. Bien qu'il puisse exister de légères variations dans les implémentations de SQL entre différents SGBDR (comme MySQL, Microsoft SQL Server ou PostgreSQL), le noyau du langage reste cohérent. Cette standardisation est un atout majeur car elle réduit considérablement la courbe d'apprentissage pour les développeurs, leur permettant de s'adapter rapidement à diverses plateformes SGBDR une fois qu'ils maîtrisent les principes de base de SQL. La puissance de SQL réside dans sa capacité à exécuter des requêtes complexes, des jointures élaborées et des agrégations de données sophistiquées, offrant ainsi une grande flexibilité pour l'extraction et la manipulation des informations [17] ;
- **Mécanismes de sécurité robustes :** La sécurité est un aspect critique des SGBDR, qui intègrent des mécanismes de protection avancés et font partie intégrante de leur conception. Ces fonctionnalités incluent des contrôles d'authentification pour vérifier l'identité des utilisateurs, des systèmes d'autorisation pour définir précisément les privilèges d'accès aux tables et aux opérations, des capacités de chiffrement pour protéger les données sensibles au repos et en transit, des contrôles d'intégrité pour garantir la non-altération des données, et des fonctions d'audit pour suivre toutes les activités et débusquer les accès non autorisés ou les comportements suspects. Ces mécanismes de sécurité intégrés offrent une protection complète des données, faisant des SGBDR un choix privilégié pour les applications nécessitant un niveau élevé de confidentialité et de conformité [18] ;
- **Maturité, Robustesse et Écosystème développé :** Les SGBDR sont une technologie mature, existant depuis 1979, et ont dominé le paysage des bases de données pendant plus de trois décennies. Cette longue histoire a permis un développement continu, une optimisation approfondie et l'établissement d'un écosystème très riche. La maturité des

SGBDR se traduit par leur robustesse avérée, leur stabilité opérationnelle et la disponibilité d'une multitude d'outils, de documentations, de communautés d'utilisateurs et d'experts. Pour de nombreuses applications, notamment celles à volume de données limité mais nécessitant une cohérence élevée, les SGBDR sont faciles à implémenter et offrent une fiabilité qui a fait ses preuves [17].

1.8 Limites

Malgré leur efficacité dans de nombreux domaines, les SGBD relationnels présentent des limites notables face aux exigences modernes de gestion des données :

- **Limites de scalabilité et performance (scalabilité verticale dominante) :** L'une des principales limitations des SGBDR réside dans leur modèle de scalabilité prédominant : la scalabilité verticale. Cela implique d'augmenter la puissance d'un seul serveur en lui ajoutant des ressources matérielles (RAM, CPU, stockage). Cependant, cette approche est coûteuse, exige un effort significatif de la part des administrateurs pour les mises à niveau, et est intrinsèquement limitée par les capacités physiques du matériel disponible. Cette limitation de la scalabilité rend les SGBDR moins adaptés aux exigences des applications modernes du web 2.0, 3.0 et du Big Data, qui nécessitent une capacité à gérer des volumes de données massifs et des charges d'utilisateurs extrêmement élevées, souvent mieux prises en charge par la scalabilité horizontale. De plus, les SGBDR sont généralement plus lents dans le traitement des informations par rapport aux bases NoSQL, qui bénéficient souvent d'une récupération des données à partir de la mémoire volatile, plus rapide que la mémoire non-volatile utilisée par les SGBDR [19] ;
- **Coût élevé et investissement important :** Les SGBDR, en particulier les solutions propriétaires et avancées comme Oracle Database ou Microsoft SQL Server, représentent un investissement financier considérable pour les organisations. Le coût ne se limite pas seulement aux licences logicielles, mais englobe également les dépenses associées à l'acquisition de matériel performant pour la scalabilité verticale, ainsi que les coûts de maintenance, de support technique et d'expertise spécialisée. Cela rend les SGBDR une approche onéreuse pour le stockage et la gestion des données, particulièrement pour les petites et moyennes entreprises ou les projets à budget limité, contrastant avec l'approche souvent open source et moins coûteuse des solutions NoSQL [18] ;

- **Difficulté à gérer les gros volumes et la variété des données modernes :** L'avènement du Big Data et la prolifération des applications web 2.0 et 3.0 ont considérablement augmenté le volume et la variété des données que les bases de données doivent gérer. Les SGBDR, optimisés pour les données structurées et relationnelles, peinent à s'adapter efficacement à ces nouveaux paradigmes. Ils rencontrent des difficultés à traiter et stocker des volumes massifs de données provenant de diverses sources, ainsi qu'à gérer des données non structurées (comme des fichiers audio, vidéo, ou texte libre) ou semi-structurées (comme le JSON ou XML) sans nécessiter des conversions complexes qui peuvent entraîner une perte d'informations contextuelles ou de nuances [17] ;
- **Vulnérabilité aux points de défaillance uniques et disponibilité limitée :** Par nature, de nombreux SGBDR souffrent d'un point de défaillance unique. Même les serveurs les plus puissants peuvent subir des pannes, ce qui, sans une architecture de haute disponibilité coûteuse et complexe, peut entraîner des interruptions de service significatives. La stratégie de scalabilité verticale accentue ce problème, car toutes les ressources critiques sont centralisées sur une seule machine. Dans le contexte des applications internet modernes, où les utilisateurs dépendent d'une disponibilité constante (ex : réseaux sociaux, e-commerce), cette limitation est un désavantage majeur, car elle ne permet pas un accès continu aux données en cas de défaillance matérielle, contrairement aux systèmes distribués [19] ;
- **Rigidité du schéma et manque de flexibilité :** La nécessité de définir un schéma strict et fixe avant toute insertion de données est une caractéristique fondamentale des SGBDR. Bien que cela garantisse l'intégrité, cela rend également le système rigide. Les applications sont contraintes de se conformer aux exigences de la base de données plutôt que l'inverse. Toute modification du schéma (ajout de colonnes, changement de type de données) sur une base de données en production peut être une opération complexe, coûteuse en temps et potentiellement disruptive, car elle nécessite des migrations de données et des ajustements applicatifs importants. Cette rigidité est un inconvénient majeur pour les applications qui évoluent rapidement et dont le modèle de données n'est pas figé [20] ;
- **Complexité dans la modélisation de certaines données :** La forte orientation des SGBDR vers les données structurées peut paradoxalement introduire de la complexité dans des situations où les informations ne se prêtent pas naturellement à une

représentation tabulaire. La conversion de données intrinsèquement non structurées ou semi-structurées en un format relationnel peut être ardue et entraîner une perte de richesse sémantique ou de nuances. De plus, pour des traitements simples comme ceux que l'on trouve dans les médias sociaux (simples opérations de lecture ou d'écriture), les SGBDR peuvent être excessivement complexes, car leurs mécanismes sophistiqués de gestion de requêtes et de transactions peuvent ne pas être nécessaires [20].

Tableau 7. Points forts et limites du SGBDR

Catégorie	Aspect	Détails
Points forts	Fiabilité et cohérence (ACID)	Garantit des transactions sûres : Atomicité, Cohérence, Isolation, Durabilité.
	Structure claire du modèle relationnel	Organisation en tables avec schéma strict, facilitant compréhension et intégrité des données.
	Langage SQL standardisé	Langage universel, puissant et bien documenté, avec compatibilité multiplateformes.
	Mécanismes de sécurité robustes	Authentification, autorisation, chiffrement, audit, contrôle d'accès strict.
	Maturité et écosystème développé	Technologie éprouvée, riche en outils, documentation et support communautaire ou commercial.
Limites	Scalabilité verticale limitée	Moins adaptée aux charges massives et au Big Data ; coûteuse et limitée physiquement.
	Coût élevé	Licences, matériel, maintenance, expertise technique représentent un investissement important.
	Faible capacité à gérer des données non structurées	Inadapté aux formats modernes (audio, vidéo, JSON, XML) sans transformations complexes.
	Point de défaillance unique	Faible tolérance aux pannes sans architecture de haute disponibilité (HA) complexe.
	Rigidité du schéma	Modifications de structure lourdes en production, peu flexible pour les environnements évolutifs.
	Complexité de modélisation	Difficile à adapter à des données non tabulaires ou peu structurées, risque de perte sémantique.

En conclusion, ce chapitre nous a permis d'appréhender de manière approfondie le fonctionnement des **Systèmes de Gestion de Bases de Données Relationnelles (SGBDR)**. Fondés sur le modèle relationnel proposé par E.F. Codd, les SGBDR se caractérisent par leur structuration rigoureuse des données en tables interconnectées et l'utilisation du langage SQL pour les opérations de définition, manipulation et contrôle. Leur force réside notamment dans leurs propriétés ACID qui garantissent la fiabilité, l'intégrité et la durabilité des transactions, ainsi que dans leur capacité à gérer des données fortement structurées avec une cohérence stricte. Ils s'imposent ainsi comme des solutions privilégiées pour les applications critiques nécessitant une forte intégrité des données, telles que la finance, la santé ou l'administration. Cependant, malgré leurs nombreux atouts, les SGBDR présentent certaines limites, notamment en matière de scalabilité horizontale et de gestion de données massives et non structurées, ce qui ouvre la voie à l'émergence d'alternatives comme les bases NoSQL, que nous explorerons dans le chapitre suivant.

Chapitre 2 : Généralité sur les NoSQL

2.1. Présentation

Le terme NoSQL désigne des systèmes de gestion de bases de données qui abandonnent l'architecture relationnelle et transactionnelle traditionnelle, afin de mieux gérer les volumes massifs et la diversité croissante des données [8].

Face à l'évolution rapide des technologies et à l'explosion des volumes de données dans la dernière décennie, les SGBDR classiques ont montré leurs limites, notamment en matière de scalabilité et de montée en charge. Ces faiblesses ont poussé les grands acteurs du web, tels que Google, Facebook ou Amazon, à rechercher des alternatives capables de répondre à ces défis [20]. Les systèmes NoSQL ont ainsi émergé comme une solution innovante, adaptée aux besoins modernes des bases de données :

- Gestion de grands volumes de données (Big Data) ;
- Scalabilité horizontale [6]pour s'adapter à des environnements massivement distribués ;
- Souplesse dans la gestion des données hétérogènes ou non structurées.

2.2 Les principales caractéristiques du NoSQL

Le NoSQL présente plusieurs caractéristiques qui font sont essences :

- **Absence de schéma fixe** : Contrairement aux bases relationnelles qui nécessitent un schéma rigide (tables, colonnes définies à l'avance), les bases NoSQL permettent de stocker des données sans structure fixe ou avec des schémas qui évoluent dynamiquement [21] ;
- **Gestion de données complexes ou imbriquées** : Les bases NoSQL prennent en charge des structures complexes (par exemple, des objets imbriqués) sans nécessiter de jointures comme dans les bases relationnelles [21] ;
- **Partitionnement horizontale** : Les bases NoSQL répartissent les données entre plusieurs nœuds pour améliorer les performances et la gestion de grandes quantités de données. Cela se fait généralement par partitionnement horizontal (sharding) [22] [23] ;
- **Réplication des données** : Pour assurer la disponibilité et la tolérance aux pannes, les bases NoSQL répliquent les données sur plusieurs serveurs. Si un nœud tombe en panne, les données restent accessibles via les copies sur d'autres nœuds [23] ;
- **Les propriétés BASE** : Les propriétés BASE (Basically Available, Soft-state, Eventual Consistency) sont conçues comme une alternative aux propriétés ACID des bases

[6] Augmentation de la puissance en ajoutant d'autres serveurs.

relationnelles. Contrairement à ACID, qui garantit une forte cohérence et des transactions atomiques, BASE est plus souple et privilégie la disponibilité et la résilience, ce qui est en adéquation avec le théorème CAP et les objectifs des bases NoSQL. Les caractéristiques de BASE sont fondées sur les limites des SGBDR relationnels [24] :

- **Basically Available (Disponibilité basique)** : Le système reste disponible même en cas de panne ou de désynchronisation d'un ou plusieurs nœuds. Dans un cluster [7]NoSQL, si un nœud devient temporairement inaccessible, les autres nœuds prennent le relais pour garantir que les données restent accessibles, même si elles ne sont pas complètement synchronisées ;
- **Soft-state (Cohérence légère)** : L'état des données dans le système n'est pas strictement cohérent à tout moment. Les bases NoSQL tolèrent que l'état du système change au fil du temps en raison de mises à jour en cours ou de délais de synchronisation entre nœuds. Cela reflète une approche plus flexible que la cohérence stricte d'ACID. Plus clairement, lorsqu'une mise à jour est effectuée sur un nœud, les autres nœuds du cluster peuvent temporairement avoir une version différente des données, jusqu'à ce qu'une synchronisation ait lieu ;
- **Eventual Consistency (Cohérence à terme)** : Le système garantit que, même en présence de désynchronisations temporaires, il atteindra un état cohérent une fois que toutes les mises à jour auront été propagées. Tant qu'aucune nouvelle modification n'est effectuée, les données se stabilisent dans leur version correcte. Dans un système NoSQL, si un utilisateur met à jour ses informations de profil, certains serveurs peuvent temporairement afficher l'ancienne version, mais après un certain temps (une fois la réplication terminée), tous les serveurs reflètent les données mises à jour.

2.3 Mode Fonctionnement (CAP)

Quand on parle des bases de données distribuées (NoSQL), on fait référence en premier lieu au théorème CAP. Il correspond au terme ACID en base de données relationnelles. Ce théorème est mis en place par Eric Brewer, il forme les trois propriétés fondamentales du NoSQL que sont :

[7] **Cluster** : un groupe de machine (serveurs ou ordinateurs) interconnectées qui fonctionnent ensemble pour offrir plus de puissance, de disponibilité ou de fiabilité.

- **Consistency (cohérence)** : La cohérence, dans le contexte du Théorème CAP, signifie que tous les nœuds du système ont une vue identique et synchronisée des données à tout instant**.** Cela implique que toute opération de lecture effectuée sur n'importe quel nœud du système doit renvoyer la donnée la plus récente écrite [25]. Si une donnée est mise à jour sur un nœud, cette modification doit être propagée et rendue visible instantanément sur tous les autres nœuds avant qu'une nouvelle lecture puisse être effectuée. Atteindre une cohérence forte nécessite souvent des mécanismes de coordination complexes ;
- **Avaibility (disponibilité)** : La disponibilité fait référence à la capacité d'un système distribué à rester opérationnel et à répondre à toutes les requêtes (lectures ou écritures) dans un délai raisonnable, sans erreur, même en cas de défaillance de certains de ses nœuds [25]. Autrement dit, si un client envoie une requête, le système doit toujours fournir une réponse, qu'elle indique le succès de l'opération ou un échec. Pour assurer une haute disponibilité, les systèmes répliquent généralement les données sur plusieurs serveurs. Si un nœud tombe en panne, d'autres nœuds peuvent prendre le relais pour servir les requêtes. L'objectif est de minimiser le temps d'arrêt et de garantir un accès continu aux services ;
- **Partition Tolerence (Tolérance aux pannes)** : La tolérance aux partitions est la capacité d'un système distribué à continuer de fonctionner correctement malgré des pannes de communication ou des ruptures de réseau entre ses nœuds. Ces ruptures, appelées "partitions", divisent le système en plusieurs sous-réseaux qui ne peuvent plus communiquer entre eux [26]. Le Théorème CAP suppose que de telles partitions sont inévitables dans un environnement distribué réel (par exemple, pannes de routeurs, coupures de câbles, problèmes de connectivité). Un système tolérant aux partitions doit donc être conçu pour que chaque sous-réseau puisse continuer à fonctionner de manière autonome et traiter les requêtes, même s'il ne peut pas communiquer avec les autres parties du système [21]. C'est cette propriété qui force le compromis : si une partition survient, le système doit choisir entre maintenir la cohérence en refusant des requêtes pour éviter des données divergentes ou la disponibilité en répondant aux requêtes même si cela peut entraîner des incohérences temporaires entre les partitions.

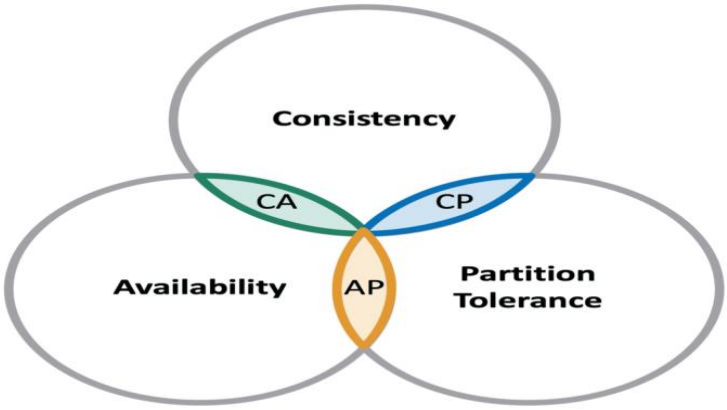

Figure 3. Théorème CAP[8]

De même, le théorème CAP stipule qu'un système distribué ne peut satisfaire simultanément les trois propriétés de Cohérence, Disponibilité et Tolérance au Partitionnement. En conséquence, les concepteurs de systèmes sont contraints d'opérer un choix parmi trois configurations principales : un système **CP** privilégie la cohérence et la tolérance au partitionnement, ce qui implique qu'il pourrait sacrifier la disponibilité en cas de partition du réseau afin de garantir l'exactitude des données. À l'opposé, un système **AP** favorise la disponibilité et la tolérance au partitionnement, ce qui signifie qu'il restera opérationnel même en cas de rupture de communication, au détriment potentiel de la cohérence des données qui pourraient être temporairement divergentes. Enfin, un système **CA** met l'accent sur la cohérence et la disponibilité, mais il est incapable de gérer correctement les scénarios de partitionnement du réseau, un compromis qui le rend rarement utilisé dans les architectures distribuées modernes où l'occurrence de pannes réseau est considérée comme inévitable. Les bases NoSQL choisissent souvent le modèle AP (Disponible + Résistant au partitionnement) au lieu du modèle CP. Cela signifie que même si toutes les copies des données ne sont pas synchronisées (cohérence éventuelle), le système répond toujours rapidement aux requêtes [25].

2.4 Usages des NoSQL

L'émergence des bases de données NoSQL n'est pas le fruit du hasard, mais une réponse directe aux limites des Systèmes de Gestion de Bases de Données Relationnelles (SGBDR) face aux défis posés par les applications modernes. Il existe plusieurs cas d'usage clés pour lesquels les bases NoSQL se révèlent être des solutions plus adaptées :

[8] https://mhd-experts.com/2024/10/13/le-theoreme-cap-comprendre-les-limites-des-bases-de-donnees-avec-l-ospf-comme-exemple/

- **Gestion de Volumes de Données Massifs et à Forte Croissance** : Les applications contemporaines, notamment celles du Web 2.0 et du Web 3.0, génèrent des quantités de données sans précédent. Les SGBDR traditionnels peinent à gérer ces volumes qui se mesurent souvent en téraoctets, voire en pétaoctets. On peut citer des exemples concrets comme la quantité de données traités quotidiennement par Google ou twitter, etc. Cette capacité à ingérer et traiter des volumes de données en croissance exponentielle est l'une des raisons fondamentales de l'adoption de NoSQL, car ces systèmes sont conçus pour opérer efficacement à de telles échelles [27] [28] ;
- **Exigence de Très Haut Débit** : Au-delà du simple volume, certaines applications nécessitent un débit de données (nombre d'opérations par seconde) exceptionnellement élevé. On prend l'exemple de Hypertable (une base orientée colonne) qui permet le stockage d'un milliard de cellules de données par jour pour un moteur de recherche local. Cette capacité à gérer un flux continu et rapide d'écritures et de lectures est un facteur décisif pour les plateformes où chaque milliseconde compte pour l'expérience utilisateur et l'efficacité opérationnelle [28] ;
- **Système avec données flexible ou sans schéma fixe strict** : Les SGBDR imposent un schéma de données rigide qui doit être défini avant toute insertion, et toute modification ultérieure peut être coûteuse et complexe. Les bases NoSQL répondent à cette contrainte en offrant une flexibilité de schéma (ou absence de schéma). Elles sont mieux adaptées pour stocker des données dont la structure est évolutive ou intrinsèquement non tabulaire, comme les structures orientées objet des langages de programmation ou les documents JSON/XML. Cette approche élimine le besoin de coûteux et parfois "tortueux" mappages objet-relationnel (ORM**)**, simplifiant le développement, notamment pour les applications aux structures de données de faible complexité ou en constante évolution [28] ;
- **Scénario où la cohérence immédiate n'est pas cruciale** : Pour des applications comme les mises à jour de statut Facebook ou les tweets, où l'intégrité des données peut être largement optionnelle et où une incohérence temporaire est acceptable (elle sera éventuellement résolue), les bases NoSQL basées sur le modèle CAP (qui privilégie la disponibilité et la tolérance aux partitions sur la cohérence forte immédiate) sont plus adaptées. Cela permet une flexibilité et une scalabilité dramatiques [29] [30] ;

- **Applications nécessitant une scalabilité dès le départ** : Pour les services web de très grande échelle où la capacité à évoluer sans limites est plus importante que la richesse des fonctionnalités de base de données (que l'application peut ensuite construire), les bases NoSQL, conçues pour la distribution et la haute disponibilité, sont privilégiées. Elles permettent une croissance plus automatisée et une disponibilité élevée [39].

2.5 Types des NoSQL

Ils existent 4 types de NoSQL que sont : NoSQL orientée colonne, NoSQL orientée document, NoSQL de type clé-valeur et NoSQL de type graphe.

2.5.1 NoSQL orientée colonne

Dans une base NoSQL orientée colonne, les données sont organisées par colonnes plutôt que par lignes, offrant une plus grande flexibilité pour les schémas dynamiques et évitant le stockage de valeurs NULL. Une colonne peut contenir plusieurs colonnes imbriquées, formant ainsi une super-colonne [25]. Dans une base de données orientée colonnes, toutes les valeurs d'une colonne sont stockées ensemble [3]. Par exemple, tous les salaires seraient regroupés dans une seule structure, ce qui facilite l'accès rapide à ces données lors des requêtes. Les bases de données orientées colonnes sont généralement plus rapides que les bases de données orientées lignes pour les opérations de lecture, en particulier lors de l'exécution de requêtes analytiques. Cela est dû au fait que seules les colonnes nécessaires sont lues, ce qui réduit le volume de données à traiter. Les requêtes qui nécessitent des agrégations ou des analyses sur des colonnes spécifiques peuvent être exécutées plus efficacement, car les données sont déjà organisées de manière à faciliter ces opérations [32]. On suppose des données qui se présentent comme suit :

Tableau 8. Exemple de données stockées [32]

EmpID	Salaire	Poste
100	10000	Vigile
200	20000	Assistant Manager
300	30000	Manager
400	40000	Chef de service

Dans une base de données NoSQL orientée document, les données se présenteront comme suit :

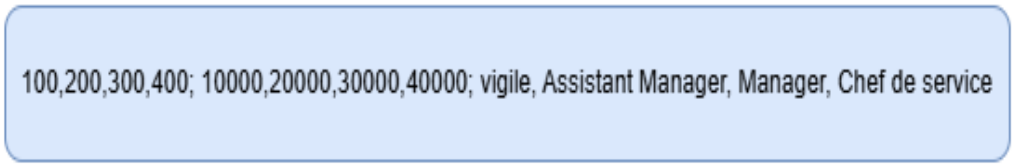

Figure 4. Aperçu des données dans une base orientée colonne [42].

Parmi les bases de données orientée colonne on peut citer Cassandra, HBase, etc.

2.5.2 Base de données orientée document

Les bases de données de type document sont des systèmes NoSQL qui organisent les données sous forme de documents. Ces bases de données peuvent contenir des documents non structurés (comme du texte) ou semi-structurés (comme des fichiers XML), qui ont souvent une structure hiérarchique. Chaque document est constitué d'un ensemble de paires clé-valeur, semblables à celles que l'on trouve dans les bases de données de type Key-Value. En effet, chaque document utilise des pointeurs pour référencer ses champs, en s'appuyant sur une méthode de hachage. Ces bases de données sont dépourvues de schéma fixe, ce qui leur confère une flexibilité dans leur structure. Les bases de données orientées documents accèdent à leurs valeurs à l'aide d'une clé unique qui leur est propre. Cela se présente sous la forme d'un tableau de bases de données, organisé en compartiments [32].

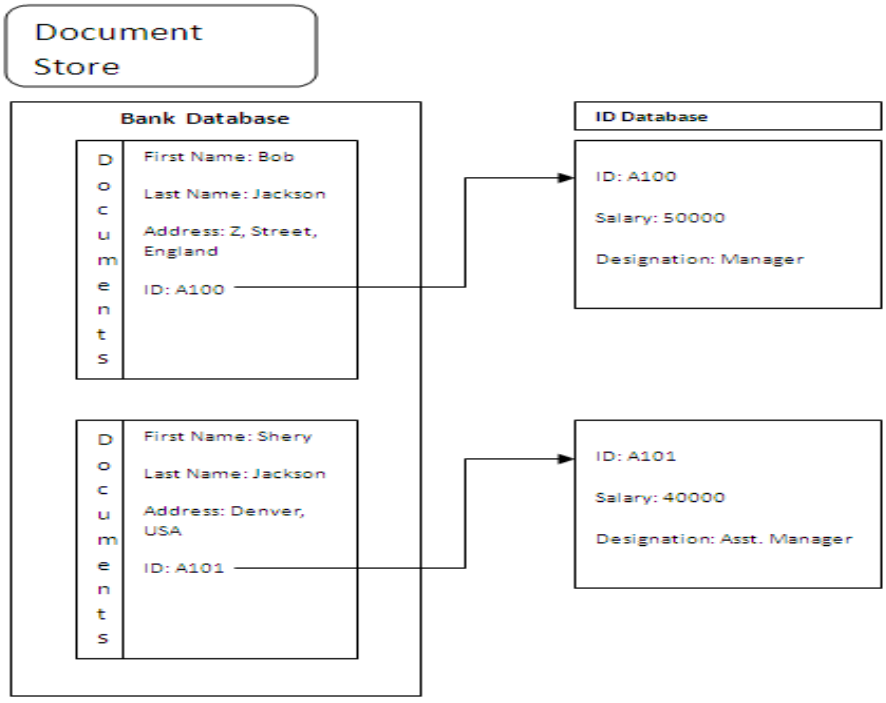

Figure 5. Aperçu des données dans une base orientée document [32]

La figure 4 illustre un exemple de bases de données de type Document, en se concentrant sur une base de données bancaire qui utilise un identifiant (ID) comme clé unique. Cette base de

données inclut également une base de données d'ID, qui contient toutes les informations relatives à chaque personne associée à un ID spécifique. Dans cette configuration, chaque ID dans la base de données bancaire fait référence à la base de données d'ID à l'aide de pointeurs. Dans cette base de données bancaire, nous avons deux documents : (1) un individu nommé Bob Jackson, résidant à Z Street, Angleterre, avec l'ID A100 ; (2) une personne nommée Sherry Hilton, vivant à Denver, États-Unis, avec l'ID A101. Les ID de ces documents servent de clés pour établir des relations. Ces clés doivent être uniques et ne peuvent pas être nulles, ce qui signifie qu'il n'est pas possible d'avoir des clés en double. L'ID a été choisi comme clé, car les noms et adresses ne sont pas toujours uniques, tandis que chaque ID est distinctif pour chaque individu. De plus, il existe une autre base de données, appelée base de données d'ID, qui contient des informations sur le salaire et le poste des personnes. Cette base de données d'ID inclut les enregistrements pour les ID A100 et A101. Les documents de la base de données bancaire contiennent les ID et font référence aux données spécifiques associées à ces identifiants. Ainsi, dans une base de données de type Document Store, les données sont organisées en collections, étiquettes, métadonnées, etc. La récupération des données se fait par le biais de paires clé-valeur. Cela est clairement illustré par l'exemple où la clé A100 est liée à ses attributs correspondants, tels que le salaire et le poste occupé [32].

Parmi les bases de données orientée colonne on peut citer MongoDB, CouchDB, etc.

2.5.3 Base de données de type clé-valeur

Les bases de données clé-valeur comme leur nom l'indique, sont basées sur une combinaison de deux éléments : une clé et une valeur. Elles présentent un système simples et flexibles, associent chaque valeur à une clé unique. Elles se distinguent par leur scalabilité horizontale, permettant une répartition facile des données entre serveurs, et offrent d'excellentes performances en lecture/écriture. Cependant, leur simplicité limite les requêtes aux clés, empêchant des recherches complexes. Ces bases conviennent à des usages spécifiques comme le stockage de cache ou la gestion de sessions, où l'intégrité des données est moins critique [4]. Les bases de données Key-Value fonctionnent de manière similaire aux tables de hachage ou aux tables de recherche. Dans ce type de base de données, la seule méthode de requête consiste à utiliser la clé unique. Les clés peuvent être nommées de manière arbitraire et sont généralement organisées par ordre alphabétique. Pour garantir une disponibilité élevée, les objets de données peuvent être répliqués [13].

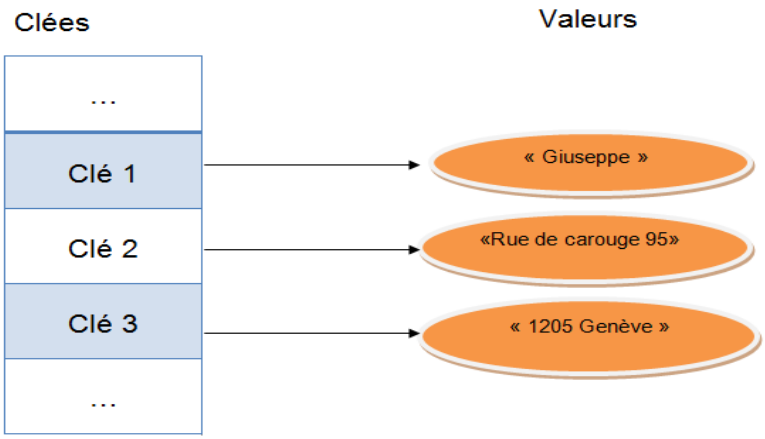

Figure 6. Aperçu des données dans une base de données type clé-valeur [21]

Dans cette figure, nous constatons que les valeurs « Giusepp », « Rue de carouge 95 » et « 1205 Genève » se voient attribuées respectivement les clé 1, 2 et 3.

Parmi les bases de données orientée colonne on peut citer Redis, DynamoDB, etc.

2.5.4 Base de données de type graphe

Les bases de données orientées graphe, contrairement aux bases clé-valeur, colonne ou document qui se concentrent sur les performances de traitement, sont particulièrement efficaces pour résoudre des problèmes complexes liés aux relations entre les données [21]. Par exemple, dans les réseaux sociaux comme Facebook ou Twitter, il existe de nombreuses relations entre les utilisateurs (amis, famille, etc.), et gérer ces connexions est bien plus difficile avec des bases relationnelles. En effet, pour chaque élément, il pourrait y avoir un nombre de relations exponentiel à stocker, ce qui génère des problèmes de performance et de complexité. Les bases de données graphe permettent de surmonter ces limitations en représentant les données sous forme de **nœuds** (utilisateurs) connectés par des **arêtes** (relations), simplifiant ainsi les requêtes complexes. Les bases de données graphe sont également utilisées pour gérer des structures informatiques complexes et créer des liens entre les intérêts des utilisateurs, comme dans le cas des publicités ciblées sur Facebook ou des recommandations d'achat sur des sites comme eBay et Amazon, basées sur les comportements d'achat précédents. Elles reposent sur trois principes fondamentaux :

- Un objet (par exemple, un utilisateur sur Facebook) est un **nœud** ;
- Les objets peuvent être reliés entre eux par des **relations** ;
- Chaque objet peut avoir des **attributs** (nom, statut social, etc.).

Cette approche simplifie les opérations sur les relations entre les données, qui seraient autrement très complexes à gérer avec des bases relationnelles [21].

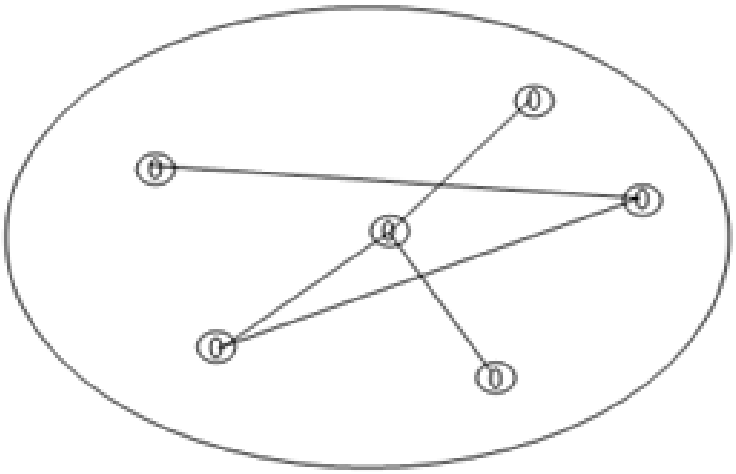

Figure 7. Structure de NoSQL orienté graph

La figure 7 illustre la structure d'une base de données graphique, montrant comment représenter les nœuds, les relations et les propriétés. Dans cet exemple, nous avons deux nœuds, représentés par des cercles A et B, qui correspondent respectivement à Banque et comptes. La relation entre les nœuds A et B est représentée par une ligne, indiquant qu'une banque possède plusieurs comptes. Cette relation peut être interprétée comme "une banque à un certain nombre de comptes".

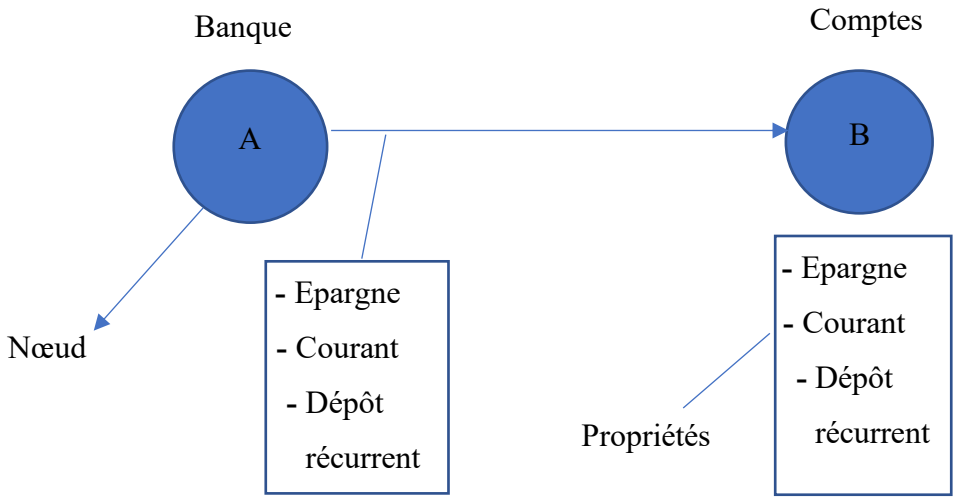

Figure 8. Modèle base de donnée NoSQL de type graphe [32]

Cette structure permet de visualiser les relations complexes entre les entités dans une base de données graphique. Les nœuds représentent les entités, les relations montrent comment ces entités interagissent, et les propriétés fournissent des détails supplémentaires sur chaque entité. Les bases de données graphiques sont particulièrement utiles pour modéliser des données

interconnectées, comme celles que l'on trouve dans les réseaux sociaux, les systèmes de recommandation, et bien sûr, les applications bancaires.

Parmi les bases de données orientée colonne on peut citer Neo4j, ArangoDB, etc.

Voici un tableau qui récapitule les modèles avec des exemples de bases de données NoSQL

Tableau 9. Tableau récapitulatif des modèles avec des exemples de base de données

Modèle NoSQL	Description	Exemples de Bases de Données
Clé-Valeur	Stocke les données sous forme de paires clé-valeur. Utilisé pour des requêtes simples.	Redis, Riak, DynamoDB
Colonne	Organise les données en colonnes plutôt qu'en lignes, adapté aux grandes quantités de données.	Cassandra, HBase, ScyllaDB
Document	Les données sont stockées sous forme de documents (souvent JSON ou BSON), offrant flexibilité et structure.	MongoDB, CouchDB, RavenDB
Graphe	Stocke les données sous forme de graphes, avec des nœuds et des arêtes représentant les relations.	Neo4j, ArangoDB, Amazon Neptune

2.6 Interrogations des NoSQL

Les bases de données NoSQL, en s'éloignant du paradigme relationnel, adoptent des approches d'interrogation spécifiques à leur modèle de données. Ces approches varient en fonction du type de base : clé-valeur, orientée colonnes, orientée documents ou orientée graphe. À ces modèles s'ajoutent des mécanismes transversaux pour optimiser la recherche dans des environnements massivement distribués [28].

2.6.1 Bases de données clé–valeur

Les bases clé–valeur, telles que Redis, Tokyo Cabinet ou Voldemort, reposent sur un principe d'accès direct aux données par une clé unique. Les opérations d'interrogation y sont généralement limitées à des commandes simples telles que GET, PUT, et DELETE. Cette simplicité assure des performances élevées mais limite considérablement les possibilités de filtrage ou de requête complexe. Pour contourner cette limitation, certaines bases ajoutent des extensions, comme Redis avec ses structures de données enrichies (listes, ensembles, sorted sets) ou des filtres sur les plages de valeurs. Des mécanismes alternatifs existent également, notamment l'association à une base relationnelle secondaire contenant les attributs

interrogeables. Cette dernière permet d'effectuer des recherches complexes, puis d'utiliser les clés retournées pour accéder aux données réelles dans la base clé-valeur [28].

Si on prend l'exemple de la figure 5, si on fait la requête GET 'id :1', cette requête récupère la valeur associée à la clé 1 qu'est « Giusepp ».

2.6.2 Bases de données orientées colonnes

Les bases orientées colonnes comme Cassandra ou HBase sont conçues pour la lecture en masse de données organisées par familles de colonnes. L'interrogation repose en grande partie sur l'accès par clé ou par plage de clés, ce qui rend ces bases particulièrement efficaces dans les systèmes d'analytique ou de séries temporelles. La capacité de recherche est généralement construite autour de structures comme les arbres B+ distribués, les tries préfixés ou les filtres probabilistes. Ces bases sont optimisées pour un schéma d'accès bien défini, et les modèles de requête complexes doivent être anticipés dès la conception du schéma, avec une attention particulière portée à la manière dont les données sont partitionnées dans le cluster [28].

A l'exemple de la figure 3, on peut avoir comme requête : SELECT EmpID, Salaire, Poste FROM employes WHERE EmpID = 200 ; et dans ce cas nous aurons comme résultat les informations de l'employé qui a l'identifiant numéro 200.

2.6.3 Bases de données orientées documents

Les bases orientées documents telles que MongoDB ou CouchDB proposent des capacités d'interrogation beaucoup plus avancées. MongoDB, par exemple, permet de formuler des requêtes complexes en utilisant une syntaxe de type JSON, intégrant des opérateurs de comparaison, des expressions logiques (AND, OR, NOT), des conditions sur les tableaux, et des fonctions d'agrégation poussées. De son côté, CouchDB repose sur un modèle d'interrogation basé sur la définition de vues fonctionnelles, construites à l'aide de fonctions map et reduce en JavaScript. Ces vues servent d'index consultables et permettent une interrogation optimisée des structures stockées. Ces bases sont bien adaptées aux cas où les données sont semi-structurées, évolutives, et nécessitent une flexibilité dans les critères de recherche [28]. A l'exemple de la figure 4, si on fait la requête db.bankdatabase.find(), cette requête recherche tous les documents de la collection (2 documents) et affiches toutes les informations correspondantes.

2.6.4 Bases de données orientées graphe

Les bases orientées graphe, à l'image de Neo4j, offrent un modèle d'interrogation particulièrement performant pour l'analyse des relations entre entités. Ces bases utilisent un

langage dédié, tel que Cypher, qui permet d'exprimer des motifs de relations complexes sous forme de requêtes déclaratives. Par exemple, il est possible de rechercher tous les chemins entre deux types de nœuds ou d'identifier des motifs relationnels tels que des triangles ou des chaînes. Ce type de base est extrêmement efficace pour des usages comme l'analyse de réseaux sociaux, la détection de fraude ou la recommandation. Les performances sont souvent supérieures à celles des autres modèles, notamment lorsque les requêtes impliquent de nombreux sauts entre entités [32].

En se basant sur la figure 7, si on tape la requête MATCH (a:Banque)-[:POSSÈDE]->(b:Compte)

WHERE b.type = 'Épargne'

RETURN a, b ;

Cette requête trouve toutes les banques qui possèdent des comptes de type « Épargne » et retourne les nœuds de la banque et du compte.

2.6.5 Stratégies d'interrogation distribuée

Dans les systèmes NoSQL distribués, plusieurs mécanismes ont été développés pour pallier l'absence d'un langage de requête unifié. L'approche scatter/gather consiste à diffuser la requête sur l'ensemble des nœuds du système, chacun exécutant la recherche localement avant un regroupement des résultats [33]. Par exemple, imaginons un système NoSQL distribué où nous souhaitons rechercher des utilisateurs par leur nom. La requête est diffusée à tous les nœuds du système. Chaque nœud exécute la recherche localement dans sa propre base de données. La recherche se fera en 3 étapes :

- **Étape 1** : La requête "Rechercher tous les utilisateurs avec le nom 'Mamadou'" est envoyée à tous les nœuds ;
- **Étape 2** : Chaque nœud recherche localement les utilisateurs correspondants ;
- **Étape 3** : Les résultats de chaque nœud sont renvoyés au nœud d'origine, qui les regroupe et renvoie la réponse finale.

D'autres techniques reposent sur des structures d'index distribuées, telles que les arbres B+ partitionnés ou les tries répartis [33]. L'utilisation de filtres de Bloom permet également d'optimiser les performances en éliminant à l'avance les recherches infructueuses[9]. Enfin, dans certains cas, des stratégies hybrides peuvent être mises en œuvre, combinant les capacités de

[9] https://hypertable.com/documentation/architecture/

traitement des bases NoSQL à des systèmes relationnels auxiliaires pour permettre des interrogations plus expressives.

2.7 Points forts

Les bases de données NoSQL présentent plusieurs atouts voire points forts :

- **Gestion efficace de volume massifs de données** : Les bases NoSQL sont conçues pour stocker et traiter des volumes de données extrêmement importants, dépassant souvent les capacités des SGBD traditionnelles. Elles sont capables de gérer des données à l'échelle du web, souvent réparties sur des centaines de serveurs, ce qui les rend particulièrement adaptées aux applications de type Big Data [22] ;
- **Scalabilité horizontale** : Contrairement aux systèmes relationnels qui nécessitent souvent un renforcement vertical (ajout de puissance à une seule machine), les bases NoSQL permettent d'ajouter facilement de nouveaux serveurs (nœuds) pour augmenter la capacité du système. Ce mécanisme assure une montée en charge fluide et sans interruption de service [22] ;
- **Absence de schéma rigide d'où la flexibilité du modèle de données** : Dans les bases NoSQL, les données peuvent être insérées sans respecter un schéma prédéfini. Cette absence de contrainte permet de stocker des enregistrements avec des structures différentes dans une même base, ce qui est très utile dans les environnements où les données changent fréquemment ou sont hétérogènes [24] ;
- **Evolutivité des données et leur variété** : Les bases NoSQL supportent naturellement des formats variés, comme les données semi-structurées (JSON, XML) ou non structurées (texte libre, images, logs). Elles permettent ainsi d'intégrer facilement de nouvelles sources de données sans avoir à restructurer l'ensemble du système [25] ;
- **Suppression du mapping objet-relationnel (ORM)** : Les SGBD relationnels doivent nécessairement une traduction (ORM) entre les objets utilisés dans les langages de programmation et les tables relationnelles. Les bases NoSQL, en adoptant des modèles orientés documents ou objets, suppriment ce besoin, ce qui accélère le développement et réduit les erreurs de transformation [28] ;
- **Haute disponibilité et tolérance aux pannes** : Les données sont généralement répliquées sur plusieurs nœuds, ce qui garantit leur accessibilité même en cas de panne partielle du système. Certains moteurs NoSQL intègrent des mécanismes automatiques de reprise après incident et de répartition des charges [26] ;

- **Performances élevées :** Grâce à des stratégies de cache en mémoire, de réplication intelligente et de partitionnement des données, les bases NoSQL offrent des temps de réponse très courts, même en cas de forte sollicitation. Elles sont optimisées pour les opérations simples, très fréquentes et massivement parallèles [34].

2.8 Limites

Par contre, comme tout système, les base de données NoSQL présentent certaines limites que sont :

- **Négligence de la cohérence immédiate des données** : Pour assurer une haute disponibilité et tolérance aux partitions (théorème CAP), de nombreuses bases NoSQL sacrifient la cohérence immédiate des données. Cela signifie que les mises à jour ne sont pas instantanément visibles sur tous les nœuds, ce qui peut poser problème dans les applications sensibles à la cohérence (ex. : banque, santé) [21] ;
- **Absence de standardisation du langage de requête** : Contrairement à SQL qui est universel et normé, chaque base NoSQL possède son propre langage ou interface (MongoDB, Cassandra, etc.). Cela rend la formation, la migration et l'interopérabilité plus complexes, car les compétences ne sont pas transférables entre moteurs [21] ;
- **Performance des Requêtes et Jointures (implicite)** : les bases NoSQL, qui sont souvent dénormalisées et n'ont pas de capacités de jointure natives équivalentes à SQL, pourraient être moins efficaces pour les requêtes structurées avec des jointures complexes [30] ;
- **Interopérabilité et portabilité limitées** : Le passage d'un moteur NoSQL à un autre n'est généralement pas simple, en raison de l'incompatibilité des modèles de données, des langages de requête et des mécanismes internes. Cela rend les architectures plus rigides sur le long terme [28].

2.9 Comparaison avec SGBDR

Plusieurs critères peuvent-être utilisés pour comparer ces deux formes de SGBD.

2.9.1 Comparaison suivant l'usage

L'utilisation d'un système dépend fortement du contexte fonctionnel et technique dans lequel il est déployé. Les SGBDR sont particulièrement adaptés aux systèmes classiques de gestion d'entreprise (comptabilité, facturation, gestion RH, finance, santé…), par exemple, une entreprise de santé peut utiliser un SGBDR pour gérer les dossiers médicaux de ses patients où

les règles métier sont bien définies, les données structurées et les transactions critiques. Dans ce contexte, la robustesse et la fiabilité des SGBDR sont largement reconnues, surtout lorsque des transactions ACID (Atomicité, Cohérence, Isolation, Durabilité) sont essentielles pour assurer que les opérations, comme l'enregistrement d'un nouveau patient ou la mise à jour de son traitement, sont exécutées de manière fiable et sécurisée. Les bases NoSQL, en revanche, sont conçues pour des cas d'usage émergents caractérisés par des données massives, variées et en constante évolution [34]. Elles sont privilégiées dans les applications web modernes**,** les réseaux sociaux**,** le Big Data**,** les systèmes de recommandation**,** l'IoT ou encore les plateformes de streaming. Par exemple, une plateforme de streaming comme Netflix utilise une base de données NoSQL pour gérer les recommandations de films et de séries. Dans ce contexte, les données peuvent provenir de sources diverses, comme les historiques de visionnage des utilisateurs, les évaluations, et les métadonnées des films, qui peuvent être de nature structurée, semi-structurée ou non structurée La flexibilité structurelle des bases NoSQL permet de s'adapter à des modèles de données en constante évolution, sans nécessiter de schéma rigide. De plus, leur évolutivité horizontale leur permet d'ajouter facilement de nouveaux nœuds au système pour gérer l'augmentation des volumes de données et des utilisateurs. Cela leur permet de répondre efficacement aux exigences de disponibilité, de montée en charge, et de tolérance aux pannes, ce qui est crucial dans des environnements où la continuité du service et la rapidité d'accès aux données sont primordiales. En somme, le choix entre SGBDR et NoSQL dépend des besoins spécifiques de l'application et des caractéristiques des données à gérer.

2.9.2 Suivant le type de données

Les SGBDR reposent sur une structure tabulaire stricte, où les données sont organisées en lignes et colonnes selon un schéma défini. Cette approche convient parfaitement aux données fortement structurées et homogènes, telles que les enregistrements comptables, les fiches clients ou les inventaires [24]. Par exemple, une petite entreprise de parfumerie peut utiliser un SGBDR pour gérer son inventaire, où chaque produit est représenté par une ligne dans une table, avec des colonnes pour le nom, le prix, la quantité en stock, etc. Cependant, cette structure impose une certaine rigidité : toute modification du schéma, comme l'ajout d'une nouvelle colonne pour un attribut supplémentaire, nécessite souvent une intervention lourde sur la base existante, ce qui peut entraîner des temps d'arrêt et des coûts supplémentaires.

En revanche, les bases NoSQL prennent en charge une plus grande diversité de formats de données. Elles peuvent stocker des données non structurées ou semi-structurées, comme des

fichiers JSON, XML, des documents texte, des images, des vidéos, ou encore des graphiques complexes. Par exemple, une application de réseau social peut utiliser une base de données NoSQL pour gérer les profils des utilisateurs, où chaque profil peut contenir des informations variées, telles que des photos, des publications, et des commentaires, sans nécessiter un schéma rigide. Chaque famille de base NoSQL (clé-valeur, document, colonne, graphe) est optimisée pour un type de données ou d'accès particulier, offrant ainsi une adaptabilité élevée aux besoins des applications modernes. Cette flexibilité permet aux développeurs de s'adapter rapidement aux changements de données et aux exigences des utilisateurs, rendant les bases NoSQL particulièrement adaptées aux environnements dynamiques et en constante évolution.

2.9.3 Suivant la volumétrie

En termes de gestion de la volumétrie, les SGBDR montrent leurs limites au-delà d'un certain seuil. Leur architecture centralisée et leur dépendance à la scalabilité verticale impliquent que l'augmentation du volume de données nécessite des serveurs plus puissants. Cela signifie que pour chaque augmentation significative des données, les entreprises doivent investir dans des serveurs de plus en plus performants, ce qui peut s'avérer coûteux et peu flexible. De plus, cette approche peut entraîner des temps d'arrêt lors des mises à niveau, ce qui perturbe les opérations commerciales. Par exemple, une entreprise qui gère une base de données de clients et de transactions peut se retrouver à devoir faire face à des coûts d'infrastructure élevés et à des interruptions de service, rendant difficile la gestion efficace de ses opérations.

À l'inverse, les bases NoSQL ont été conçues spécifiquement pour gérer de très grandes quantités de données, ce qui les rend particulièrement adaptées aux environnements modernes où les volumes de données explosent [29]. Leur capacité à répartir les données sur plusieurs serveurs grâce au partitionnement horizontal permet une gestion fluide de la volumétrie, sans dégradation significative des performances. Ce modèle de traitement est non seulement efficace, mais il est également essentiel dans les domaines du Big Data, où les données sont générées en continu et en grande quantité. Par exemple, une plateforme de streaming comme YouTube ou Twitter utilisent des bases NoSQL pour stocker et gérer des millions de vidéos et de métadonnées associées. En répartissant les données sur de nombreux serveurs, exemple, YouTube peut assurer un accès rapide et efficace à son contenu, même lorsque le nombre d'utilisateurs et le volume de données augmentent de manière exponentielle. Cette flexibilité et cette capacité à évoluer en fonction des besoins font des bases NoSQL un choix privilégié pour les applications modernes nécessitant une gestion efficace de la volumétrie, permettant aux

entreprises de s'adapter rapidement aux défis posés par l'augmentation des données tout en maintenant des performances optimales.

2.9.4 Suivant la scalabilité

La scalabilité est un critère fondamental dans le choix d'un système de gestion de bases de données (SGBD), car elle détermine la capacité d'un système à s'adapter à l'augmentation des volumes de données et des utilisateurs. Les SGBDR offrent principalement une scalabilité verticale [32] : pour augmenter la capacité, il est nécessaire de renforcer la machine hôte en améliorant des composants tels que le processeur, la RAM ou le disque. Ce modèle, bien que simple à mettre en œuvre dans un premier temps, atteint rapidement ses limites physiques et économiques [28]. Par exemple, une entreprise qui doit gérer une augmentation soudaine de ses données peut se retrouver à devoir investir dans des serveurs de plus en plus puissants, ce qui peut entraîner des coûts colossaux et des temps d'arrêt lors des mises à niveau.

À l'inverse, les bases NoSQL sont fondées sur une scalabilité horizontale, ce qui leur permet d'ajouter dynamiquement de nouveaux nœuds au cluster. Cette approche permet de répartir automatiquement les données et la charge de travail entre plusieurs serveurs, ce qui garantit une montée en charge progressive, économique et sans interruption du service [25]. Par exemple, une entreprise de commerce électronique qui connaît des pics de trafic pendant les périodes de soldes peut facilement ajouter de nouveaux nœuds à son infrastructure NoSQL pour gérer l'augmentation du nombre d'utilisateurs et des transactions, tout en maintenant des performances optimales. Ce mécanisme de scalabilité horizontale en fait la solution privilégiée pour les infrastructures cloud et les systèmes à trafic variable, car il permet aux entreprises de s'adapter rapidement aux fluctuations de la demande sans compromettre la qualité du service. En somme, la scalabilité horizontale des bases NoSQL offre une flexibilité et une efficacité qui sont essentielles dans le paysage numérique actuel, où les besoins en données évoluent rapidement.

2.9.5 Suivant la performance

Du point de vue des performances, les SGBDR (Systèmes de Gestion de Bases de Données Relationnelles) se distinguent par leur efficacité dans le traitement des requêtes complexes, des opérations transactionnelles et des jointures relationnelles, à condition que la volumétrie des données reste raisonnable [22]. Grâce aussi aux systèmes d'optimisation de requête comme les indexations, ces systèmes peuvent exécuter des requêtes complexes de manière efficace, tout en garantissant des transactions conformes aux propriétés ACID (Atomicité, Cohérence,

Isolation, Durabilité). Cette maturité logicielle, acquise au fil des décennies, en fait un choix privilégié pour des applications critiques, telles que les systèmes bancaires ou de gestion d'entreprise, où la précision et la fiabilité des données sont primordiales.

Cependant, les bases NoSQL adoptent une approche différente, souvent en sacrifiant certaines garanties, comme la cohérence forte, pour offrir une meilleure rapidité d'exécution, notamment en lecture et écriture. Ces systèmes sont particulièrement performants dans les opérations simples et massivement parallèles, grâce à leur architecture distribuée qui permet de traiter les données sur plusieurs nœuds simultanément. Par exemple, dans des contextes où la latence doit être minimale, comme la recherche en temps réel, l'analyse de flux de données ou les systèmes embarqués, les bases NoSQL peuvent surpasser les SGBDR en termes de rapidité. Des entreprises comme Twitter ou Facebook utilisent des bases NoSQL pour gérer des volumes massifs de données générées en temps réel, où la capacité à traiter rapidement des requêtes simples est cruciale pour l'expérience utilisateur. En somme, bien que les SGBDR soient excellents pour des opérations complexes dans des environnements contrôlés, les bases NoSQL se révèlent souvent plus adaptées aux applications nécessitant une haute performance et une faible latence dans des contextes de données massives et dynamiques.

2.9.6 Analyse comparative

Afin d'approfondir cette distinction entre les modèles relationnels et non relationnels, une analyse comparative a été réalisée selon plusieurs critères techniques et fonctionnels en se basant sur des études de différents auteurs. Cette analyse permet de mieux comprendre les contextes d'usage adaptés à chaque technologie. En effet, les bases relationnelles (SQL) sont historiquement privilégiées pour assurer l'intégrité et la cohérence des données grâce à leur modèle structuré et leurs mécanismes de transactions (ACID) [7]. Cependant, elles rencontrent des limites en matière de scalabilité horizontale, ce qui rend leur utilisation difficile dans des environnements nécessitant un traitement rapide et distribué de grandes quantités de données (vélocité et volume) [22]. En revanche, les bases NoSQL privilégient la flexibilité et la scalabilité en permettant de stocker des données non structurées ou semi-structurées (variété) sur plusieurs serveurs en parallèle (scalabilité horizontale), mais elles sacrifient parfois la cohérence stricte au profit de la disponibilité et de la performance [23]. Certains travaux montrent que le choix entre SQL et NoSQL dépend du contexte et des besoins spécifiques, notamment en termes de scalabilité, redondance, flexibilité et vitesse d'accès. Le NoSQL excelle dans les environnements Big Data ou avec des données semi-structurées, tandis que le

SQL est adapté aux systèmes nécessitant une forte cohérence et des structures bien définies [7]. De plus, certains chercheurs ont démontré que l'utilisation de NoSQL en complément des SGBD relationnels améliore considérablement les performances des systèmes, en particulier pour les environnements nécessitant un traitement rapide et efficace des données [15]. Par ailleurs, certains travaux expliquent que les bases NoSQL, bien que moins sécurisées que les bases relationnelles, offrent une meilleure capacité de gestion des grandes quantités de données, ce qui les rend adaptées aux besoins modernes comme le cloud computing. Cependant, elles présentent des faiblesses en matière de sécurité [16]. En outre, certains chercheurs ont comparé les performances des bases SQL et NoSQL, notamment pour les opérations de lecture, écriture et suppression. Ils ont constaté que, bien que les bases de données NoSQL soient généralement optimisées pour les magasins clé-valeur, toutes les bases NoSQL ne performent pas mieux que les bases SQL testées [22]. Enfin, les modèles relationnels et NoSQL ne sont pas mutuellement exclusifs mais complémentaires, chaque modèle présentant des avantages et des limites dans différents contextes. Les bases NoSQL sont plus adaptées aux applications décentralisées avec des données non structurées ou semi-structurées nécessitant une évolutivité horizontale et une disponibilité permanente [23].

L'hybridation des technologies SQL et NoSQL a également été explorée par certains chercheurs pour concilier les avantages des deux approches. Par exemple, certaines entreprises combinent un SGBD relationnel pour la gestion des transactions et un système NoSQL pour l'analyse de données en temps réel. Cette solution permettrait d'utiliser NoSQL pour le stockage distribué et SQL pour la gestion de transactions complexes. Toutefois, cette intégration peut s'avérer complexe à mettre en œuvre et impliquer des coûts de maintenance élevés, notamment en raison de la nécessité de synchroniser les deux bases [33].

2.9.7 DISCUSSION

En définitive, l'analyse comparative des modèles relationnels et non relationnels montre que chacun répond à des exigences techniques spécifiques, qu'aucun ne couvre entièrement à lui seul. Les bases relationnelles s'illustrent par leur fiabilité transactionnelle, leur structure rigide adaptée aux environnements fortement normalisés, ainsi que par une maturité qui leur confère une stabilité dans les systèmes critiques. Elles restent irremplaçables dans les domaines où la cohérence stricte, l'intégrité référentielle et le respect de contraintes métier sont incontournables. À l'inverse, les bases NoSQL tirent leur force de leur capacité à s'adapter aux nouvelles formes de données (non structurées ou semi-structurées), à absorber des volumes

massifs et à supporter une forte variabilité des charges via la scalabilité horizontale. Elles offrent une réponse technique pertinente aux exigences des systèmes modernes orientés vers la rapidité, la flexibilité et la distribution. Leurs mécanismes de réplication et leur tolérance aux pannes répondent bien aux attentes des applications décentralisées et des plateformes en temps réel. Au-delà de cette opposition apparente, l'étude révèle surtout une complémentarité croissante entre les deux approches. L'émergence de systèmes hybrides tend à démontrer qu'une stratégie combinée permet d'optimiser les performances globales : le relationnel pour la rigueur et la sécurité des transactions et le NoSQL pour la gestion fluide de la masse de données et la vitesse d'accès dans des environnements distribués.

En somme, le choix entre SQL et NoSQL ne doit pas être envisagé comme un affrontement technologique, mais plutôt comme une réponse contextuelle aux besoins spécifiques d'une application. La nature des données, les exigences de cohérence, la volumétrie, la structure du système et les contraintes de temps réel doivent guider l'architecte dans sa décision. Dans de nombreux cas, l'intégration judicieuse des deux modèles peut constituer la solution la plus efficace.

Tableau 10.Tableau récapitulatif de l'étude comparative

Critères	**SGBDR**	**NoSQL**
Usage principal	Applications classiques d'entreprise : comptabilité, finance, santé, gestion RH.	Applications modernes : Big Data, web, réseaux sociaux, streaming, IoT.
Type de données	Données structurées : Organisées en tables avec un schéma rigide	Données variées : structurées, semi-structurées, non structurées (JSON, XML, images, vidéos).
Volumétrie	Gestion inefficace de gros volumes de données évolutives	Gestion efficace de très gros volumes sans interruption.
Scalabilité	Verticale : ajouter ressources (CPU, RAM) au même serveur.	Horizontale : ajouter facilement des nœuds au cluster, économique, flexible et sans interruption.
Performance	Optimisé pour requêtes complexes et transactions ACID.	Haute performance sur requêtes simples et massivement parallèles grâce à la réplication.
Type de schéma	Schéma fixe, ne s'adapte pas aux évolutions de données	Schéma flexible et dynamique, s'adapte facilement aux évolutions des données.

Chapitre 3 : Migration SGBDR et NoSQL

3.1 La migration de données

La migration des données consiste à transférer ou convertir des données d'un système ou d'un emplacement vers un autre. Ce processus est généralement lié à l'introduction d'un nouveau système, à un changement d'emplacement des données ou à une mise à jour de leur format. Elle accompagne souvent des projets de migration ou de consolidation, comme le remplacement de systèmes obsolètes ou l'intégration de nouvelles applications utilisant les mêmes données ou aussi le fait qu'un système ne peut plus gérer la quantité de données qu'elle génère [27] comme c'est le cas des SGBDR. Les entreprises effectuent ce type de migration pour moderniser leur infrastructure, optimiser leurs applications ou adopter de nouvelles solutions de stockage.[10] En ce qui nous concerne, la migration de données consistera à passer d'un système à un autre, autrement dit, d'un type de base de donnée en un autre, et plus explicitement, il s'agira soit de passer du SQL vers le NoSQL ou soit de passer du NoSQL vers le SQL. Compte tenu des tendances actuelles, les entreprises passent du SQL vers la NoSQL.

La migration des données se divise en deux grandes étapes : la traduction du schéma et la migration des données elles-mêmes.

3.1.1 La Réingénierie des données

La réingénierie des données consiste à extraire, transformer et migrer une base de données existante vers un nouveau système tout en adaptant les logiciels associés, sans perte de sémantique ni de fonctionnalités [34]. Elle se compose de trois étapes principales [34] :

- **La conversion du schéma :** Elle est la première étape. Elle consiste à adapter la structure des données (le schéma) de la base source au schéma compatible avec la base cible. Elle se décompose en deux sous-étapes [35] :
    - **Récupération du schéma conceptuel** : Cette étape vise à analyser et comprendre le modèle conceptuel sous-jacent de la base source (par exemple, un modèle entité-relation). Elle permet d'identifier les sémantiques explicites, Il s'agira de comprendre les relations, c'est-à-dire, comment les tables ou entités sont connectées. Tout de même, comprendre les attributs c'est-à-dire les colonnes ou propriétés des tables. Aussi comprendre les clés, c'est-à-dire les identifiants uniques ou clés primaires et enfin comprendre les dépendances des

[10] https://www.netapp.com/fr/data-management/what-is-data-migration/#:~:text=La%20migration%20des%20donn%C3%A9es%20consiste,nouvel%20emplacement%20pour%20les%20donn%C3%A9es.

données, c'est-à-dire, les relations entre attributs (exemple : une clé étrangère). Les sémantiques implicites sont également extraites. Celles-ci incluent des informations qui ne sont pas directement visibles dans le schéma, comme des relations implicites ou des règles métiers qui régissent les données. On appelle tout ce processus une rétro-ingénierie ou ingénierie inverse, parce qu'il s'agit d'analyser une structure existante pour en extraire sa logique et sa conception ;

- **Génération du schéma physique cible :** Cette étape utilise le schéma conceptuel récupéré à l'étape précédente pour construire un schéma physique adapté à la base cible. Ce schéma physique comprend la structure détaillée des données (tables, colonnes, types de données, etc.) selon les spécifications et la technologie de la base cible. Ce processus est appelé l'ingénierie directe.

Une fois que le schéma cible est défini, les données de la base source sont chargées, transformées et migrées vers la base cible. Cette étape implique souvent une conversion des formats de données pour qu'elles s'adaptent aux spécifications de la base cible.

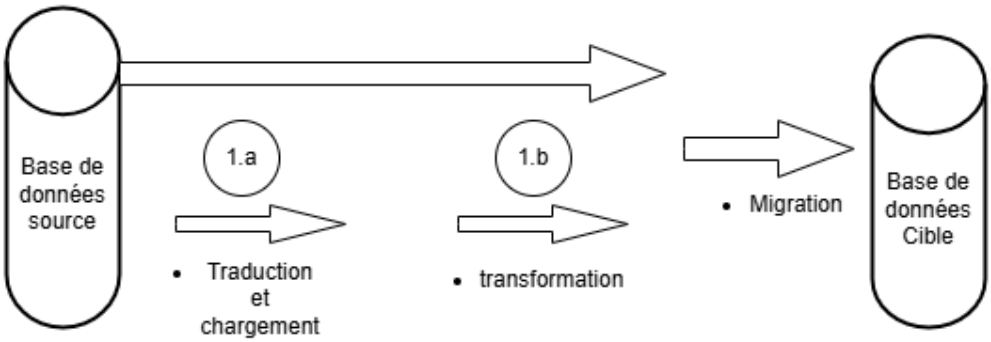

Figure 9. Processus de migration de données

- **La conversion des données** : Cela consiste à faire une migration des instances de données selon le nouveau schéma, tout en respectant les contraintes et en assurant la qualité des données migrées [34] ;
- **La conversion des programmes** : Cela consiste à faire des modifications des applications afin qu'elles puissent accéder aux données migrées sans altérer leur logique métier [34].

3.1.1.1 Stratégies de conversions

Deux grandes stratégies de conversion de schéma sont présentées [34] :

- **Conversion physique directe (D1)** : Chaque table et champ du système legacy est traduit en table et colonne correspondantes dans le tout nouveau système, sans interprétation sémantique. Cette méthode est rapide et peu coûteuse mais produit une base de qualité médiocre, héritant des défauts du système initial donc elle n'est pas recommandée ;

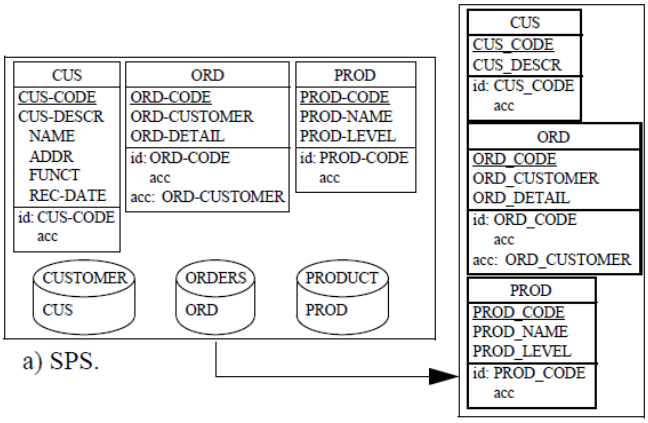

Figure 10. Conversion physique directe [34]

- **Conversion conceptuelle (D2) :** Elle repose sur l'ingénierie inverse complète, permettant d'extraire un modèle conceptuel riche et précis, qui est ensuite transformé en schéma physique pour le nouveau SGBD. Cette approche est coûteuse mais produit une base de haute qualité, normalisée et bien documentée.

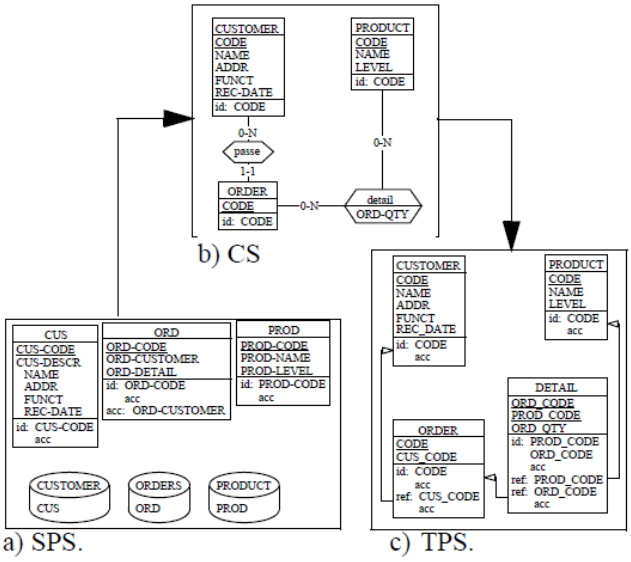

Figure 11. Conversion conceptuelle [34]

Tout de même, pour la conversion des programmes, trois stratégies sont identifiées [34] :

- **Utilisation de wrappers (P1)** : Des composants intermédiaires (wrappers) encapsulent l'accès aux données, permettant aux programmes legacy d'utiliser la nouvelle base sans modification de leur code. Cette méthode est rapide et maintient la logique existante ;

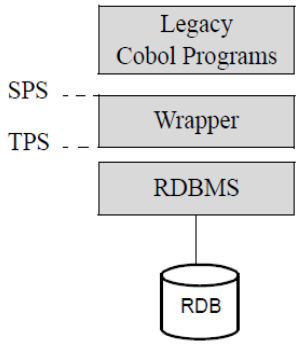

Figure 12. Utilisation de Wrappers [36]

- **Réécriture des accès (P2)** : Remplacer les instructions d'accès (ex : READ en COBOL) par des requêtes adaptées au nouveau SGBD (ex : SQL SELECT). Cette stratégie est partiellement automatisable mais peut complexifier le code ;
- **Réécriture complète (P3)** : La logique du programme est réécrite pour exploiter pleinement la puissance et les fonctionnalités du nouveau SGBD. C'est la méthode la plus complexe mais elle produit un code plus propre, moderne et performant.

3.2 Formes de migrations

Ils existent plusieurs formes de migrations qui peuvent être aussi appelées techniques de migrations. On en trouve : la technique par utilisation de framework, par outils spécialisés, par transformation directe des requêtes, par ETL, par conversion optimisée des schémas, par migration de schéma et mappage, par algorithme automatisé et la technique avec utilisation d'API et de scripts.

3.2.1 Technique de migration par utilisation de framework

Des recherches sur le domaine notamment [36] met en lumière une bonne technique de migration : l'utilisation de frameworks spécialisés conçus pour automatiser tout ou partie du processus de migration des bases relationnelles vers les bases NoSQL. Parmi ces outils, NoSQLayer se distingue comme l'un des plus pertinents. Ce framework a été conçu pour faciliter la transformation des modèles relationnels SQL vers des structures adaptées à MongoDB, l'une des bases NoSQL les plus populaires, notamment dans le domaine des applications web et des systèmes distribués [36]. NoSQLayer fonctionne en analysant le schéma relationnel existant (tables, relations, clés, types de données) et en le convertissant

automatiquement en documents JSON. Cette conversion respecte les principes fondamentaux du modèle document-oriented, où chaque enregistrement est représenté comme un document structuré et potentiellement imbriqué [37]. Par exemple, une relation maître-détail dans une base SQL pourra être représentée dans MongoDB par un document contenant un tableau d'éléments enfants.

Les avantages de l'utilisation de NoSQLayer sont multiples [37] :

- ✓ **Automatisation partielle ou complète** du processus de migration, réduisant ainsi le temps et les erreurs humaines ;
- ✓ **Traitement efficace de grands volumes de données**, ce qui est essentiel dans les contextes Big Data ;
- ✓ **Fiabilité accrue** de la structure migrée, grâce à des règles de conversion codifiées ;
- ✓ **Souplesse** dans l'adaptation des structures relationnelles vers un modèle plus flexible comme celui de MongoDB.

Cependant, bien que les frameworks comme NoSQLayer représentent une avancée importante, ils ne sont pas toujours adaptés à toutes les situations. En effet, leur efficacité dépend fortement de la complexité du schéma initial, de la qualité des métadonnées et des besoins fonctionnels spécifiques de l'application cible. De plus, ces outils se concentrent souvent sur un seul type de base NoSQL (MongoDB), ce qui limite leur portabilité vers d'autres modèles comme les bases orientées colonnes ou les bases orientées graphes et cette technique n'est pas bidirectionnelle.

En conclusion, les frameworks spécialisés comme NoSQLayer offrent une solution intéressante pour automatiser et fiabiliser les migrations vers NoSQL. Ils s'intègrent parfaitement dans une démarche plus large combinant guidelines, approche méthodologique, et outillage afin d'assurer une transition efficace, scalable et conforme aux exigences métier.

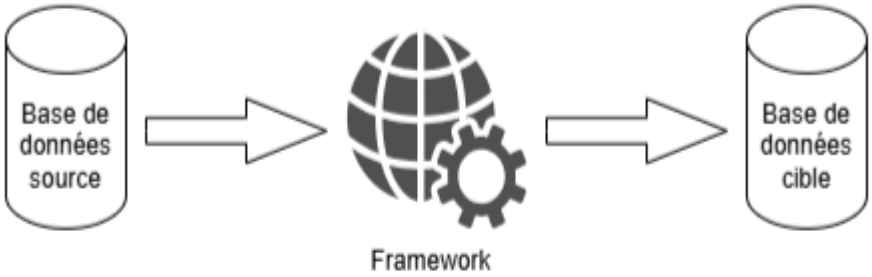

Figure 13. Migration par framework

3.2.2 Technique de migration par outils spécialisés

Plusieurs travaux de recherche et retours d'expérience soulignent que l'utilisation d'Apache Sqoop est devenue courante dans les projets de migration de bases de données relationnelles vers des systèmes distribués comme HBase, en particulier dans des contextes de traitement de données massives [36]. En effet, avec l'émergence du Big Data, les entreprises et institutions se retrouvent face à des volumes de données exponentiels, générés par des applications web, mobiles, IoT ou encore des systèmes d'information traditionnels. Les bases relationnelles classiques atteignent rapidement leurs limites en matière de scalabilité et de performance, ce qui rend nécessaire l'adoption de solutions alternatives comme les bases NoSQL distribuées. Dans ce contexte, Apache Sqoop se positionne comme un outil stratégique facilitant le transfert automatisé de données entre les SGBD relationnels (comme MySQL, PostgreSQL, Oracle ou SQL Server) et les plateformes distribuées de l'écosystème Hadoop, telles que HDFS (Hadoop Distributed File System) ou HBase. Son architecture repose sur le framework MapReduce, ce qui lui permet de paralléliser les opérations d'importation et d'exportation, réduisant ainsi considérablement les temps de traitement [35].

L'un des principaux atouts de Sqoop est sa capacité à gérer efficacement des volumes très importants de données, tout en maintenant une intégrité et une cohérence structurelle lors du transfert. Cette automatisation permet de [35] :

- Réduire significativement les erreurs humaines liées aux migrations manuelles ;
- Accélérer le processus de migration grâce à la parallélisation ;
- Maintenir la synchronisation entre deux systèmes hétérogènes (SQL et NoSQL) dans le cadre d'une architecture hybride.

Sqoop prend en charge les imports complets ou incrémentiels, permet de filtrer les données à migrer (via requêtes SQL), et autorise également l'exportation des données NoSQL vers des systèmes relationnels, ce qui en fait un outil bidirectionnel et flexible. Apache Sqoop couplé à HBase constitue un duo incontournable pour les projets nécessitant une migration efficace de données relationnelles vers des architectures NoSQL distribuées [35]. Cette approche offre un gain de temps considérable, une réduction des risques d'erreurs, et surtout, une capacité d'évolution vers des systèmes plus puissants, résilients et adaptés aux exigences du Big Data. Elle permet aux entreprises de valoriser leurs données historiques tout en s'inscrivant dans une logique d'innovation technologique et de transformation numérique. Mais par contre, cette

technique a 3 limites : Fonctionne plus avec les BDD type colonne, fonctionne en mode batch et fonctionne avec l'utilisation de MapReduce.

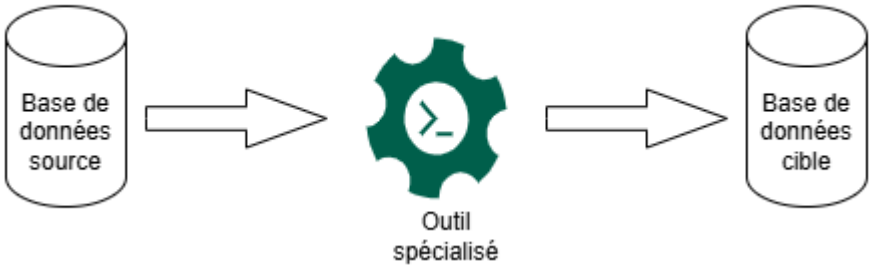

Figure 14. Technique par utilisation d'outils spécialisés

3.2.3 Technique de migration par transformation directe des requêtes

Les techniques de migration des bases de données relationnelles vers des systèmes NoSQL varient considérablement en fonction des objectifs du projet, des contraintes techniques, du budget et du temps disponible. Parmi ces méthodes, une approche particulièrement intéressante est celle dite de la transformation directe des requêtes. Cette stratégie repose sur l'idée de convertir automatiquement les requêtes SQL existantes souvent profondément ancrées dans la logique applicative en requêtes compatibles avec les bases de données NoSQL cibles, sans altérer la structure fondamentale de l'application. C'est le cas, par exemple, de la méthode appelée Mid-model Approach using Data and Query Features, décrite dans certains travaux de recherche récents [38] [39]. Dans ce modèle, l'objectif principal est de préserver la logique métier en réutilisant les requêtes SQL existantes tout en les adaptant à la nouvelle structure NoSQL. Cela est rendu possible grâce à une analyse des caractéristiques des données (types, relations, cardinalité) et des requêtes elles-mêmes (sélections, jointures, agrégations, etc.).
L'approche suit généralement les étapes suivantes [39] :

- Analyse syntaxique et sémantique des requêtes SQL ;
- Mapping entre les entités relationnelles et les structures NoSQL cibles (documents, collections, colonnes, etc.) ;
- Génération automatique de requêtes NoSQL équivalentes ;
- Insertion dans l'application sans toucher à la logique métier ni à l'interface utilisateur.

Cette approche présente plusieurs avantages, parmi ceux on peut citer [38] :

- ✓ **Migration rapide et non intrusive** : elle permet de conserver la structure applicative et les processus métiers existants ;

- ✓ **Réduction du coût de refactoring** : pas besoin de réécrire toute l'application ou de redéfinir les règles de gestion ;
- ✓ **Bonne adaptation aux environnements agiles ou à forte contrainte temporelle**, où la mise en œuvre rapide est essentielle ;
- ✓ **Moins de risques de régression fonctionnelle**, car la logique reste intacte.

Mais par contre, celle-ci présente des limites, parmi ceux on peut énumérer [38] :

- **Complexité des requêtes SQL avancées :** Les requêtes imbriquées ou fonctions analytiques peuvent être difficiles à convertir de manière équivalente dans un langage NoSQL ;
- **Perte potentielle d'optimisation** : les moteurs NoSQL ne traitent pas les requêtes de la même manière que les SGBD relationnels ;
- **Incompatibilité avec certains systèmes** : Cette approche ne convient pas à tous les types de bases NoSQL, notamment celles très éloignées du modèle relationnel comme les bases orientées graphes.

Ainsi, ce type d'approche est particulièrement utile dans les contextes suivants :

- Projets nécessitant une migration progressive, sans interruption du service ;
- Systèmes existants fortement couplés au SQL via des frameworks ORM ou des composants tiers.
- Environnements critiques où la stabilité métier est prioritaire sur l'optimisation structurelle de la base [25].

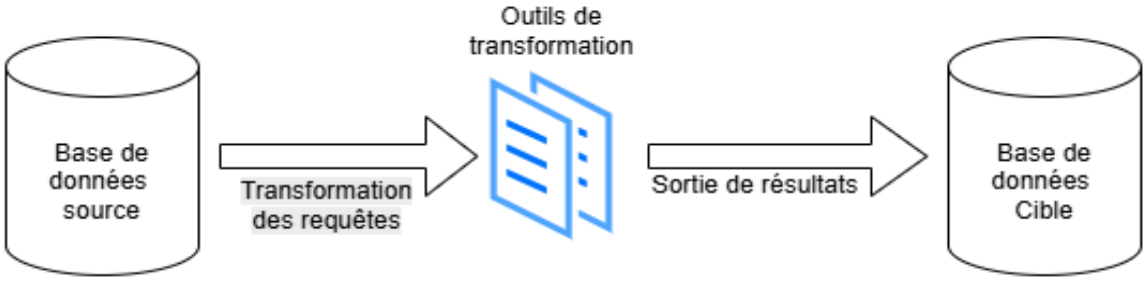

Figure 15. Migration par transformation directe des requête

3.2.4 Technique de migration par ETL

Une autre stratégie couramment utilisée dans le cadre de la migration de données relationnelles vers des bases NoSQL est l'approche ETL (Extract–Transform–Load). Cette méthode repose sur trois grandes étapes fondamentales [40] :

- **Extraction (Extract)** : les données sont extraites de la base source relationnelle après leur évaluation, en général à l'aide de requêtes SQL ou via des connecteurs spécifiques ;
- **Transformation (Transform)** : les données sont ensuite nettoyées, enrichies, réorganisées ou converties dans un format adapté à la structure ;
- **Chargement (Load)** : enfin, les données transformées sont chargées dans la base de destination.

L'approche ETL se distingue par sa capacité à offrir un contrôle poussé sur chaque étape du processus de migration. Contrairement aux méthodes de transformation directe ou aux simples exports de tables, le processus ETL permet de filtrer les données pertinentes, de corriger les erreurs ou les incohérences (données manquantes, doublons, types non conformes) et d'adapter le modèle relationnel à la structure souple des bases NoSQL [40]. Ce contrôle granulaire est particulièrement précieux dans les projets complexes où la qualité, la cohérence et la traçabilité des données sont essentielles. Cependant, cette méthode nécessite une configuration avancée, notamment pour : définir les règles de transformation, gérer les dépendances entre données, orchestrer les flux dans le bon ordre, garantir la reprise sur erreur (cas de panne ou de données incomplètes) et transition vers les cas d'usage. Dans ce contexte, l'approche ETL est particulièrement adaptée aux projets à forte exigence en matière de gouvernance de données ou conformité réglementaire [41]. Par exemple, elle est souvent privilégiée dans les secteurs de la finance, de la santé ou de la logistique, où la moindre erreur de données peut avoir des conséquences majeures. Elle est aussi indiquée dans les cas où les données doivent être normalisées, enrichies ou anonymisées avant d'être utilisées dans un environnement NoSQL. L'approche ETL est fréquemment mise en œuvre à l'aide d'outils spécialisés, comme Studio 3T pour MongoDB, Talend, Apache NiFi ou Apache HOP, ou encore Pentaho Data Integration, qui permettent de concevoir graphiquement les pipelines de transformation tout en assurant la robustesse du processus [41]. Ces limites sont les suivantes : elle nécessite une configuration très avancée, fonctionne en mode batch (migration par lot) et elle n'est pas bidirectionnelle.
Pour conclure, L'approche ETL, bien qu'exigeante en termes de configuration, constitue une solution robuste, fiable et évolutive pour les projets de migration vers des bases NoSQL. Elle combine rigueur, qualité des données et adaptabilité aux modèles non relationnels, ce qui en fait une méthode de choix dans les environnements critiques.

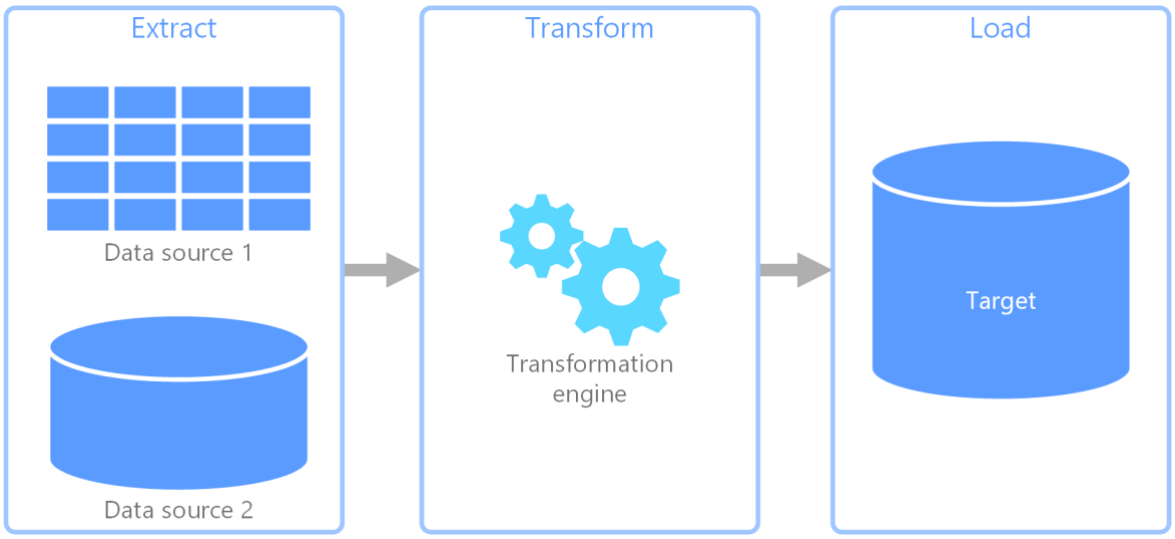

Figure 16. Technique par ETL[11]

3.2.5 Technique de migration par conversion optimisée des schémas

Dans certains projets de migration, notamment ceux où la structure des données relationnelles est déjà relativement stable et cohérente, une conversion complète ou manuelle du schéma vers NoSQL peut être à la fois inutile et coûteuse. Dans ce contexte, des chercheurs ont proposé des solutions basées sur la conversion automatisée et optimisée des schémas, telles que le framework Automatic Mapping Framework (AMF) [42].

Le principe de cette approche repose sur la génération automatique de schémas NoSQL à partir de schémas relationnels existants. L'Automatic Mapping Framework analyse les entités, les relations, les attributs et les contraintes présentes dans la base relationnelle, puis produit un modèle NoSQL équivalent en tenant compte des types de relations (1–1, 1–N, N–N), des clés primaires et étrangères, des hiérarchies ou regroupements logiques de données et des règles de transformation vers des modèles orientés documents ou colonnes [43]. Cette automatisation permet d'optimiser les correspondances structurelles sans intervention manuelle importante, tout en respectant les exigences de performance et de cohérence du système cible. Cette approche présente les avantages suivants [42] :

- Gain de temps considérable dans la phase de modélisation ;
- Réduction des erreurs humaines liées à la conversion manuelle ;
- Simplicité de mise en œuvre, notamment dans des projets à faible complexité structurelle ;

[11] https://learn.microsoft.com/fr-fr/azure/architecture/data-guide/relational-data/etl

- Standardisation du processus, ce qui facilite la reproductibilité de la migration ;
- Compatibilité avec des outils d'analyse de schéma ou des environnements de génération de code.

Ainsi, cette approche est particulièrement adaptée aux projets de migration à grande échelle ou à faible marge de transformation, comme les bases de données de systèmes de gestion de clients, inventaires, les systèmes hérités (legacy) où l'on souhaite minimiser les risques liés à la refonte complète du modèle de données ou dans les environnements devops ou de déploiement continu, où le temps de migration doit être réduit au minimum. De plus, l'Automatic Mapping Framework peut être intégré à des pipelines de migration automatisés, notamment lorsqu'il est combiné avec d'autres outils comme Sqoop (pour le transfert de données) ou des scripts d'optimisation de performance post-migration [42].

En définitive, l'approche de conversion optimisée des schémas, illustrée par des solutions comme **Automatic Mapping Framework**, constitue une alternative efficace et rapide aux méthodes classiques de migration. Elle permet de capitaliser sur la structure existante tout en facilitant l'adoption des bases NoSQL, notamment dans des contextes où la stabilité, la rapidité et la simplicité sont des critères majeurs.

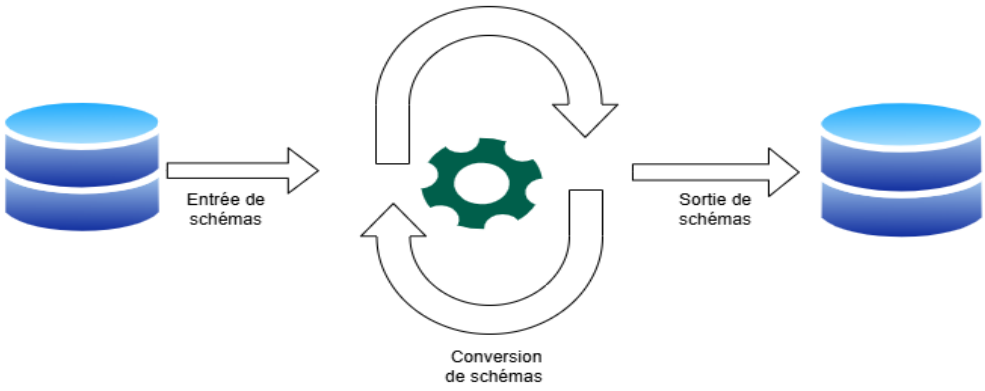

Figure 17. Migration par conversion de schémas

3.2.6 Technique de la migration des schémas et mappage

Parmi les stratégies récentes en matière de migration de bases de données relationnelles vers des systèmes NoSQL, certains chercheurs ont proposé une approche combinée, articulée autour de deux volets complémentaires : la migration des schémas d'un côté, et le mappage des requêtes SQL vers le langage NoSQL de l'autre côté [44]. Cette méthode vise à maintenir une continuité logique et fonctionnelle entre le système source et le système cible, en assurant non seulement la conversion des structures de données (tables, colonnes, clés), mais aussi la

transposition des requêtes existantes, afin que l'application continue à fonctionner sans nécessiter une réécriture complète de la couche d'accès aux données.

Le processus se déroule généralement en deux phases [44] :

- **Migration du schéma** : transformation des entités relationnelles (tables) vers des structures NoSQL compatibles (documents, collections, familles de colonnes), en conservant la sémantique d'origine ;
- **Mappage des requêtes SQL** : les requêtes SQL courantes sont automatiquement traduites ou simulées par des équivalents NoSQL, tout en tenant compte des spécificités du moteur cible (MongoDB, Cassandra, etc.).

Cette technique présente des avantages, comme la préservation de la logique métier, une migration fluide, la réduction du coût de migration, en évitant une reconstruction complète du backend et une grande flexibilité avec possibilité d'adapter progressivement les requêtes selon les besoins d'optimisation. Néanmoins les limites n'y manquent pas, du fait que certaines fonctionnalités SQL (comme les jointures multiples, sous-requêtes imbriquées ou fonctions analytiques) peuvent être difficiles à représenter de manière équivalente dans NoSQL, aussi les syntaxes et capacités des bases NoSQL varient fortement, ce qui peut limiter la portabilité du mappage automatique mais aussi un problème de surcharge potentielle des performances si la couche de mappage est mal optimisée ou trop abstraite.

Cette approche est particulièrement adaptée dans les projets où [44] :

- La couche d'accès aux données repose fortement sur des requêtes SQL codées en dur ou via un ORM ;
- Il est essentiel de réduire le temps d'interruption du service lors de la migration ;
- Les équipes souhaitent conserver le code applicatif existant, en minimisant les changements tout en adoptant une base NoSQL plus adaptée aux nouveaux besoins.

Des frameworks ou outils de transformation semi-automatisés intègrent ce type de logique, notamment dans les migrations de systèmes web, d'ERP ou d'applications SaaS nécessitant une adaptation rapide sans refonte complète [44].

Figure 18. Technique par migration schémas et mappage[12]

3.2.7 Technique de migration avec algorithme automatisé

Enfin, certaines recherches récentes proposent une approche innovante reposant sur l'utilisation d'un algorithme automatisé, implémenté via des outils tels que **VB.NET**, pour faciliter la migration des bases relationnelles vers des bases NoSQL. Cette méthode vise à automatiser l'ensemble du processus de migration, depuis l'analyse du schéma source jusqu'au transfert final des données, en passant par la génération automatique des structures cibles [45].

L'approche algorithmique repose sur la définition d'un flux automatisé intégrant plusieurs étapes clés [45] :

- **Analyse du schéma relationnel existant** : détection des entités, attributs, types, relations, dépendances ;
- **Génération du schéma NoSQL correspondant** : transformation automatique en structures adaptées (documents, collections, colonnes, etc.) ;
- **Transfert des données** : conversion et migration automatique des données en respectant les nouveaux formats.

L'algorithme est conçu pour optimiser les performances, minimiser les interventions manuelles et réduire les risques d'erreurs humaines, souvent présents dans les migrations traditionnelles ou manuelles. L'approche avec un algorithme automatisé présente plusieurs avantages [45] :

✓ Automatisation complète ou partielle du processus, réduisant la charge de travail humaine ;

[12] https://thierrysdynamics365fortalent.com/d365fo-hr-presentation-de-la-gestion-des-donnees

- ✓ Fiabilité renforcée, car les règles de transformation sont codées et exécutées systématiquement ;
- ✓ Adaptabilité aux environnements en temps réel, où la rapidité et la cohérence des migrations sont cruciales ;
- ✓ Optimisation de la structure de données NoSQL générée, notamment en termes de performance et de scalabilité.

Cependant, cette méthode présente certaines limites que sont [45] :

- Dépendance à la qualité de l'algorithme, autrement dit, un algorithme mal conçu peut produire des structures inefficaces ou incohérentes ;
- Manque de flexibilité pour les cas spécifiques ou non standards où une intervention manuelle serait nécessaire ;
- Maintenance technique du logiciel, ce qui signifie que les outils développés sur mesure comme en VB.NET nécessitent un suivi régulier et des compétences spécifiques.

Ce type d'approche est particulièrement pertinent pour [45] :

- Les projets nécessitant une migration rapide et fiable de bases volumineuses ;
- Les systèmes où les ressources humaines sont limitées, et où l'automatisation représente un gain stratégique ;
- Les contextes où l'on souhaite standardiser et répéter le processus de migration sur plusieurs bases similaires.

L'intégration d'un algorithme automatisé permet ainsi de transformer la migration d'une tâche manuelle, technique et sujette aux erreurs, en un processus reproductible, efficace et évolutif, en phase avec les exigences modernes des systèmes distribués.

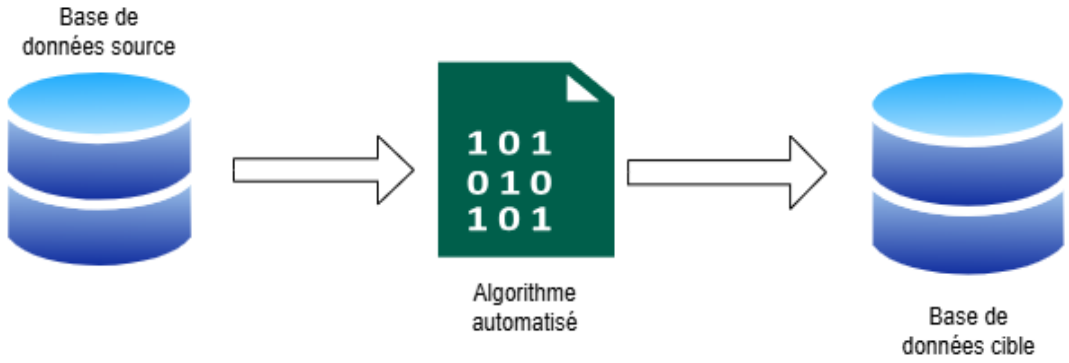

Figure 19. Migration par algorithme automatisé

3.2.8 Technique de migration avec API et scripts

Une autre stratégie flexible et adaptée aux environnements techniques hétérogènes est l'approche par API et scripts personnalisés. Certains investigateurs soulignent que cette méthode permet d'optimiser considérablement les performances de lecture dans les bases de données NoSQL, tout en assurant une gestion fine des relations complexes issues des schémas SQL. Cette approche repose sur le développement de programmes spécifiques, souvent écrits en Python, JavaScript, VB.NET ou Java, qui interagissent avec des interfaces de programmation (API) pour extraire, transformer et insérer les données [38]. Contrairement aux outils de migration tout-en-un ou aux frameworks automatisés, cette méthode offre une flexibilité maximale. Les développeurs conçoivent leurs propres scripts pour interroger les bases relationnelles via SQL, transformer les données selon une logique métier ou structurelle spécifique, utiliser les API des bases NoSQL (comme MongoDB, Firebase, Couchbase) pour insérer les données au format attendu (documents JSON, objets imbriqués, etc.) et traiter dynamiquement les relations complexes (ex. : jointures 1–N ou N–N converties en structures imbriquées ou référencées). Cette approche est souvent qualifiée d'hybride, car elle combine : contrôle manuel du processus de transformation, automatisation ciblée via scripts réutilisables, connexion directe aux systèmes via API pour maximiser l'efficacité [38].

Les avantages de cette technique se présentent comme suit [38] :

- ✓ **Souplesse maximale** : adaptée à des besoins très spécifiques et à des cas non couverts par les outils standards ;
- ✓ **Contrôle complet sur le processus de migration** : possibilité de personnaliser chaque étape selon le contexte applicatif **;**
- ✓ **Bonne gestion des relations complexes** : souvent difficiles à exprimer dans un modèle NoSQL sans traitement intermédiaire ;
- ✓ **Performances élevées :** performe en lecture et écriture car les structures sont optimisées selon le besoin métier.

Toutefois, cette méthode présente aussi des limites [38] :

- Temps de développement important, surtout si la migration concerne de grands volumes ou des structures complexes ;
- Maintenance des scripts car tout changement de schéma ou d'API nécessite une mise à jour manuelle ;

- Dépendance aux compétences techniques des développeurs impliqués dans la conception des scripts ;
- Risque d'erreurs non contrôlées, notamment si la validation des données n'est pas rigoureusement intégrée au script.

Cette approche est particulièrement pertinente dans [38] :

- Les systèmes avec des modèles de données très personnalisés ;
- Les applications nécessitant une migration incrémentale ou en temps réel ;
- Les environnements DevOps où les scripts peuvent être intégrés dans les pipelines de déploiement ;
- Les projets où la logique métier est fortement couplée aux données, nécessitant des adaptations spécifiques.

En définitive, l'approche basée sur l'utilisation d'API et de scripts personnalisés constitue une solution puissante, adaptable et performante pour la migration de bases de données relationnelles vers des systèmes NoSQL. Bien qu'elle demande un investissement initial en temps et en expertise, elle s'avère particulièrement efficace dans les contextes complexes, en constante évolution, où les méthodes standardisées atteignent leurs limites.

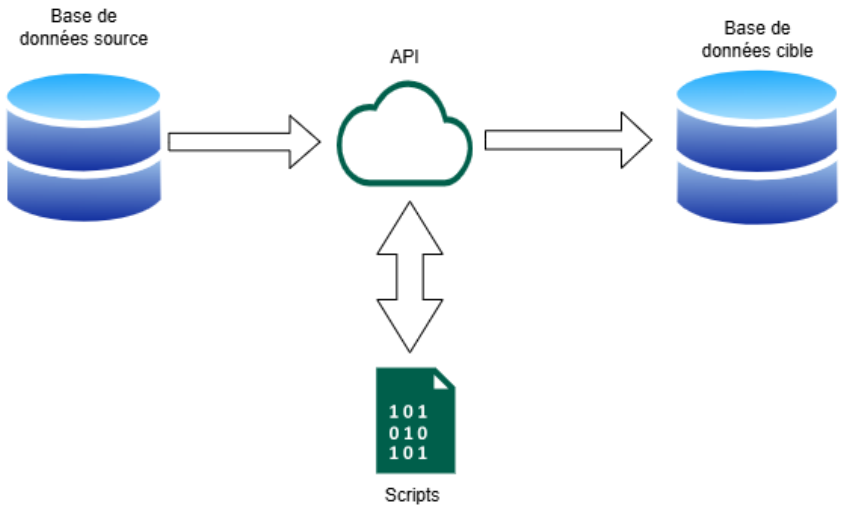

Figure 20. Migration par API et scripts

3.3 Les règles de transformations

Certains travaux proposent des règles de transformation spécifiques pour migrer les schémas relationnels vers des bases NoSQL. Ces règles incluent des transformations de type [46] :

- One-to-one (Modélisation d'une relation 1 :1 en intégrant ou en liant deux entités distinctes) ;
- One-to-many (Modélisation d'une relation 1 : N en imbriquant les entités secondaires ou en maintenant des références explicites) ;
- Many-to-many (Modélisation d'une relation N : N en utilisant des références croisées ou une collection intermédiaire pour stocker les associations) ;
- Spécialisation (Modélisation de l'héritage en regroupant les entités dans une seule collection ou en les séparant en collections distinctes) ;
- Union (Fusion de plusieurs entités similaires en une seule collection avec un attribut différenciant) ;
- Agrégation (Réorganisation des données pour minimiser les requêtes en regroupant les informations souvent consultées ensemble).

Ces modèles permettent de conserver la logique des relations entre les données tout en adaptant les structures aux modèles NoSQL [48].

3.4 Discussions

L'étude des différentes techniques de migration des bases relationnelles vers les systèmes NoSQL met en lumière une diversité d'approches, présentant chacune des avantages spécifiques et des contraintes contextuelles. Cette variété traduit la complexité des processus de migration, qui ne peuvent être réduits à une solution universelle, mais doivent être choisis en fonction des besoins métiers, des ressources techniques, du temps disponible et de la complexité des données à transférer.

Les **frameworks spécialisés** comme NoSQLayer ou **Automatic Mapping Framework** se distinguent par leur capacité à automatiser partiellement ou totalement les étapes de transformation, en facilitant la conversion des structures SQL vers des modèles NoSQL (notamment documentaires). Ils offrent un gain de temps, une standardisation et une réduction des erreurs humaines, mais leur utilisation est souvent limitée à certains moteurs NoSQL comme MongoDB, et leur efficacité peut être réduite face à des schémas complexes ou des règles métier implicites [49]. Les **outils spécialisés** tels qu'Apache Sqoop sont particulièrement adaptés aux contextes Big Data. Leur architecture distribuée et leur intégration avec des systèmes comme Hadoop ou HBase en font des solutions robustes pour gérer de très grands volumes de données, avec une parallélisation des traitements et une synchronisation fiable entre systèmes hétérogènes. Toutefois, leur mise en œuvre suppose une infrastructure adaptée et une

maîtrise de l'écosystème Hadoop [50]. Les **techniques de transformation directe des requêtes** ou de **mappage SQL–NoSQL** permettent une migration rapide, non intrusive et centrée sur la préservation de la logique métier existante. Elles sont recommandées lorsque l'on souhaite minimiser les modifications côté applicatif. Cependant, elles atteignent leurs limites lorsque les requêtes SQL sont très complexes (sous-requêtes, fonctions analytiques) ou lorsque le moteur NoSQL cible une logique très éloignée du modèle relationnel [51]. L'approche **ETL**, quant à elle, se distingue par sa granularité, sa flexibilité et sa capacité à garantir la qualité des données. Elle permet des contrôles précis à chaque étape du processus, ce qui en fait une méthode privilégiée dans les contextes réglementés ou critiques (finance, santé, logistique). Toutefois, elle nécessite un investissement technique important, tant en termes de configuration que de gestion des flux et de reprise sur erreur et parfois, les outils sont commercialisés [39]. Les approches basées sur des **algorithmes automatisés** ou des **scripts personnalisés via API** s'adressent aux environnements complexes ou très spécifiques. Elles permettent un haut niveau de personnalisation, d'optimisation et d'intégration avec les processus existants. Elles sont particulièrement efficaces pour des migrations en temps réel, incrémentales ou nécessitant une adaptation fine à la logique métier. Leur principale limite réside dans leur coût de développement, leur dépendance à des compétences techniques spécifiques, et leur maintien à long terme [51].

En définitive, aucune méthode ne saurait répondre à tous les besoins. Le choix de la technique de migration doit être fondé sur une analyse préalable rigoureuse du système source, des objectifs de migration et des contraintes techniques.

Chapitre 4 : Etude de cas MySQL et MongoDB

4.1 Positionnement

Dans le cadre de ce mémoire, le positionnement méthodologique repose sur des choix stratégiques, tenant compte de la nature des données à migrer, des objectifs de performance et des contraintes temporelles, afin de proposer une solution adaptée et efficace pour la migration de bases de données relationnelles vers NoSQL. En effet, pour ce mémoire, le choix des technologies repose sur des critères stratégiques liés à leur popularité, leurs fonctionnalités, leur adoption dans l'industrie et leur adéquation avec les objectifs de l'étude. Ainsi, **MySQL** a été retenu comme système de gestion de base de données relationnelle (SGBDR) pour son statut de leader dans le domaine des bases de données structurées. Il s'agit d'une solution éprouvée, stable et largement utilisée à travers le monde dans des contextes variés, allant des petites applications aux systèmes complexes. MySQL bénéficie également d'une documentation abondante et d'une vaste communauté de développeurs, ce qui facilite la résolution des problèmes techniques et l'accès aux bonnes pratiques. De plus, son support des propriétés ACID garantit une gestion rigoureuse de l'intégrité des données, un critère essentiel pour comprendre les limites des bases relationnelles face aux volumes de données croissants et non structurés. En parallèle, **MongoDB** a été sélectionné comme base de données NoSQL pour sa popularité et son aptitude à gérer des données semi-structurées et non structurées. Tant que l'une des bases de données NoSQL les plus adoptées au niveau mondial, MongoDB est reconnue pour son évolutivité horizontale, sa haute disponibilité et sa flexibilité dans les environnements distribués. Elle se distingue également par sa simplicité d'utilisation grâce à son modèle basé sur les documents JSON, qui permet une transition naturelle depuis des bases relationnelles, tout en répondant aux besoins modernes en termes de gestion de Big Data et d'applications en temps réel. Le choix de MongoDB est donc justifié par sa capacité à répondre aux exigences des systèmes actuels tout en restant accessible aux utilisateurs novices.

Dans le cadre de ce mémoire, j'ai opté pour une double migration de données : d'une part, de la base de données relationnelle MySQL vers une base NoSQL, en l'occurrence MongoDB, et d'autre part, de MongoDB vers MySQL. Pour réaliser ces migrations bidirectionnelles, j'ai choisi de développer une solution en utilisant des **scripts développés en Python**. Cette approche me permet de garder une maîtrise fine de l'ensemble du processus, depuis l'extraction des données jusqu'à leur transformation et insertion, que ce soit vers un modèle documentaire (MongoDB) ou un modèle relationnel (MySQL).

Cette méthode offre une grande souplesse pour traiter les spécificités des types de données, gérer les relations existantes entre les entités, et adapter dynamiquement la structure des documents ou des tables selon les exigences du système cible. Elle permet également d'aborder de manière concrète les défis liés à la transformation de schémas rigides en modèles flexibles, et inversement, tout en assurant la cohérence et l'intégrité des données. Cette démarche vise donc à concilier rigueur technique, flexibilité et adaptation aux contraintes spécifiques du projet, tout en s'inscrivant dans une logique de compréhension approfondie des différences fondamentales entre les modèles relationnel et NoSQL, dans les deux sens.

Pour conclure, utiliser MySQL, MongoDB et des scripts sur mesure est un choix à la fois pratique et réfléchi pour réussir les migrations bidirectionnelles de données. Cette méthode flexible permet d'automatiser et d'adapter chaque étape du processus selon les besoins des systèmes utilisés, tout en garantissant une transition fluide, efficace et reproductible. Elle aide à mieux comprendre les défis techniques de la migration entre paradigmes de bases de données, et à poser les bases de meilleures décisions architecturales pour l'avenir.

4.2 Architecture générale

4.2.1 Présentation des technologies utilisés

Les technologies qui seront utilisées dans le cadre de notre étude de cas sont :

- **WampServer** : WAMP, acronyme de Apache, MySQL et PHP/Perl/Python, est une pile logicielle pour Windows qui permet le développement et le déploiement d'applications Web. Il nous permet aussi d'utiliser un serveur localement. Chaque lettre de WAMP représente un composant clé de la pile :[13]
  - ✓ **Apache** : Apache HTTP Server c'est le web server composant de WAMP. Il est responsable du traitement des demandes des clients (généralement navigateurs web) et la diffusion de pages Web et d'autres contenus sur Internet ou sur des réseaux locaux ;
  - ✓ **MySQL** : MySQL est un système de gestion de bases de données SQL open source populaire, développé, distribué et pris en charge par Oracle Corporation. MySQL gère un ensemble structuré de données. Une base de données MySQL vous permet d'ajouter, d'accéder et de traiter les données qu'elle contient. MySQL stocke les données dans des tables distinctes. Les structures de la base

[13] https://fr.wikipedia.org/wiki/WampServer

de données sont organisées en fichiers physiques optimisés pour la vitesse. Le modèle logique, avec des objets tels que des bases de données, des tables, des vues, des lignes et des colonnes, offre un environnement de programmation flexible. La partie SQL de « MySQL » signifie « Structured Query Language », le langage standardisé le plus couramment utilisé pour accéder aux bases de données. Le logiciel MySQL utilise la licence GPL (GNU General Public) et est un logiciel open source [33] ;

- ✓ **PHP/Perl/Python :** Ce composant fait référence au langages de programmation la pile WAMP prend en charge le développement de contenu Web dynamique. Ces langages permettent la création de scripts capables de générer des pages Web de manière dynamique, d'interagir avec des bases de données et d'effectuer diverses tâches, server-tâches secondaires[14] ;

- **MongoDB et Compass** : MongoDB est un programme de base de données orienté document. Classé comme produit de base de données NoSQL , MongoDB utilise des documents de type JSON avec des schémas optionnels . Lancé en février 2009 par 10gen (aujourd'hui MongoDB Inc.), il prend en charge des fonctionnalités telles que le sharding , la réplication et les transactions ACID (à partir de la version 4.0). MongoDB Atlas, son service cloud géré, fonctionne sur AWS , Google Cloud Platform et Microsoft Azure . Les versions actuelles sont sous licence Server Side Public License (SSPL). MongoDB est membre de la MACH Alliance[15]. Après l'installation de MongoDB, il est impératif d'installer aussi compass, ce dernier permet de visualiser et de manipuler les données via l'interface.[16] Après leur installation, on les lance via le cmd ;
- **Python** : Python est un langage de programmation très répandu, conçu en 1991 par Guido van Rossum. Il est reconnu pour sa simplicité et sa polyvalence, ce qui le rend adapté à de nombreux domaines. Python est utilisé pour le développement web, notamment côté serveur, la création de logiciels, y compris avec des interfaces graphiques, la résolution de problèmes mathématiques, l'automatisation du système de tâches et le développement d'applications réseaux**.** Le langage Python permet de

[14] https://phoenixnap.fr/glossaire/qu%27est-ce-que-wamp
[15] https://en.wikipedia.org/wiki/MongoDB
[16] https://www.mongodb.com/fr-fr/products/tools/compass

concevoir des applications web fonctionnant sur des serveurs, d'automatiser des flux de travail liés à d'autres logiciels, d'interagir avec des bases de données ou manipuler des fichiers, de migrer des données d'une base à un autre même avec des systèmes différents, de traiter de grandes quantités de données et effectuer des calculs avancés, de réaliser rapidement des prototypes ou développer des applications prêtes à l'emploi [52]. Il présente de nombreux avantages :

- Il est compatible avec de nombreux systèmes (Windows, macOS, Linux, Raspberry Pi…) ;
- Sa syntaxe simple et intuitive le rend accessible aux débutants ;
- Il permet de réduire le nombre de lignes de code nécessaires ;
- C'est un langage interprété, ce qui facilite les tests et le développement rapide ;
- Il prend en charge plusieurs styles de programmation : procédurale, orientée objet et fonctionnelle [34].

Parmi les usages possibles du langage, le point qui va nous intéresser c'est la capacité de python à pouvoir faire migrer des données d'une base à un autre même avec des systèmes différents. En effet, python possède une bibliothèque très riche nous permettant l'extraction, le traitement, la transformation et le transfert de données. Les principales bibliothèques utilisées lors de cette migration sont les bibliothèques standards de Python et les bibliothèques externes à installer via pip. Ces bibliothèques sont les suivantes :

- Datetime pour la gestion des dates (notamment conversion datetime.date vers datetime.datetime) ;
- Json pour la sérialisation et transformation des données ;
- Os pour manipulation des chemins de fichiers ;
- Sys pour la manipulation du chemin d'importation pour aux modules internes ;
- Tkinter pour la création de l'interface graphique utilisateur ;
- Tkinter.simpledialog, tkinter.messagebox pour les boîtes de dialogue de saisie ou d'alerte.
- Pymysql ou mysql.connector qui permet de se connecter à une base MySQL depuis Python ;
- Pymongo qui permet de se connecter à une base MongoDB et de manipuler les documents ;

- **Visual code :** Visual Studio Code (ou VS Code) est un éditeur de code source gratuit**,** léger**,** open-source et multiplateforme **(**Windows, macOS, Linux), développé par Microsoft [17] ;
- **L'interface Tkinter** : Tkinter (Tk interface) est un module intégré à la bibliothèque standard de Python, permettant de créer des interfaces graphiques :
  - Des fenêtres ;
  - Des widgets (boutons, zones de texte, cases à cocher, …) ;
  - Des évènements (clavier, souris, …).

Tkinter est disponible sur Windows et la plupart des systèmes Unix : les interfaces créées avec Tkinter sont donc portables.[18]

4.2.2 Présentation des bases de données Source

Dans le cadre cette étude, nous allons travailler avec deux bases de données : une base de données MySQL nommée « bigcommerce » pour la migration MySQL vers MongoDB et une base de données MongoDB nommée « entreprise » pour la migration MongoDB vers MySQL.

4.2.2.1 Structure de la base de données MySQL

La base de données MySQL « bigcommerce » est structurée comme suit :

```
mysql> show tables;
+-----------------------+
| Tables_in_bigcommerce |
+-----------------------+
| clients               |
| commandes             |
| ligne_commandes       |
| produits              |
+-----------------------+
4 rows in set (0.00 sec)

mysql>
```

Figure 21. Structuration de la base de données MySQL

```
mysql> desc clients;
+------------------+--------------+------+-----+---------+----------------+
| Field            | Type         | Null | Key | Default | Extra          |
+------------------+--------------+------+-----+---------+----------------+
| client_id        | int          | NO   | PRI | NULL    | auto_increment |
| prenom           | varchar(50)  | NO   |     | NULL    |                |
| nom              | varchar(50)  | NO   |     | NULL    |                |
| email            | varchar(100) | NO   | UNI | NULL    |                |
| date_inscription | date         | YES  |     | NULL    |                |
+------------------+--------------+------+-----+---------+----------------+
5 rows in set (0.08 sec)

mysql> desc commandes;
+--------------+---------------+------+-----+---------+----------------+
| Field        | Type          | Null | Key | Default | Extra          |
+--------------+---------------+------+-----+---------+----------------+
| id           | int           | NO   | PRI | NULL    | auto_increment |
| client_id    | int           | YES  | MUL | NULL    |                |
| date_commande | date          | YES  |     | NULL    |                |
| total        | decimal(10,2) | YES  |     | NULL    |                |
+--------------+---------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Figure 22. Structuration des tables clients et commandes

[17] https://fr.wikipedia.org/wiki/Visual_Studio_Code
[18] https://info.blaisepascal.fr/tkinter/

```
mysql> desc ligne_commandes;
+--------------+---------------+------+-----+---------+----------------+
| Field        | Type          | Null | Key | Default | Extra          |
+--------------+---------------+------+-----+---------+----------------+
| id           | int           | NO   | PRI | NULL    | auto_increment |
| commande_id  | int           | YES  | MUL | NULL    |                |
| produit_id   | int           | YES  | MUL | NULL    |                |
| quantite     | int           | YES  |     | NULL    |                |
| prix_unitaire | decimal(10,2) | YES  |     | NULL    |                |
+--------------+---------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)

mysql> desc produits;
+-------------+---------------+------+-----+---------+----------------+
| Field       | Type          | Null | Key | Default | Extra          |
+-------------+---------------+------+-----+---------+----------------+
| id          | int           | NO   | PRI | NULL    | auto_increment |
| nom         | varchar(100)  | YES  |     | NULL    |                |
| description | text          | YES  |     | NULL    |                |
| prix        | decimal(10,2) | YES  |     | NULL    |                |
| stock       | int           | YES  |     | NULL    |                |
+-------------+---------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)

mysql>
```

Figure 23. Structuration des tables ligne_commandes et produits

La base de données est composée de 4 tables que sont :

- La table « clients » qui stocke les informations sur les clients ;
- La table « commandes » qui enregistre les commandes passées par les clients ;
- La table « ligne_commande » qui détaille les articles inclus dans chaque commande, représentant chaque article en tant que "ligne" distincte ;
- La table « produits » qui contient les détails des produits disponibles.

NB : la création de la base de données cible sera fait de l'autre côté portant le même nom.

4.2.2.2 Structure de la base de données MongoDB

La base de données MongoDB « entreprise » est structurée comme suit :

```
test> use entreprise
switched to db entreprise
entreprise> show collections
departements
employes
projets
salles
entreprise>
```

Figure 24. Structuration de la base de données MongoDB

```
entreprise> db.departements.find()
[
  { _id: 1, nom: 'Informatique' },
  { _id: 2, nom: 'Ressources Humaines' },
  { _id: 3, nom: 'Gestion de projet' },
  { _id: 4, nom: 'Marketing' }
]
entreprise> db.employes.find()
[
  {
    _id: ObjectId('6840a70db558f9a1ef760adf'),
    nom: 'Alice',
    poste: 'Développeuse',
    departement_id: 1
  },
  {
    _id: ObjectId('6840a70db558f9a1ef760ae0'),
    nom: 'Karim',
    poste: 'RH',
    departement_id: 2
  },
  {
entreprise>
    nom: 'Nina',
    poste: 'Chef de projet',
    departement_id: 3
  },
  {
    _id: ObjectId('6840a70db558f9a1ef760ae2'),
    nom: 'Ibra',
    poste: 'Support',
    departement_id: 2
  }
]
entreprise>
```

Figure 25. Structuration des collections departements et employes

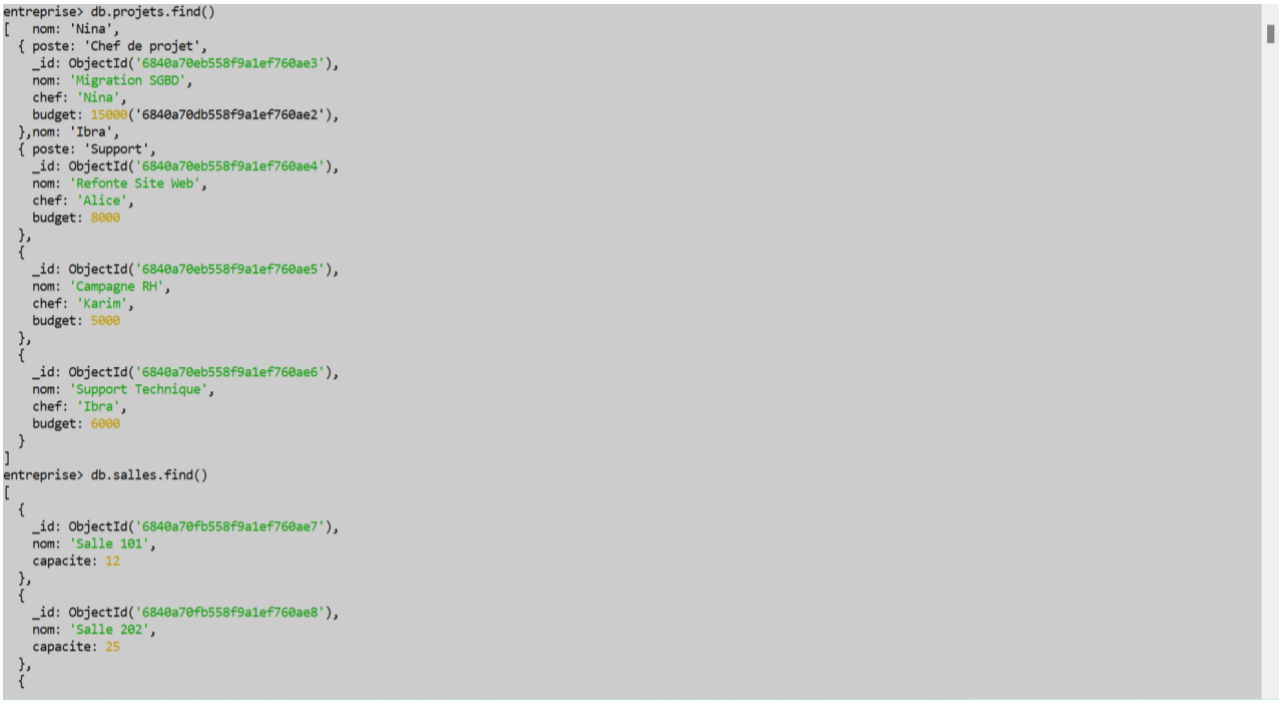

Figure 26. Structuration des collections projets et salles

La base de données « entreprise » est composée de 4 collections :

- La collection « departements » qui stocke les noms des différents départements ;
- La collection « employes » qui stocke les informations sur le personnel ;
- La collection « projets » qui stocke les détails sur les projets en cours ;
- La collection « salles » qui détient les informations du local.

De même, une base de données nommée « entreprise » sera créée dans MySQL.

4.3 Migration

Nous allons faire en premier lieu une migration de la base de données MySQL « bigcommerce » vers MongoDB et en deuxième et dernier lieu, une migration de la base de données MongoDB « entreprise » vers MySQL.

4.3.1 MySQL vers MongoDB

Les fonctionnalités de ce volet se présente comme suit :

4.3.1.1 La page d'accueil

La page propose 4 options que sont : lister les tables MySQL, Migrer une table MySQL vers MongoDB, Migrer toutes les tables MySQL vers MongoDB (migration complète) ou d'effectuer une migration personnalisée, c'est-à-dire, de personnaliser notre migration en choisissant la ou les colonnes à migrer dans une table définie.

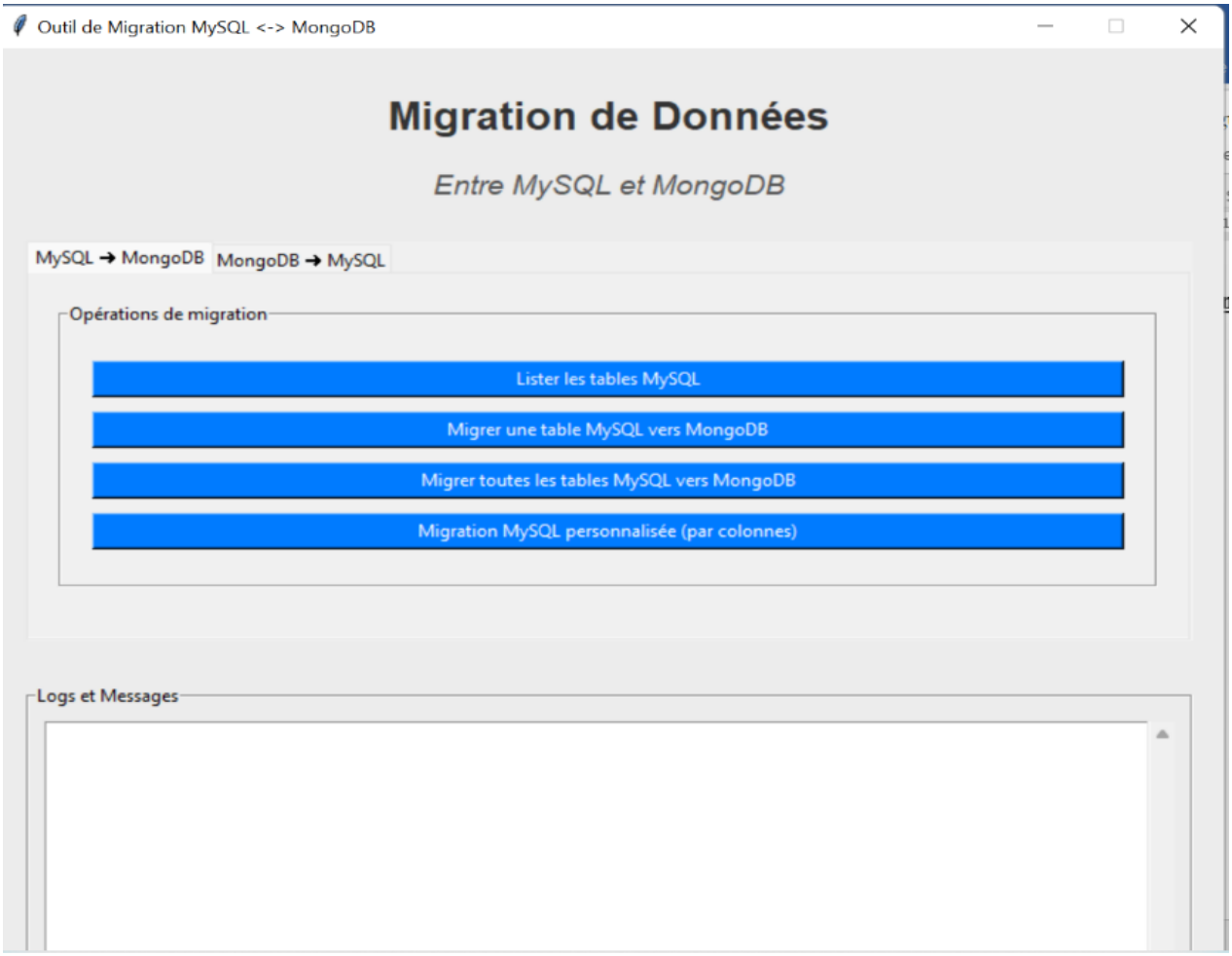

Figure 27. La page d'accueil de l'interface MySQL vers MongoDB

4.3.1.2 Lister les tables MySQL

On aura l'ensemble des tables existantes dans la base de données « bigcommerce », et pour ce cas, on voit qu'on a 4 tables au total.

Une zone de journal, placée en bas de l'interface, affiche en temps réel les messages de connexion, les erreurs éventuelles, ainsi que les informations sur les tables détectées et les opérations exécutées. Cette console de journalisation permet à l'utilisateur de suivre le déroulement du processus de migration et de vérifier la bonne exécution de chaque étape.

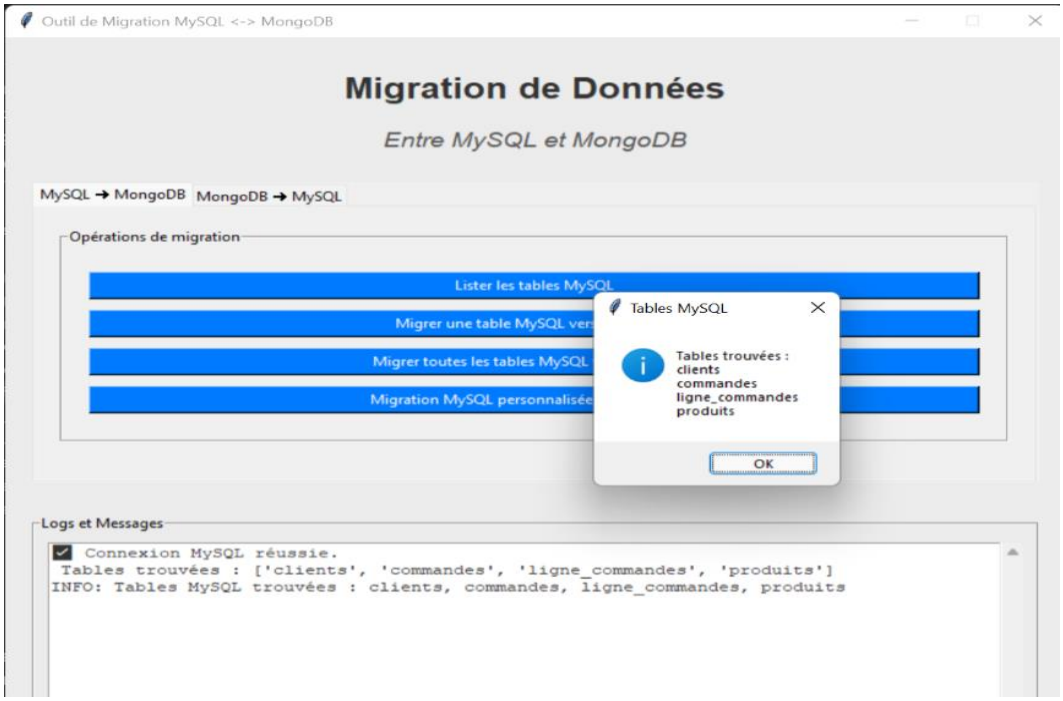

Figure 28. Option de lister les tables MySQL

4.3.1.3 Migrer une table MySQL vers MongoDB

Dans cette option, on a choisi de faire migrer la table « clients » vers MongoDB. L'interface et la zone de journal nous notifie que la migration est réussite.

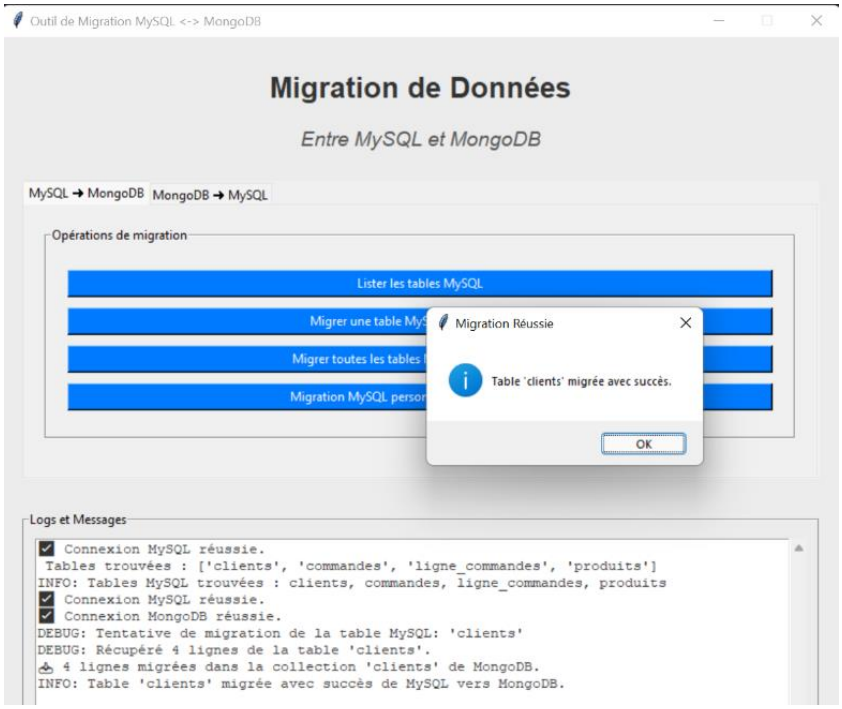

Figure 29. Migration de la table « clients » vers MongoDB

On va vérifier dans la base MongoDB :

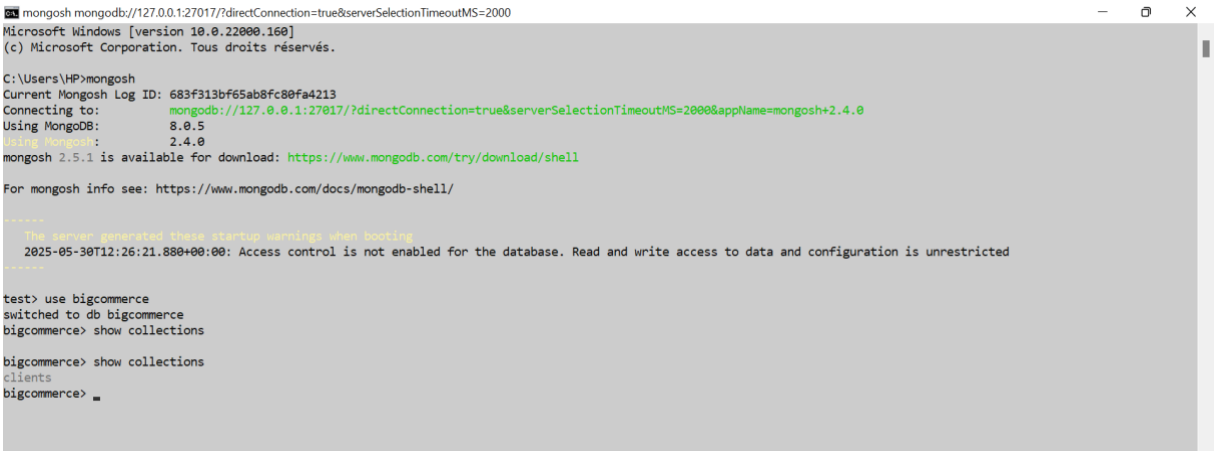

Figure 30. Aperçu de la nouvelle collection « clients » dans MongoDB

On voit que la table « clients » est devenue une collection dans la base de données MongoDB ;

4.3.1.4 Migrer toutes les tables vers MongoDB

Cette fonctionnalité nous permet de faire une migration complète en migrant d'un coup toutes les tables existantes dans la base de données MySQL.

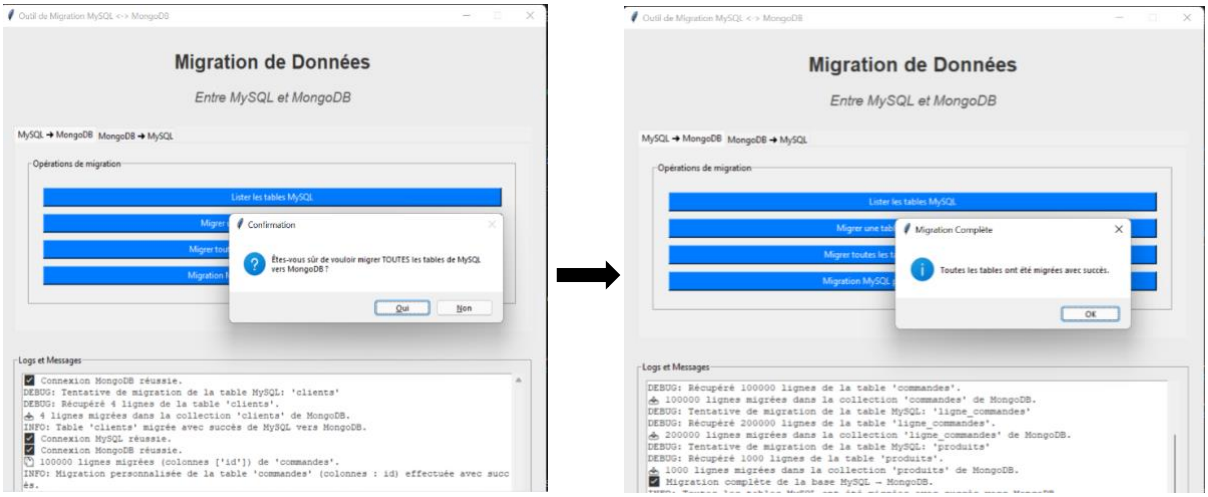

Figure 31. Images d'une migration complète MySQL vers MongoDB

Après avoir cliqué sur l'option, la fenêtre nous demande une confirmation de notre demande, une fois la confirmation, la fonction s'exécute. On verra que les 4 tables ont été migrées avec succès. On va vérifier dans la base MongoDB :

```
bigcommerce> show collections
clients
commandes
ligne_commandes
produits
bigcommerce>
```

Figure 32. Affichage de toutes les tables migrées dans MongoDB

4.3.1.5 Migration MySQL personnalisée

Cette option nous permet de migrer seulement la ou les colonnes choisies pour une migration partielle

Figure 33. Migration personnalisée (colonne).

Dans cette partie, on a migré la colonne « id » de la table « commandes ». On va vérifier dans la base MongoDB :

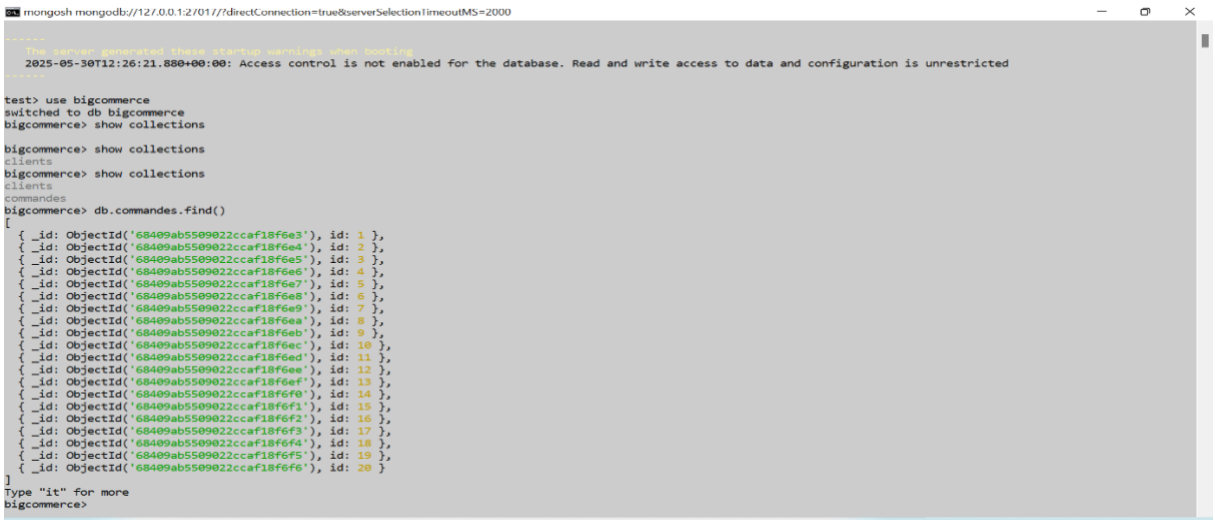

Figure 34. Affichage de la colonne migrée dans MongoDB

La migration de la colonne « id » de la table « commande » a automatiquement créée une nouvelle collection nommée « commandes » qui ne prend un seul champ qu'est le « id » migré.

4.3.2 MongoDB vers MySQL

Les fonctionnalités de ce volet se présente comme suit :

4.3.2.1 La page d'accueil

La page d'accueil de cette partie présente les différentes options pour effectuer une migration MongoDB vers MySQL. Ces options sont au nombre de 4 : Lister les collections MongoDB, Migrer une collection MongoDB vers MySQL, Migrer toutes les collections MongoDB vers MySQL et Migration MongoDB personnalisée (par champs).

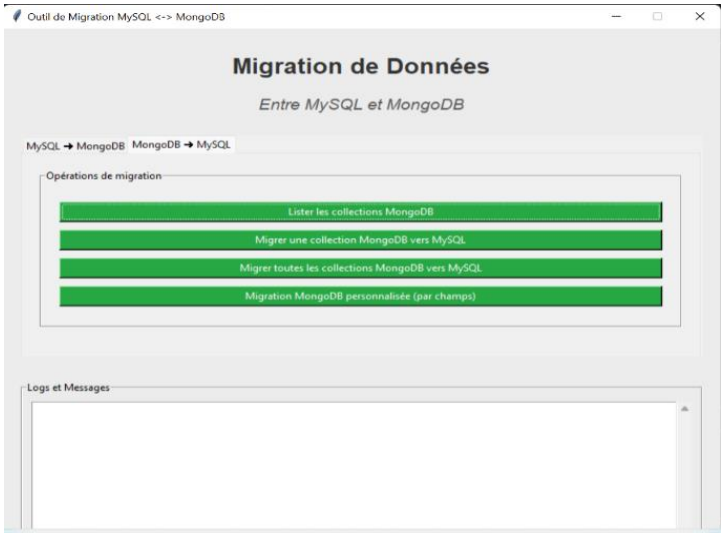

Figure 35. Page d'accueil de la section MongoDB vers MySQL

4.3.2.2 Lister les collections MongoDB

Cette option va lister toutes les collections existantes dans la base de données MongoDB. Pour ce cas, elles sont au nombre de 4 : salles, departements, employes et projets.

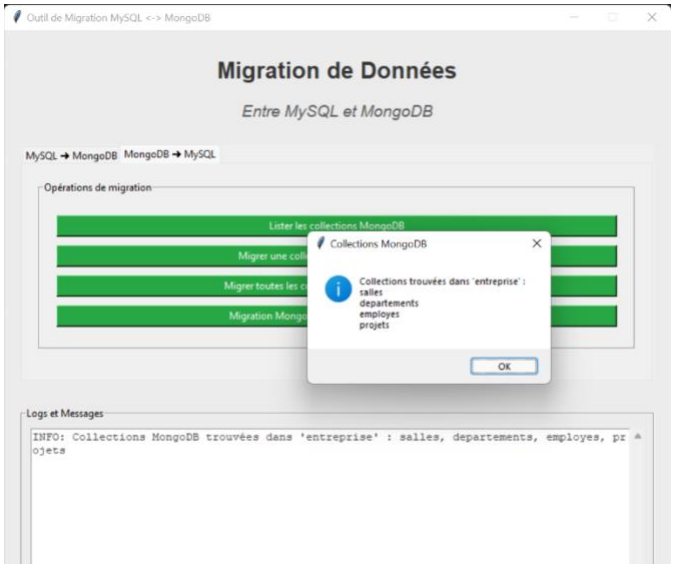

Figure 36. Les collections existantes dans « entreprise » de Mongo

4.3.2.3 Migrer une collection MongoDB vers MySQL

On va faire migrer la collection « salles » vers MySQL. Ceci va créer une nouvelle table dans MySQL nommée « salles ».

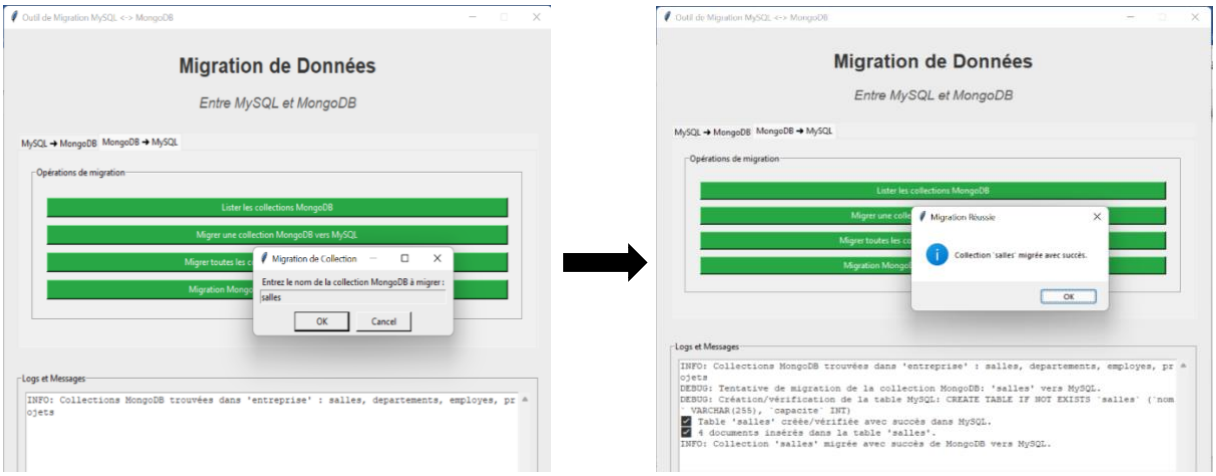

Figure 37. Migration de la collection « salles » vers MySQL

La petite fenêtre et la zone de journal nous notifie que la migration de la collection est réussite. On va vérifier :

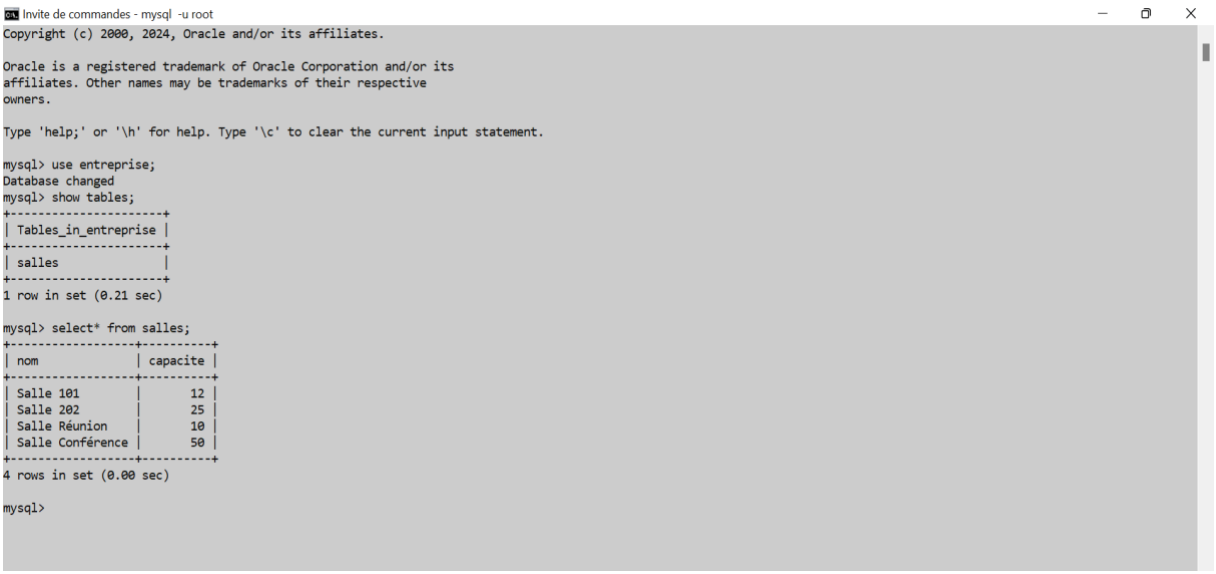

Figure 38. Aperçu de la nouvelle table « salles » dans MySQL

On voit que la nouvelle table est créée et toutes les données de cette table sont visibles.

4.3.2.4 Migrer toutes les collections MongoDB vers MySQL

Cette option nous permet de migrer toutes les collections existantes dans la base de données MySQL, autrement dit, on fera une migration complète MongoDB vers MySQL.

Mise en œuvre :

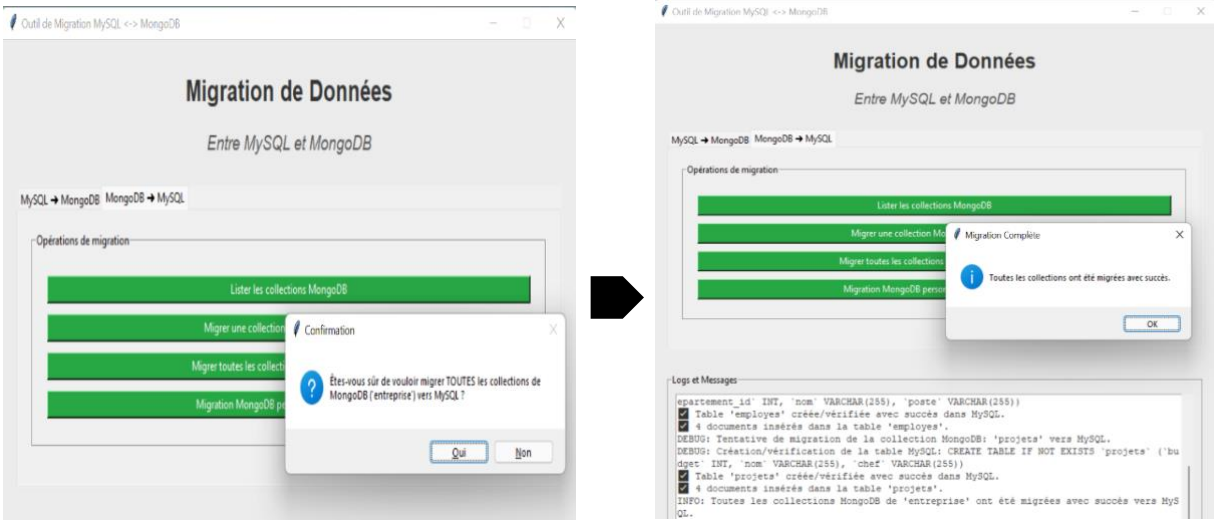

Figure 39. Migration complète MongoDB vers MySQL

Pour ce cas, après lancement de la demande, on va la confirmer et la fonction s'exécute. La zone de notification nous informe que les collections ont été créés et les documents concernés ont été migrés respectivement. On va vérifier dans MySQL :

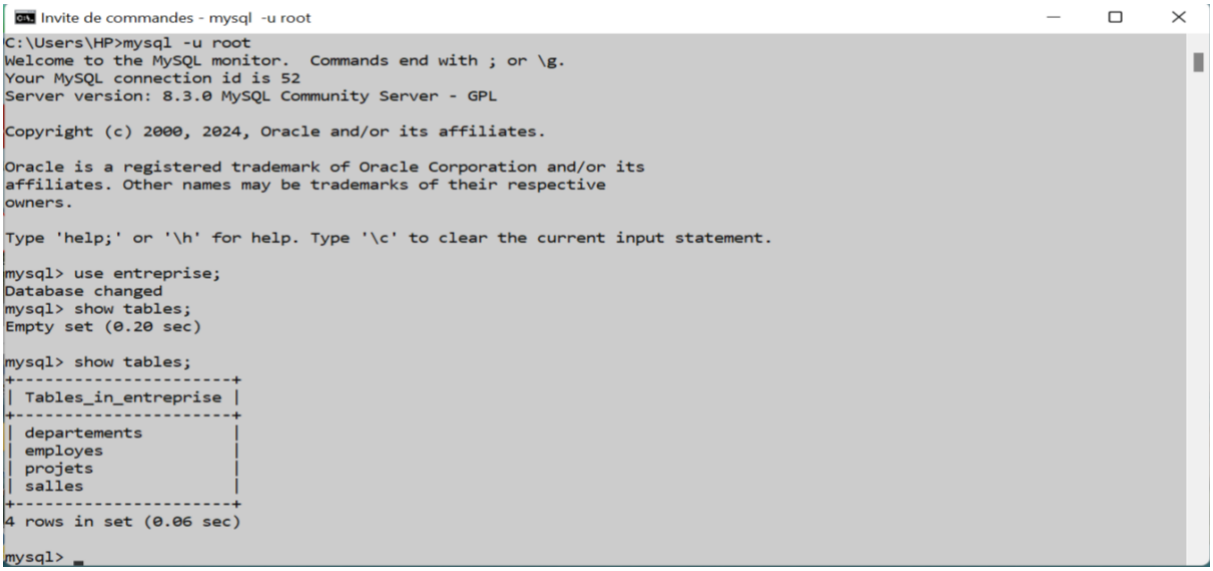

Figure 40. Aperçu des nouvelles tables en provenance de MongoDB

On voit que les collections : departements, employes, projets et salles sont devenues des tables dans la base de données MySQL « entreprise ».

4.3.2.5 Migration MongoDB personnalisée

Dans le cas où on veut faire migrer seulement un champ d'une collection donnée, on choisit cette option. Pour cette étude, nous allons faire migrer les champs « nom » et « budget » de la collection « projets ». Voici la mise en œuvre :

Figure 41. Migration des champs « nom » et « budget » vers MySQL

On va vérifier dans la base de données MySQL :

```
mysql> select*from projets;
+-------------------+--------+
| nom               | budget |
+-------------------+--------+
| Migration SGBD    | 15000  |
| Refonte Site Web  | 8000   |
| Campagne RH       | 5000   |
| Support Technique | 6000   |
+-------------------+--------+
4 rows in set (0.02 sec)
```

Figure 42. La nouvelle table « projets » dans MySQL

La migration est réussite et les champs « nom » et « budget » sont devenus deux colonnes dans la table « projet ».

4.4Discussions

Le module de migration entre **MySQL** et **MongoDB** offre une solution pratique et efficace pour transférer des données entre deux types de bases de données très différents. Il permet de gérer la migration dans les deux sens : de MySQL vers MongoDB, mais aussi de MongoDB vers MySQL, ce qui est très utile selon les besoins du système ou de l'application. Pour la migration de MySQL vers MongoDB, l'outil commence par repérer les tables dans la base MySQL. Cette étape est importante car elle permet de comprendre la structure des données. Ensuite, l'utilisateur peut choisir de :

- Migrer une seule table ;
- Migrer toutes les tables ;
- Ou faire une migration personnalisée, en sélectionnant seulement certaines colonnes.

Cette flexibilité est un vrai avantage, car elle permet de s'adapter à plusieurs cas : que ce soit une petite migration de test, ou une migration plus grande et définitive. La migration de MongoDB vers MySQL, elle, est un peu plus complexe. MongoDB n'a pas de schéma fixe, contrairement à MySQL qui exige une structure bien définie. L'outil est capable d'analyser les types de données dans MongoDB et de transformer les documents JSON en lignes de tables SQL, en respectant au mieux les types comme les dates, les objets, ou les listes. L'utilisateur peut là aussi choisir :

- De migrer une seule collection ;
- Toutes les collections ;
- Ou seulement certains champs spécifiques.

Même si la structure de MongoDB est plus souple, l'outil parvient à convertir les données de manière fiable, en respectant le format de MySQL. Enfin, une fois la migration terminée, il est très important de vérifier que les données ont bien été transférées, sans erreur, et qu'elles gardent leur cohérence dans le nouveau système. Cette étape de validation garantit que les données sont utilisables et fiables. En somme, cet outil de migration permet de passer facilement d'une base relationnelle à une base NoSQL et inversement, tout en offrant plusieurs options de migration. Il est utile pour les entreprises qui veulent changer de technologie, combiner deux types de bases, ou simplifier leur gestion des données.

Conclusion

À l'heure où les volumes de données explosent et où les exigences des systèmes informatiques se complexifient, les choix technologiques en matière de gestion des bases de données s'avèrent cruciaux. Cette mémoire a exploré de manière approfondie les deux paradigmes dominants dans ce domaine : les bases relationnelles (SGBDR), historiquement robustes et cohérentes, et les bases NoSQL, conçues pour répondre aux limites du système relationnel.

L'objectif principal de ce travail était de proposer une méthodologie de migration bidirectionnelle entre MySQL et MongoDB après une étude comparative de ces deux systèmes. À travers une étude théorique rigoureuse, une analyse comparative et une mise en œuvre concrète, nous avons démontré la faisabilité technique et les apports stratégiques d'une telle migration dans un contexte réel.

Les expérimentations réalisées ont permis de valider plusieurs techniques de migration. Chaque méthode a été appliquée selon des critères de flexibilité, de performance, de complexité de mise en œuvre et d'adaptation aux besoins métiers. Ainsi, migrer d'un SGBDR vers une base NoSQL (ou l'inverse) n'est pas un simple transfert de données, mais une transformation en profondeur de la manière dont l'information est structurée, manipulée et exploitée. Ce travail apporte ainsi un guide pratique pour mener cette transition de manière sécurisée, cohérente et adaptée aux enjeux actuels des systèmes d'information. L'outil développé avec Tkinter permet une migration partielle, complète ou personnalisée, tout en respectant la structure, la logique métier et l'intégrité des données. L'outil a montré sa capacité à gérer la migration avec souplesse et fiabilité, tout en assurant la cohérence des données transférées.

Au-delà des aspects techniques, ce travail met en lumière la complémentarité entre les systèmes relationnels et NoSQL. Il ne s'agit plus d'une opposition stricte, mais plutôt d'un choix stratégique dicté par la nature des données, les contraintes de traitement et les objectifs organisationnels. De plus en plus, les systèmes hybrides combinant SQL et NoSQL s'imposent comme des solutions viables pour tirer parti des avantages de chaque modèle.

Ce travail ouvre plusieurs pistes de réflexion et d'amélioration pour l'avenir. D'abord, il serait pertinent d'étendre cette étude à d'autres systèmes NoSQL, notamment les bases orientées graphiques ou colonnes ou clé valeur, afin d'évaluer la compatibilité et les défis spécifiques à ces modèles. Une comparaison plus poussée avec d'autres SGBDR comme PostgreSQL ou Oracle pourrait également enrichir l'analyse. De même, on pourrait développer dans la solution une possibilité de générer des rapports post-migration, incluant des statistiques sur les données

migrées, les transformations appliquées, et d'éventuelles anomalies. Elargir l'outil afin de permettre aux utilisateurs de définir des règles de transformation personnalisées pour des champs ou des collections spécifiques (ex : renommer des colonnes, fusionner des champs, scinder des données, appliquer des fonctions de nettoyage, …). Enfin, dans un contexte où les données sont au cœur de la prise de décision, il serait intéressant d'explorer l'impact de la migration, la sécurité des données, et la conformité réglementaire (ex. RGPD). Ces aspects deviennent cruciaux pour les entreprises qui manipulent des données sensibles et souhaitent évoluer vers des infrastructures plus modernes, tout en garantissant la qualité et la protection des informations. Et pour se faire, l'idée sera d'intégrer des fonctionnalités permettant de masquer ou d'anonymiser certaines données sensibles durant la migration, pour des raisons de conformité.

Ces éléments renforcent l'importance d'aborder la migration non seulement comme une technique de transfert, mais aussi comme un projet stratégique de modernisation des systèmes d'information.

Références

[1] Duvallet, C., Mammeri, Z., & Sadeg, B. (1999). Les SGBD temps réel. Technique et science informatiques, 18(5), 479-517.

[2] Joachim, T. (2022). SGBD Relationnels - Tome 1: Bases de données relationnelles - État de l'art.

[3] Gardarin, G., Gardarin, G., Gardarin, G., Gardarin, G., & Informaticien, F. (1983). Bases de données: les systèmes et leurs langages. Eyrolles.

[4] Gardarin, G. (2003). *Bases de données*. Editions Eyrolles

[5] Scholl, M., & Amann, B. (2001). Bases de Données Relationnelles.

[6] Codd, E. F. (1990). Le modèle relationnel pour la gestion de bases de données : version 2. Addison-Wesley Longman Publishing Co., Inc.

[7] Amann, B., Bidoit, N., Boughanem, M., Bouzeghoub, M., Doucet, A., Gross-Amblard, D., Petit, J.-M., Hacid, M.-S., & Vargas Solar, G. (2013). De la gestion de bases de données à la gestion de grands espaces de données. *Ingénierie des systèmes d'information*, *18*(4), 11-31.

[8] Bekaddour, A., & Bechlaghem, S. A. (2017). *Etude comparative des performances des bases de données SQL et NoSQL MySQL vs MongoDB* (Doctoral dissertation, University of Tlemcen).

[9] Chamberlin, D. D. (1976). Relational data-base management systems. *ACM Computing Surveys (CSUR)*, *8*(1), 43-66.

[10] Post, GV (2009). *Systèmes de gestion de bases de données*. PHI Learning Pvt. Limited.

[11] Rajaram, K., Sharma, P., & Selvakumar, S. (2023, January). DLoader: Migration of Data from SQL to NoSQL Databases. In Proceedings of the International Conference on Cognitive and Intelligent Computing: ICCIC 2021, Volume 2 (pp. 193-204). Singapore: Springer Nature Singapore.

[12] Kim, W. (1979). Systèmes de bases de données relationnelles. ACM Computing Surveys (CSUR), 11 (3), 187-211.

[13] Li, Y., & Manoharan, S. (2013, August). A performance comparison of SQL and NoSQL databases. In 2013 IEEE Pacific Rim conference on communications, computers and signal processing (PACRIM) (pp. 15-19). IEEE.

[14] Astrahan, Blasgen, Chamberlin, Lindsay, Lorie, Mehl, ... & Yost. (1979). System R: A relational data base management system. *Computer*, *12*(5), 42-48.

[15] Choi, Y., Jeon, W., & Yoon, SH (2014). Amélioration des performances des systèmes de bases de données en appliquant NoSQL. *J. Inf. Process. Syst., 10*

[16] Espinasse, B., & Mantha, R. W. (1986). Bases de données relationnelles et connaissances. Université Laval, Faculté des sciences de l'administration.

[17] Desamsetti, H. (2020). Systèmes de gestion de bases de données relationnelles dans les stratégies d'entreprise et d'organisation. *Global Disclosure of Economics and Business*, *9* (2), 151-162.

[18] Christudas, B., et Christudas, B. (2019). MySQL. Modèles d'architecture de microservices pratiques : microservices Java basés sur les événements avec Spring Boot et Spring Cloud, 877-884

[19] Audibert, L. (2007). Base de Données et langage SQL.

[20] Steuert, J., et Goldman, J. (1974, mai). Le système de gestion de données relationnelles : une perspective. Dans *les actes de l'atelier SIGFIDET (aujourd'hui SIGMOD) de l'ACM de 1974 sur la description, l'accès et le contrôle des données* (pp. 295-320).

[21] Piazza, A. G. (2013). NoSQL.

[22] JUGANARU, M. (2022). Bases de données NoSQL

[23] Mihai, G. (2020). Comparison between relational and NoSQL databases. Econ. Appl. Inform, 3, 38-42.

[24] Ali, W., Shafique, M. U., Majeed, M. A., & Raza, A. (2019). Comparison between SQL and NoSQL databases and their relationship with big data analytics. Asian Journal of Research in Computer Science, 4(2), 1-10.

[25] Sharma, V., et Dave, M. (2012). Bases de données SQL et NoSQL. Revue internationale de recherche avancée en informatique et en génie logiciel, 2 (8).

[26] Meier, A., Kaufmann, M., Meier, A., & Kaufmann, M. (2019). Bases de données NoSQL. Bases de données SQL et NoSQL : modèles, langages, options de cohérence et architectures pour la gestion du Big Data, 201-218.

[27] Brouard, F., Bruchez, R., & Soutou, C. (2012). SQL. Pearson Education France.

[28] Strauch, C., Sites, ULS, et Kriha, W. (2011). Bases de données NoSQL. *Notes de cours, Université des médias de Stuttgart*, *20* (24), 79.

[29] Bhandari, H. L., & Chitrakar, R. (2020). Comparison of Data Migration Techniques from SQL Database to NoSQL Database. J Comput Eng Inf Technol 9, 6, 2.

[30] Rigaux, P. (2001). *Cours de bases de données*. CNAM/Médias.

[31] Kunda, D., et Phiri, H. (2017). Étude comparative de NoSQL et des bases de données relationnelles. *Zambia ICT Journal*, *1* (1), 1-4.

[32] Joy, Jumoke & Castro, Harold. (2024). Comparative Analysis of NoSQL vs. SQL Databases for Big Data Analytics.

[33] Vaish, G. (2013). Premiers pas avec NoSQL (p. 9). Packt Pub.

[34] Hainaut, J. L., Henrard, J., Englebert, V., Roland, D., & Hick, J. M. (2018). Database reverse engineering. In *Encyclopedia of database systems* (pp. 967-973). Springer, New York, NY.

[35] Stanier, C. (2012). Introducing nosql into the database curriculum. In 10th International Workshop on the Teaching, Learning and Assessment of Databases (p. 61).

[36] Jumaa, AK (2018). Conversion, migration, traitement et récupération de données sécurisées entre bases de données SQL et Big Data NoSQL. Diyala Journal for Pure Science, 14

[37] Bhandari, H., et Chitrakar, R. (2024). Amélioration d'un algorithme de transformation pour migrer une base de données SQL vers une base de données graphique NoSQL. *Data Science Journal, 23*, 35.

[38] Rocha, L., Vale, F., Cirilo, E., Barbosa, D., & Mourão, F. (2015). A framework for migrating relational datasets to NoSQL. *Procedia Computer Science*, *51*, 2593-2602.

[39] Yassine, F., & Awad, M. A. (2018, November). Migrating from SQL to NOSQL database: Practices and analysis. In 2018 International Conference on Innovations in Information Technology (IIT) (pp. 58-62). IEEE.

[40] Ghotiya, S., Mandal, J., & Kandasamy, S. (2017, novembre). Migration d'une base de données relationnelle vers une base de données NoSQL. Dans la série de conférences IOP : science et ingénierie des matériaux (vol. 263, n° 4, p. 042055). Editions IOP.

[41] Sarmah, SS (2018). Migration de données. Science et technologie, 8 (1), 1-10.

[42] Bouamama, S. (2018). Migration from a relational database to NoSQL. International Journal of Knowledge-Based Organizations, 8, Article 10.4018/ijkbo.2018070104.

[43] Zhao, G., Lin, Q., Li, L., & Li, Z. (2014, November). Schema conversion model of SQL database to NoSQL. In 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (pp. 355-362). IEEE.

[44] Kuderu, N., & Kumari, V. (2016). Conversion d'une base de données relationnelle en NoSQL par migration et mappage de schéma. Revue internationale de génie informatique dans les tendances de la recherche, 3 (9), 506

[45] Mughees, M. (2013). *Data migration from standard SQL TO NoSQL* (Doctoral dissertation, University of Saskatchewan).

[46] Mahmood, A. A. (2018). Automated algorithm for data migration from relational to NoSQL databases. *ALNAHRAIN JOURNAL FOR ENGINEERING SCIENCES*, *21*(1), 60-65.

[47] Alotaibi, O., & Pardede, E. (2019). Transformation du schéma d'une base de données relationnelle (RDB) en bases de données NoSQL. *Données,* 4

[48] Christudas, B., et Christudas, B. (2019). MySQL. Modèles d'architecture de microservices pratiques : microservices Java basés sur les événements avec Spring Boot et Spring Cloud, 877-884.

[49] Bhosale, S. T., Patil, T., & Patil, P. (2015). Sqlite: Light database system. Int. J. Comput. Sci. Mob. Comput, 44(4), 882-885.

[50] Bansal, N., Soni, K., & Sachdeva, S. (2021, décembre). Parcours de migration de bases de données de SGBDR vers des magasins de données NoSQL. Dans la Conférence internationale sur l'analyse des mégadonnées (pp. 159-177). Cham : Springer International Publishing.

[51] Gomez, A., Ouanouki, R., April, A., & Abran, A. (2014). Building an experiment baseline in migration process from SQL database to column oriented No-SQL databases. *J Inform Technol Softw Eng*, *4*(2), 1-7.

[52] Potey, M., Digrase, M., Deshmukh, G., & Nerkar, M. (2015). Database migration from structured database to non-structured database. Int. J. Comput. Appl, 975, 8887

[53] Van Rossum, G., & Drake, F. L. (2009). Python.

Table des matières

Dédicace i
Remerciements ii
Résumé iii
Abstract iv
Sommaire v
Liste des figures vi
Liste des tableaux viii
Sigles et abréviations ix
Introduction générale 1
Chapitre 1 : Généralité sur les SGBDR 4
1.1. Bases de données et les SGBD 5
1.2. Les SGBDR 5
1.2.1 Le modèle relationnel 7
1.2.1.1 Une table, Colonne et ligne 9
1.2.1.2 Clé candidate et clé primaire 9
1.2.1.3 Clé étrangère 10
1.2.1.4 Les contraintes d'intégrités dans les bases de données 10
1.2.2 L'algèbre relationnel 11
1.2.2.1 La projection 12
1.2.2.2 La sélection 12
1.2.2.3 L'union 12
1.2.2.4 L'intersection 12
1.2.2.5 La différence 12
1.2.2.6 Produit cartésien 12
1.2.2.7 La jointure 13
1.2.2.8 La division cartésienne 13
1.2.2.9 Les fonctions d'agrégation 13
1.3 Mode de Fonctionnement (ACID) 14
1.3.1 Atomicité 14
1.3.2 Cohérence 15
1.3.3 Isolation 15

1.3.4 Durabilité 15
1.4 Les Usages des SGBDR 16
1.4.1 Utilisation dans les systèmes de gestion traditionnels 16
1.4.2 Intégration dans les applications web dynamiques 16
1.4.3 Utilisation dans les systèmes décisionnels 16
1.4.4 Support dans les systèmes critiques et distribués 17
1.5 Interrogation des SGBDR (le SQL) 18
1.5.1 Langage de définition de données 19
1.5.2 Le Langage de Manipulation de Données 19
1.5.3 Le langage de contrôle de données 20
1.5.4 Le langage de contrôle de transactions 20
1.6 SGBDR les plus répandus 21
1.7 Points forts 23
1.8 Limites 25
Chapitre 2 : Généralité sur les NoSQL 29
2.1. Présentation 30
2.2 Les principales caractéristiques du NoSQL 30
2.3 Mode Fonctionnement (CAP) 31
2.4 Usages des NoSQL 33
2.5 Types des NoSQL 35
2.5.1 NoSQL orientée colonne 35
2.5.2 Base de données orientée document 36
2.5.3 Base de données de type clé-valeur 37
2.5.4 Base de données de type graphe 38
2.6 Interrogations des NoSQL 40
2.6.1 Bases de données clé–valeur 40
2.6.2 Bases de données orientées colonnes 41
2.6.3 Bases de données orientées documents 41
2.6.4 Bases de données orientées graphe 41
2.6.5 Stratégies d'interrogation distribuée 42
2.7 Points forts 43
2.8 Limites 44

2.9 Comparaison avec SGBDR 44
2.9.1 Comparaison suivant l'usage 44
2.9.2 Suivant le type de données 45
2.9.3 Suivant la volumétrie 46
2.9.4 Suivant la scalabilité 47
2.9.5 Suivant la performance 47
2.9.6 Analyse comparative 48
2.9.7 DISCUSSION 49
Chapitre 3 : Migration SGBDR et NoSQL 51
3.1 La migration de données 52
3.1.1 La Réingénierie des données 52
3.1.1.1 Stratégies de conversions 53
3.2 Formes de migrations 55
3.2.1 Technique de migration par utilisation de framework 55
3.2.2 Technique de migration par outils spécialisés 57
3.2.3 Technique de migration par transformation directe des requêtes 58
3.2.4 Technique de migration par ETL 59
3.2.5 Technique de migration par conversion optimisée des schémas 61
3.2.6 Technique de la migration des schémas et mappage 62
3.2.7 Technique de migration avec algorithme automatisé 64
3.2.8 Technique de migration avec API et scripts 66
3.3 Les règles de transformations 67
3.4 Discussions 68
Chapitre 4 : Etude de cas MySQL et MongoDB 70
4.1 Positionnement 71
4.2 Architecture générale 72
4.2.1 Présentation des technologies utilisés 72
4.2.2 Présentation des bases de données Source 75
4.2.2.1 Structure de la base de données MySQL 75
4.2.2.2 Structure de la base de données MongoDB 76
4.3 Migration 77
4.3.1 MySQL vers MongoDB 77

4.3.1.1 La page d'accueil 77
4.3.1.2 Lister les tables MySQL 78
4.3.1.3 Migrer une table MySQL vers MongoDB 79
4.3.1.4 Migrer toutes les tables vers MongoDB 80
4.3.1.5 Migration MySQL personnalisée 81
4.3.2 MongoDB vers MySQL 82
4.3.2.1 La page d'accueil 82
4.3.2.2 Lister les collections MongoDB 83
4.3.2.3 Migrer une collection MongoDB vers MySQL 83
4.3.2.4 Migrer toutes les collections MongoDB vers MySQL 84
4.3.2.5 Migration MongoDB personnalisée 85
4.4 Discussions 87
Conclusion 88
Références 90
Table des matières 94
Annexe 1 : Code de connexion MySQL avec Python 98
Annexe 2 : Code de connexion MongoDB avec Python 99
Annexe 3 : Code de la fonction pour lister les tables MySQL 100
Annexe 4 : Code pour la migration d'une table vers MongoDB 101
Annexe 5 : Code pour la migration de toutes les tables vers MongoDB 103
Annexe 6 : Code pour la migration personnalisée 104
Annexe 7 : Code du fichier main 104
Annexe 8 : Code de connexion pour la deuxième phase de migration 105
Annexe 9 : Code pour lister les collections 106
Annexe 10 : Code de migration d'une collection vers MySQL 107
Annexe 11 : Code de migration de toutes les collections vers MySQL 108
Annexe 12 : Code pour une migration personnalisée vers MySQL 109
Annexe 14 : Code de l'interface 110

Annexe 1 : Code de connexion MySQL avec Python

```
# Importation du module mysql.connector qui permet d'établir une connexion entre Python et une base de données MySQL
Import mysql.connector

# Définition d'une fonction nommée mysql_connexion, destinée à établir la connexion à la base de données MySQL
def mysql_connexion():
    Try :
        # Tentative de connexion à la base MySQL avec les paramètres spécifiés
        conn = mysql.connector.connect(
            host="localhost",
            user="root",
            password="",
            database="bigcommerce"
        )
        # Message affiché en cas de succès de la connexion
        print("✅ Connexion MySQL réussie.")
        return conn
    except mysql.connector.Error as err:
        # Gestion des erreurs en cas d'échec de connexion : affichage du message d'erreur
        print(f"❌ Erreur MySQL : {err}")
        return None
```

Annexe 2 : Code de connexion MongoDB avec Python

```
# Importation de la classe MongoClient depuis le module pymongo, utilisée pour se connecter à un serveur MongoDB
from pymongo import MongoClient

# Définition d'une fonction nommée mongodb_connexion, destinée à établir une connexion avec MongoDB
def mongodb_connexion():
    try:
        # Création d'un client MongoDB en se connectant au serveur local (localhost) sur le port 27017 (port par défaut de MongoDB)
        client = MongoClient("mongodb://localhost:27017/")

            # Sélection de la base de données nommée "bigcommerce" (elle sera créée automatiquement si elle n'existe pas)
        db = client["bigcommerce"]

        # Message affiché en cas de succès de la connexion
        print("✅ Connexion MongoDB réussie.")

        return db  # Retourne la base de données pour l'utiliser dans d'autres opérations
    except Exception as e:
        # En cas d'erreur, capture de l'exception et affichage d'un message d'erreur
        print(f"❌ Erreur MongoDB : {e}")
        return None  # Retourne None si la connexion échoue
```

Annexe 3 : Code de la fonction pour lister les tables MySQL

```
# Définition d'une fonction list_tables qui prend en paramètre une connexion active à une base de données MySQL
def list_tables(mysql_connexion):
    # Création d'un curseur pour exécuter des requêtes SQL via la connexion fournie
    cursor = mysql_connexion.cursor()

     # Exécution de la requête SQL pour lister toutes les tables présentes dans la base de données
    cursor.execute("SHOW TABLES")

    # Récupération du résultat de la requête sous forme de liste de noms de tables
    tables = [row[0] for row in cursor.fetchall()]

    # Fermeture du curseur pour libérer les ressources
    cursor.close()

    # Affichage des tables trouvées dans la console
    print(f" Tables trouvées : {tables}")

    # Retourne la liste des noms de tables
    return tables
```

Annexe 4 : Code pour la migration d'une table vers MongoDB

```
# Importation des modules nécessaires pour la gestion des types de données
from datetime import datetime, date        # Pour traiter les dates et datetimes
from decimal import Decimal                # Pour convertir les types numériques décimaux (ex : DECIMAL de MySQL)
# Définition d'une fonction permettant de migrer une seule table MySQL vers une collection MongoDB
def MysqlToMongo_onetable(mysql_connexion, mongodb_connexion, table_name):
    print(f"DEBUG: Tentative de migration de la table MySQL: '{table_name}'")
    try:
        # Création d'un curseur avec un résultat au format dictionnaire (chaque ligne = dict clé:valeur)
        cursor = mysql_connexion.cursor(dictionary=True)
        # Exécution de la requête pour récupérer toutes les lignes de la table spécifiée
        cursor.execute(f"SELECT * FROM {table_name}")
        rows = cursor.fetchall()          # Récupération de toutes les lignes
        cursor.close()                    # Fermeture du curseur après exécution
        print(f"DEBUG: Récupéré {len(rows)} lignes de la table '{table_name}'.")

        # Vérifie si la table est vide
        if not rows:
            print(f" ⚠ Table '{table_name}' vide ou aucune donnée récupérée.")
            return

        processed_rows = []               # Liste qui contiendra les documents transformés pour MongoDB

        for row in rows:                  # Parcours de chaque ligne (dictionnaire) récupérée depuis MySQL
            processed_doc = {}
            for key, value in row.items():   # Pour chaque colonne et valeur de la ligne
                # Si la valeur est une date (mais pas datetime), on la convertit en datetime
                if isinstance(value, date) and not isinstance(value, datetime):
                    processed_doc[key] = datetime(value.year, value.month, value.day)
                # Si la valeur est de type DECIMAL, on la convertit en float (car MongoDB ne gère pas Decimal)
```

```
            elif isinstance(value, Decimal):
                processed_doc[key] = float(value)
            # Si la valeur est binaire (BLOB), on essaie de la décoder en texte UTF-8
            elif isinstance(value, bytes):
                try:
                    processed_doc[key] = value.decode('utf-8')
                except UnicodeDecodeError:
                        processed_doc[key] = value.hex()  # Si ce n'est pas décodable, on le stocke en hexadécimal
            else:
                processed_doc[key] = value  # Sinon, on conserve la valeur telle quelle

        processed_rows.append(processed_doc)  # Ajout du document formaté à la liste

    # Si des documents ont été créés, insertion en masse dans la collection MongoDB
    if processed_rows:
        result = mongodb_connexion[table_name].insert_many(processed_rows)
        print(f"📥 {len(result.inserted_ids)} lignes migrées dans la collection '{table_name}' de MongoDB.")
    else:
        print(f"⚠ Aucune ligne à insérer après traitement pour la table '{table_name}'.")

# Gestion d'erreur générale pour capturer toute exception pendant le processus
except Exception as e:
    print(f"❌ ERREUR lors de la migration de la table '{table_name}' vers MongoDB: {e}")
```

Annexe 5 : Code pour la migration de toutes les tables vers MongoDB

```
# Importation de la fonction list_tables depuis le module lister_tables
# Cette fonction permet de récupérer toutes les tables de la base MySQL
from lister_tables import list_tables

# Importation de la fonction MysqlToMongo_onetable depuis le module migration_table_complete
# Cette fonction assure la migration des données d'une seule table vers MongoDB
from migration_table_complete import MysqlToMongo_onetable

# Définition de la fonction permettant de migrer toutes les tables d'une base MySQL vers MongoDB
def Mysqltomongo_all_tables(mysql_connexion, mongo_db):
    # Récupération de la liste des tables présentes dans la base MySQL
    tables = list_tables(mysql_connexion)

    # Parcours de chaque table pour lancer une migration individuelle
    for table in tables:
        MysqlToMongo_onetable(mysql_connexion, mongo_db, table)

    # Message de confirmation une fois que toutes les tables ont été traitées
    print("✅ Migration complète de la base MySQL → MongoDB.")
```

Annexe 6 : Code pour la migration personnalisée

```
# Définition d'une fonction permettant de migrer uniquement certaines colonnes d'une table MySQL vers MongoDB
def mysqltomongo_personnalised(mysql_connexion, mongo_db, table_name, columns):
    # Création d’un curseur avec des résultats sous forme de dictionnaire (clé: nom de colonne, valeur: donnée)
    cursor = mysql_connexion.cursor(dictionary=True)

    # Construction dynamique de la liste des colonnes à sélectionner (ex : "id, nom, email")
    cols_str = ", ".join(columns)

    # Exécution de la requête SQL avec uniquement les colonnes spécifiées
    cursor.execute(f"SELECT {cols_str} FROM {table_name}")

    # Récupération des lignes sélectionnées
    rows = cursor.fetchall()
    cursor.close()  # Fermeture du curseur pour libérer les ressources

    # Si des données ont été récupérées
    if rows:
        # Insertion de toutes les lignes dans MongoDB sous forme de documents
        mongo_db[table_name].insert_many(rows)
        print(f"🎁 {len(rows)} lignes migrées (colonnes {columns}) de '{table_name}'.")
    else:
        # Si aucune donnée n’est récupérée, affichage d’un message d’erreur ou d’avertissement
        print(f"⚠ Table '{table_name}' vide ou colonnes incorrectes.")
```

Annexe 7 : Code du fichier main

```
# Importation des fonctions nécessaires depuis les modules définis précédemment
from connexion_mysql import mysql_connexion            # Connexion à la base MySQL
from connexion_mongo import mongodb_connexion          # Connexion à la base MongoDB
from lister_tables import list_tables                  # Fonction pour lister les tables MySQL
from migration_table_complete import MysqlToMongo_onetable      # Migration d'une table unique
from migration_base_complete import Mysqltomongo_all_tables     # Migration complète de la base
from migration_table_partielle import mysqltomongo_personnalised # Migration partielle (colonnes spécifiques)

# Fonction principale du programme
def main():
    print("🔗 Connexion aux bases de données...")
    mysql_conn = mysql_connexion()      # Connexion à MySQL
    mongo_db = mongodb_connexion()      # Connexion à MongoDB

    # Si l'une des connexions échoue, on arrête le programme
    if mysql_conn is None or mongo_db is None:
        print("❌ Impossible d'établir une connexion à une des bases.")
        return
    # Boucle principale pour afficher le menu de migration
    while True:
        print("\n=== MENU MIGRATION MySQL ➜ MongoDB ===")
        print("1. Lister les tables MySQL")
        print("2. Migrer une seule table")
        print("3. Migrer toutes les tables")
        print("4. Migrer des colonnes spécifiques d'une table")
        print("5. Quitter")
….
```

Annexe 8 : Code de connexion pour la deuxième phase de migration

```
# Fichier de configuration centralisé pour les connexions aux bases de données

# URI de connexion à MongoDB
MONGO_URI = "mongodb://localhost:27017"
# Ce paramètre définit l'adresse du serveur MongoDB, ici en local sur le port par défaut 27017

# Paramètres de connexion MySQL regroupés dans un dictionnaire
MYSQL_CONFIG = {
    "host": "localhost",       # Adresse du serveur MySQL (local dans ce cas)
    "user": "root",            # Nom d'utilisateur MySQL
    "password": "",            # Mot de passe (vide ici, à adapter en production)
    "database": "entreprise"   # Nom de la base de données MySQL cible
}
```

Annexe 9 : Code pour lister les collections

```
# Définition d'une fonction pour lister les collections dans une base MongoDB
def lister_les_collections(mongo_db):
    # Récupère la liste des noms de collections existantes dans la base MongoDB
    collections = mongo_db.list_collection_names()
    # Vérifie si des collections sont présentes
    if collections:
        print("🎁 Collections trouvées dans la base 'parfumerie' :")
        # Affiche le nom de chaque collection
        for collection in collections:
            print(f" - {collection}")
    else:
        # Si aucune collection n'existe, affiche un message d'avertissement
        print("❌ Aucune collection trouvée dans la base 'parfumerie'.")
```

Annexe 10 : Code de migration d'une collection vers MySQL

```
from datetime import datetime
import json  # Pour convertir les objets complexes (listes, dictionnaires) en chaînes JSON

# Fonction pour migrer une collection MongoDB vers une table MySQL
def migrate_onecollection_to_mysql(mongo_db, mysql_conn, collection_name):
    print(f"DEBUG: Tentative de migration de la collection MongoDB: '{collection_name}' vers MySQL.")
    # Accès à la collection MongoDB et récupération de tous les documents
    collection = mongo_db[collection_name]
    data = list(collection.find())

    if not data:
        print(f"⚠ Collection '{collection_name}' vide ou aucune donnée trouvée.")
        return
    cursor = mysql_conn.cursor()
     table_name = collection_name  # Le nom de la collection devient le nom de la table MySQL

    # Étape 1 : Création du schéma MySQL
    # Détection automatique des clés (colonnes) présentes dans tous les documents
    all_keys = set()
    for doc in data:
        all_keys.update(doc.keys())
    all_keys.discard('_id')  # L'identifiant MongoDB (_id) est ignoré

    if not all_keys:
        print(f"AVERTISSEMENT: Aucune clé valide trouvée dans la collection '{.
....
```

Annexe 11 : Code de migration de toutes les collections vers MySQL

```
# Importation de la fonction de migration d'une seule collection depuis le module dédié
from migration_One_Collection import migrate_onecollection_to_mysql

# Fonction permettant de migrer automatiquement toutes les collections MongoDB vers MySQL
def migrate_all_collections_to_mysql(mongo_db, mysql_conn):
    # Récupère la liste des collections disponibles dans la base MongoDB
    collections = mongo_db.list_collection_names()

    # Parcourt chaque collection et appelle la fonction de migration spécifique
    for collection_name in collections:
        migrate_onecollection_to_mysql(mongo_db, mysql_conn, collection_name)
```

Annexe 12 : Code pour une migration personnalisée vers MySQL

```
# Définition d'une fonction pour migrer uniquement certains champs d'une collection MongoDB vers MySQL
def migration_partielle(mongo_db, mysql_conn, collection_name, selected_fields):
    # Accès à la collection MongoDB spécifiée
    collection = mongo_db[collection_name]

    #Récupère uniquement les champs sélectionnés pour chaque document (projection MongoDB)
    documents = collection.find({}, {field: 1 for field in selected_fields})
    cursor = mysql_conn.cursor()
    table_name = collection_name
    # Création de la table dans MySQL avec uniquement les champs sélectionnés, de type TEXT
    champs_sql = ", ".join([f"{field} TEXT" for field in selected_fields])
        create_table_query = f"CREATE TABLE IF NOT EXISTS {table_name} ({champs_sql})"
    cursor.execute(create_table_query)
    # Parcours des documents MongoDB récupérés
    for doc in documents:
        # Récupère les valeurs des champs choisis, converties en chaîne de caractères
        values = [str(doc.get(field, "")) for field in selected_fields]
# Placeholders SQL pour insertion
        placeholders = ", ".join(["%s"] * len(selected_fields))
        insert_query = f"INSERT INTO {table_name} ({', '.join(selected_fields)}) VALUES ({placeholders})"
        cursor.execute(insert_query, values)  # Exécution de la requête d'insertion
    # Validation des insertions dans la base MySQL
    mysql_conn.commit()
     print(f"✅ Migration partielle de la collection '{collection_name}' terminée avec les champs : {', '.join(selected_fields)}.")
```

Annexe 14 : Code de l'interface

```
import tkinter as tk
from tkinter import messagebox, simpledialog, scrolledtext, ttk  # Import des widgets Tkinter et ttk pour interface graphique
import sys
import os

# Ajout du chemin absolu du sous-dossier 'MongoToSql' pour permettre l'import des modules situés dans ce dossier
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), 'MongoToSql')))
# Imports pour la migration MySQL vers MongoDB (modules situés à la racine du projet)
# Gestion connexion MySQL
from connexion_mysql import mysql_connexion
# Gestion connexion MongoDB
from connexion_mongo import mongodb_connexion
# Liste des tables MySQL
from lister_tables import list_tables
# Migration complète d'une table MySQL vers MongoDB
from migration_table_complete import MysqlToMongo_onetable
# Migration complète de toutes les tables MySQL vers MongoDB
from migration_base_complete import Mysqltomongo_all_tables
# Migration partielle (sélection de champs) MySQL vers MongoDB
from migration_table_partielle import mysqltomongo_personnalised
# Imports pour la migration MongoDB vers MySQL (modules situés dans le sous-dossier 'MongoToSql')
from MongoToSql.connexion import connect_mongo, connect_mysql
from MongoToSql.lister_collections import lister_les_collections
from MongoToSql.migration_One_Collection import migrate_onecollection_to_mysql
from MongoToSql.migration_All_Collections import migrate_all_collections_to_mysql
from MongoToSql.migration_collection_partielle import migration_partielle
….
```