Analyse et évaluation de structures orientées document

Paola Gomez Barreto

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par
Paola Andrea GOMEZ BARRETO

Thèse dirigée par **Claudia RONCANCIO** et
co-dirigée par **Rubby CASALLAS**

préparée au sein du
Laboratoire d'Informatique de Grenoble (LIG)
dans **l'école doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Analyse et évaluation de structures orientées document

Thèse soutenue publiquement le **13 décembre 2018**,
devant le jury composé de :

Madame CLAUDIA RONCANCIO
Professeur, Grenoble INP - Ensimag, Directrice de thèse
Madame RUBBY CASALLAS
Professeur, Universidad de los Andes - Colombie, Co-Directrice de thèse
Monsieur FRANCK RAVAT
Professeur, Université Toulouse I Capitole, Rapporteur
Monsieur PHILIPPE ROOSE
Maître de conférences (HDR), Université de Pau, Rapporteur
Madame LAURENCE DUCHIEN
Professeur, Université de Lille 1, Examinateur
Monsieur PATRICK REIGNIER
Professeur, Grenoble INP - Ensimag, Président

COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

ÉCOLE DOCTORALE MSTII

Description de complète de l'école doctorale

T H È S E

pour obtenir le titre de

docteur en sciences

de la Communauté Université Grenoble Alpes

Mention : INFORMATIQUE

Présentée et soutenue par

Paola Andrea GOMEZ BARRETO

Analyse et évaluation de structures orientées document

Thèse dirigée par Claudia RONCANCIO et
co-dirigée par Rubby CASALLAS préparée au Laboratoire d'Informatique de Grenoble (LIG)

soutenue le 13 décembre 2018

Jury :

Directrice :	Claudia RONCANCIO	Grenoble INP - Ensimag	LIG
Co-Directrice :	Rubby CASALLAS	Universidad de los Andes	TICSw
Rapporteurs :	Franck RAVAT	Université Toulouse I Capitole	IRIT
	Philippe ROOSE	Université de Pau	LIUPPA
Examinateur :	Laurence DUCHIEN	Université de Lille 1	CRIStAL
Président :	Patrick REIGNIER	Grenoble INP - Ensimag	LIG

Remerciements

Tout d'abord, mille mercis à mes directrices Claudia Roncancio et Rubby Casallas, femmes et scientifiques brillantes. Elles ont cru en moi. Elles m'ont guidée avec beaucoup de patience dans mes moments de faiblesse. Elles m'ont donné la force pour faire aboutir cette thèse.

Un merci très spécial à Germán Vega, Cyril Labbé, Nadine Mandran et Jaime Chavarriaga pour toutes les discussions académiques qui m'ont permis d'organiser mes idées et de clarifier certains points. Merci Germán d'être si attentif à mes progrès. Un grand merci à Jean-Pierre Giraudin pour les échanges autour de la rédaction de ce manuscrit, j'ai beaucoup appris sur la langue française.

Merci à Philippe Roose et Franck Ravat pour l'évaluation de cette thèse et pour des commentaires enrichissants. Merci à Laurence Duchien et Patrick Reignier d'avoir accepté d'être les examinateurs de ce travail.

Merci beaucoup à toute l'équipe SIGMA, permanents et doctorants ! Ces quatre années d'échanges académiques et personnels ont été très riches.

Je n'aurais pas pu arriver jusqu'à la fin de cette thèse sans le soutien de ma famille et de mes amis, pour qui ni la distance ni les longues périodes d'absence pour me consacrer à cette thèse n'ont été un obstacle.

Merci à mes amis en Colombie, toujours présents et prêts à me donner des mots d'encouragement et de joie. Merci à Angie, Anyela, Cedulf, David M., Lili, Magda et Marcela H., votre présence a été vraiment précieuse. À tous les amis que cette thèse m'a apportés, merci d'avoir été là du début à la fin de ce long processus : Alfonso, Alicia, Amira, Carole, Colin, Coralie, Cristina, Fatemeh, Luisa, Marcelo, Mélanie, Mario, Nico H., Nico E., Paula, Pierre, Ornela, Raquel, Sandra, Tien, Valen et mes chers voisins.

Merci à toute ma famille en Colombie pour m'avoir apporté le soutien et l'amour inconditionnel malgré la distance. Merci à la famille Coronado-Martinez, qui m'a adoptée et m'a accueillie avec tout son amour dès le premier instant. Merci Yola pour tout ton amour. Merci à la famille Perrier-Dunand de m'avoir accueillie si chaleureusement et de m'avoir encouragée jusqu'au bout.

Emmanuel, cariño, te trouver sur ce chemin a été un très beau cadeau auquel je ne m'attendais pas. Merci pour ton soutien, tes encouragements et ta patience.

À ma maman et à ma sœur Lorena : merci pour votre amour, votre soutien de tous les jours, pour vos paroles d'encouragement et pour le silence également lorsque j'en ai eu besoin.

Table des matières

Table des figures

Liste des tableaux

Table of Acronyms

AMISS	*Algorithme de Modélisation de Schémas Semi-structurés*
AQL	*Asterix Query Language*
CQL	*Cassandra Query Language*
DSL	*Domain Specific Language*
FLWOR	*For-Let-Where-Order by-Return*
JSON	*JavaScript Object Notation*
LIG	*Laboratoire d'Informatiue de Grenoble*
MDA	*Model driven architecture*
MDE	*Model Driven Engineering*
EMF	*Eclipse Modeling Framework*

"C'est impossible, dit la fierté.
C'est risqué, dit l'expérience.
C'est sans issue, dit la raison.
Essayons, murmure le coeur."

William Arthur Ward

1

Introduction

.

1.1 A propos des systèmes NoSQL

Le paysage de la gestion des données est devenu extrêmement riche et complexe. La grande variété des besoins des systèmes d'information actuels, a conduit à l'émergence de nombreuses solutions de gestion de données hétérogènes [1] [2] [3]. Les SGBD relationnels restent largement utilisés mais côtoient un grand nombre d'autres systèmes dits *NoSQL* [4]. Ces systèmes *NoSQL* répondent à des besoins divers tels que des systèmes de stockage fiable pour des masses de données, des mécanismes performants pour leur analyse, des structures permettant de représenter des données complexes avec souplesse.

Le modèle de données est l'une des caractéristiques importantes d'un SGBD et de tout type de système de gestion de données. Les systèmes NoSQL couvrent une gamme de solutions dont le modèle de données n'est pas relationnel. Leurs modèles de données sont généralement classés dans quatre grandes familles : clé-valeur, orientés colonnes, graphes et orientés documents [5]. Cette classification est courante mais il n'y a pas de définition de modèle de données unique pour chaque famille. Les efforts de formalisation sont peu nombreux et récents [2] [6] [7] [8]. Les concepts manipulés dans ces modèles ne sont cependant pas nouveaux et reprennent des aspects d'autres modèles tels que le relationnel et ses extensions qui ne sont pas en première forme normale [9] [10], des valeurs complexes [11] et des modèles orientés objet [12].

Les systèmes NoSQL permettent beaucoup de liberté pour représenter les données. En général ces systèmes ne manipulent pas de notion de schéma de base de données. On parle de systèmes *Schemaless* car il n'est pas nécessaire de définir un schéma de données avant de créer la base. De plus, la structure des données peut être différente même si elles sont regroupées au sein d'une même collection ou table. La vérification des types est très basique. Par exemple, dans les systèmes clé-valeur, le type de la valeur est inconnu et peut différer complètement d'une entrée à une autre.

Le typage faible des données et la grande flexibilité autorisée par certains systèmes ont un impact sur les langages de requêtes supportés. Il n'y a bien sûr pas de langage de requêtes unique et dans la plupart des systèmes, le langage n'est pas déclaratif. Les systèmes clé-valeur, par exemple, offrent des interfaces très simples (i.e., put, get) alors que d'autres, comme les systèmes orientés documents tel que MongoDB [13], requièrent une programmation impérative. Les systèmes orientés graphes constituent néanmoins une catégorie à part concernant le langage de requêtes. Le système Neo4J [14] notamment, offre le langage Cypher qui permet de naviguer dans les graphes en découvrant sa structure et les données.

En ce qui concerne le support transactionnel et la vérification de la cohérence des données, les

systèmes NoSQL sont également très hétérogènes. Certains offrent un support complet avec des transactions ACID ou divers protocoles pour maintenir la cohérence de données dupliquées. D'autres systèmes, au contraire, fonctionnent plutôt comme des systèmes de stockage tolérant aux pannes mais sans plus de fonctions transactionnelles. La responsabilité est reléguée à la couche application.
En effet, pour la plupart des systèmes NoSQL, le développeur gagne en liberté mais sa responsabilité est aussi fortement accrue. C'est le cas notamment pour la gestion des données semi-structurées qui fait l'objet de ces travaux.

1.2 Structuration des données : problématique et objectifs de la thèse

Dans le cadre de cette thèse nous nous sommes intéressés aux systèmes d'information utilisant des solutions NoSQL. Nous avons d'abord étudié la question sous l'angle de la migration d'applications utilisant un SGBD relationnel vers un système NoSQL. Nous avons étudié trois catégories de systèmes : orienté graphes Neo4J [14], orienté colonnes Cassandra [15] et orienté documents MongoDB [13]. Bien que ces systèmes soient très différents, dans les trois cas nous avons expérimenté la flexibilité de la structuration des données et l'impact de l'absence de schéma de base de données et d'une vérification automatique de contraintes d'intégrité. L'architecture ANSI/SPARC n'est pas adoptée dans ces systèmes ; cela impacte les caractéristiques des logiciels qui les utilisent mais aussi la démarche de développement elle-même. L'absence de schéma permet un développement initial rapide et l'utilisation libre des structures de données. Néanmoins cette souplesse a un impact sur divers aspects tels que la performance de l'interrogation des données, leur cohérence, la facilité d'utilisation des bases et plus largement la mise au point et la maintenabilité du logiciel [16].

Étant donnée la variété des systèmes NoSQL, nous avons focalisé les travaux de cette thèse sur des systèmes orientés documents. Nous avons particulièrement travaillé avec le système MongoDB qui figure parmi les plus utilisés actuellement [17]. Ce système s'appuie sur le format JSON [1] pour la représentation des données.

MongoDB gère des collections de documents JSON. Ce format permet une grande flexibilité de structuration des données. Les collections peuvent stocker des documents simples analogues aux n-uplets d'une relation avec des valeurs atomiques, ces collections peuvent aussi inclure des documents aux structures complexes qui *imbriquent* récursivement d'autres documents, voire

1. Plus précisément il s'agit de BSON (format binaire)

des tableaux de documents, ou qui *référencent* des documents stockés dans d'autres collections. Le système permet de combiner librement tous ces constructeurs et ouvre ainsi la possibilité de stocker des données représentées de manière très complexe. La complexité est autant plus accrue que les structures au sein d'une même collection peuvent être hétérogènes et que les données peuvent être incomplètes, fragmentées ou redondantes.

Les possibilités de structuration des données sont donc extrêmement nombreuses même lorsque conceptuellement les données sont simples. L'absence d'une séparation du niveau conceptuel, et l'absence d'un schéma, ne facilitent pas l'analyse de la structuration des données. De plus, les avantages et les inconvénients d'une structuration dépendent de plusieurs facteurs qui peuvent évoluer si les besoins métier et l'utilisation de la base évoluent.

Le choix de la structuration est néanmoins important et critique parce qu'il va conditionner l'empreinte mémoire, les programmes écrits pour traiter les données qui seront plus ou moins faciles à utiliser, à lire, à maintenir selon la structure choisie. Bien sûr les performances ne seront pas les mêmes selon les cas. Le choix de la meilleure structuration n'aura probablement pas de réponse unique définitive. Il serait alors intéressant d'envisager des alternatives temporelles voire parallèles.

Ainsi, nos travaux se placent dans le contexte des bases orientées documents. L'objectif de cette thèse est, d'une part, de clarifier l'impact de la structuration de données sur la qualité des applications et, d'autre part, de proposer des aides aux développeurs pour faire des choix éclairés au sujet de la structuration des données.

1.3 Contributions de cette thèse

Nos recherches visent à contribuer à la qualité des applications qui utilisent des systèmes NoSQL orientés documents. La qualité est une notion assez ouverte qui concerne plusieurs aspects dont certains liés à la structuration des données. Nos travaux visent à aider l'utilisateur à comprendre, évaluer, choisir, maintenir, et, potentiellement, faire évoluer la structuration des données d'une base orientée documents. Cette compréhension de la structure peut également aider à évaluer ou mettre en évidence les impacts sur la programmation des requêtes et la gestion des applications. Nos propositions facilitent ces tâches en permettant à l'utilisateur de mener une analyse structurelle à faible coût et qui passe à l'échelle en terme de nombre de structures analysées.

Principes directeurs
Nos propositions sont un ensemble d'outils [2], au sens large, permettant d'aider le développeur à mener une phase de conception et d'analyse malgré l'utilisation d'un système de gestion de données sans schéma (*Schemaless*). Ainsi, nous avons fait le choix d'expliciter le type des structures de données utilisées dans la base. Pour cela nous proposons le format AJSchéma, simple et lisible. Sans jouer le rôle de schéma dans la base, un AJSchéma permettra néanmoins de mettre en évidence les types des données et ainsi d'évaluer et de raisonner sur les caractéristiques d'une structure. L'analyse de ses caractéristiques croisées aux préférences des utilisateurs permettra de mettre en avant les structures à privilégier ou à écarter. Nos contributions incluent les éléments donnés ci-dessous. Elles sont coordonnées dans une approche appelé SCORUS composée d'algorithmes, de métriques et d'un prototype.

Approche globale de SCORUS
L'approche SCORUS distingue des phases de modélisation, de génération, d'évaluation et d'analyse de structurations de données. Chaque phase peut être aussi effectuée indépendamment à des fins d'analyse et de réglage. Nous proposons une approche où le développeur modélise ses données à l'aide d'UML. Ce modèle est traité par SCORUS qui génère des variantes de structures possibles pour représenter les données dans une base type MongoDB. Ces variantes sont produites sous forme d'AJSchémas. Ensuite, dans une phase d'évaluation, SCORUS évalue automatiquement des métriques pour chaque AJSchéma. Ces métriques constituent des indicateurs objectifs des caractéristiques des structures. Une phase d'analyse complète l'approche. Elle s'appuie sur les métriques et les critères des utilisateurs qui reflètent les priorités applicatives. Cette analyse constitue une analyse multi-critères qui aidera à faire ressortir les AJSchémas les plus favorables ou ceux qui ne le sont pas.

Algorithmes de génération automatique d'alternatives de structuration
Pour réaliser la phase de génération, nous proposons à partir d'un modèle UML de produire un ensemble de variantes d'AJSchémas pouvant le représenter. Cette génération automatique permet à l'utilisateur d'apprécier facilement de multiples choix de structures dont l'analyse peut être poursuivie ou non. Pour ce générateur nous avons suivi une approche issue du domaine des lignes de produits logiciels. Il s'agit de l'approche par modèles de caractéristiques qui permet d'exprimer la variabilité entre diverses alternatives d'un produit. [18] [19] Dans notre travail, nous cherchons à produire diverses alternatives d'AJSchéma. Nous utilisons les éléments du modèle UML et les possibilités de structuration orientés documents afin d'identifier les alternatives possibles. Cela nécessite de formaliser les variations et points communs à travers d'un modèle de caractéristiques. La stratégie des modèles de caractéristiques permet de définir la variabilité des structures possibles d'une manière compacte, et de contrôler l'explosion des possibilités qui peuvent survenir. Nous proposons l'algorithme *AMISS* qui pour un modèle UML crée un modèle de caractéristiques

2. Outil ici ressemble des propositions conceptuelles, des directives de structuration et d'évaluations ainsi que des algorithmes et des interfaces.

contenant les variations des structurations. Pour chaque alternative de structuration fournie par ce modèle nous dérivons les structures correspondantes à donner en sortie. Ces structures sont créées par les algorithmes *ADApt* et *ADJusT* et peuvent être visualisées dans divers formats dont les AJSchémas.

Métriques structurelles
Afin de contribuer à l'analyse des structures, nous souhaitons proposer des éléments de comparaison objectifs. Pour cela, nous proposons la définition formelle d'un ensemble de métriques structurelles qui reflètent la complexité d'un AJSchéma. Ces métriques permettent notamment de mettre en avant divers choix d'imbrication et redondance de données qui peuvent jouer un rôle important dans la base et pour les applications. Ces métriques structurelles facilitent l'analyse et la comparaison de schémas sans forcément créer les bases de données correspondantes.

Prototype ScorusTool
Le prototype mis en œuvre dans cette thèse porte sur la génération d'AJSchémas et l'évaluation de métriques. Le prototype est composé notamment de trois composants : un *générateur d'alternatives*, un *dérivateur de schéma* et *évaluateur de métriques*. Le *Générateur d'alternatives* récupère un modèle UML et crée le modèle de caractéristiques à l'aide de l'implémentation de l'algorithme *AMISS*. Le *Dérivateur de schéma* implémente les algorithmes *ADJusT* et *ADApt* afin de traduire une alternative de structuration dans divers formats, AJSchéma ou autre. L'*Evaluateur de métriques* évalue les métriques structurelles d'un schéma. Pour cela, nous implémentons des algorithmes de parcours d'arbre afin de calculer les métriques structurelles selon les formules proposées dans cette thèse. Ces trois composants peuvent être utilisés indépendamment ou enchaînés dans un processus. Nous fournissons également une interface graphique pour la présentation des résultats et l'interaction avec l'utilisateur.

Expérimentations
Dans le cadre de cette thèse, nous avons réalisé deux expérimentations conséquentes avec le système MongoDB : la première avec un rôle d'étude exploratoire pour avoir une compréhension fine du système et du rôle de développeurs, la deuxième pour valider avec des bases opérationnelles les structures de données préconisées suite à une analyse faite avec SCORUS. Les bases et les programmes développés pour ces expérimentations sont mis à disposition pour utilisation par d'autres chercheurs.

Publications
Ces travaux ont donné lieu aux publications suivantes [20], [21],[22] :

— *Data schema does matter, even in NoSQL systems !*, Paola Gomez, Rubby Casallas et Claudia Roncancio, Actes de IEEE Tenth International Conference on Research Challenges in Information Science (RCIS), 2016.

— *Métriques structurelles pour l'analyse de bases orientées documents*, Paola Gomez, Rubby Casallas et Claudia Roncancio, Actes du XXXVIème Congrès INFORSID, 2018.

— *Towards quality analysis for document oriented bases*. Paola Gomez, Rubby Casallas et Claudia Roncancio, Actes de International Conference on Conceptual Modeling (ER), 2018. Springer.

L'article présenté à INFORSID 2018 a été sélectionné pour le numéro spécial de la revue ISI consacré aux meilleurs articles de la conférence. Une version longue de cet article est en cours de préparation.

1.4 Organisation du manuscrit

Ce document est organisé en neuf chapitres dont cette introduction constitue le premier. Les deux chapitres suivants sont consacrés à l'état de l'art. Nous commençons, dans le chapitre 2, par présenter les systèmes orientés document sur lesquels nous avons travaillés en s'intéressant particulièrement aux aspects liés à la modélisation des données. Ensuite, dans le chapitre 3, nous abordons les aspects liés à la qualité et plus précisément aux métriques. Dans cette thèse nous avons fait le choix de proposer des métriques structurelles pour les bases de documents, ce qui n'est quasiment pas couvert dans les travaux existants. Dans le chapitre 3 nous présentons les propositions actuelles sur les métriques dans les bases de données et en génie logiciel.

Dans l'optique d'approfondir l'impact de la structuration des données, nous avons mené une étude expérimentale avec plusieurs bases de documents MongoDB. Cette étude est présentée dans le chapitre 4. L'objectif était de comparer les performances selon la structuration des données stockées. Dans un premier temps, pour un même jeu de données, nous avons créé plusieurs bases de documents avec des structurations représentatives variées. Nous décrivons les expériences faites en utilisant ces alternatives. Nous avons procédé à leur évaluation systématique en examinant les chemins d'accès habituels selon une complexité croissante. Dans ce chapitre, nous discutons les résultats des expériences et nous concluons en mettant en évidence l'impact de certains patrons de structuration et les enjeux liés au choix de structuration.

Le chapitre 5 aborde le cœur des propositions de cette thèse. Il est consacré aux principes et principaux choix de SCORUS. Il donne un aperçu des trois phases : la génération d'un ensemble d'alternatives de structuration, leur évaluation avec un ensemble des métriques structurelles proposées dans cette thèse et l'analyse des alternatives. Ce chapitre introduit également notre représentation AJSchéma qui vise à faciliter le raisonnement sur les choix de structuration des données dans les systèmes orientés documents. Dans un troisième temps, nous revenons sur les phases de modélisation et de génération préconisées dans SCORUS en s'appuyant sur la

représentation des données avec un modèle UML. Nous introduisons les éléments de traduction d'un modèle UML vers des structurations pour une base orientée documents, plus précisément, vers des types de données pouvant être exprimés sous la forme d'AJSchéma.

Ensuite, le chapitre 6 concerne la présentation du générateur d'AJSchémas. Le début de ce chapitre est consacré aux fondamentaux de la variabilité et de la stratégie de modèles de caractéristiques que nous utilisons pour le générateur. Dans un second temps, nous décrivons comment, à partir d'un modèle UML, nous modélisons les variantes des structures orientées documents. Ensuite nous présentons notre algorithme *AMISS* qui permet de créer un modèle de caractéristiques représentant un ensemble d'alternatives de structuration. Dans un troisième temps, nous introduisons les algorithmes de dérivation. Nous en proposons deux, l'algorithme *ADJusT* qui permet de dériver l'AJSchéma d'une variante, et l'algorithme *ADApt* qui permet de dériver une représentation arborescente particulièrement utile dans la phase d'évaluation.

La suite s'oriente vers nos contributions en matière d'évaluation des structures. Nous proposons un ensemble de métriques structurelles qui reflètent la complexité et les caractéristiques principales d'un AJSchéma. Le chapitre 7 est consacré à la définition formelle de ces métriques structurelles. Nous introduisons également un exemple permettant d'illustrer chacune de nos métriques. Nous distinguons plusieurs groupes de métriques permettant d'identifier des aspects structurels clés tels que l'existence, la profondeur d'imbrication, la largeur d'imbrication, le référencement, la redondance et la taille. Ce chapitre se termine par nos conclusions et une discussion en relation avec les propositions faites dans d'autres domaines.

La validation de cette thèse est présentée dans le chapitre 8. Nous montrons un scénario de validation de l'approche SCORUS qui met en œuvre toutes les phases évoquées précédemment en incluant la phase d'analyse des AJSchémas. Cette phase n'a pas été développée dans cette thèse, mais, afin de valider le concept, nous avons réalisé une analyse à l'aide d'une fonction multi-critères pour le cas d'étude. La validation est complétée par une expérimentation grande échelle avec des bases MongoDB.

La mise en œuvre du prototype *ScorusTool* est présentée dans le chapitre 8.

Le chapitre 9 présente nos conclusions et les perspectives de recherche que nous proposons.

"Impose ta chance, serre ton bonheur et va vers ton risque.
À te regarder, ils s'habitueront."

René Char

2

Modèles de données orientés document

.

2.1 Introduction aux modèles de données des systèmes NoSQL

Le volume de données et leur diversité sont tellement importants aujourd'hui que les systèmes de stockage et de gestion de données ont émergé sous de très nombreuses formes. La Figure 2.1 donne un panorama des plates-formes de gestion de données disponibles. Cela couvre des systèmes de gestion de fichiers aux systèmes fortement structurés (comme les systèmes relationnels) en passant par une variété de systèmes stockant des données semi-structurées. Cette diversité a conduit aussi à une large utilisation des formats d'échange de données. Concernant les données, certaines sont non structurées sémantiquement comme le son. Les données semi-structurées qui nous intéressent dans cette thèse ont une certaine forme de structuration qui n'est généralement pas explicite et qui n'impose pas de typage strict.

Les travaux autour des données semi-structurées ont été mis à l'ordre du jour par les systèmes NoSQL mais ils ne sont pas récents. Parmi les précurseurs nous pouvons citer les propositions sur SGML [23], le travaux de l'ODMG [24] et le format d'échange OEM [25]. Les difficultés pour proposer des langages de manipulation de données ont déjà été traitées dans ces contextes [26]. Notamment, les complications induites par la connaissance partielle ou la méconnaissance de la structure des données, du type des attributs et le caractère, potentiellement, très imbriqué et cyclique des données font partie d'études. Ces travaux notent également déjà les besoins d'opérations de restructuration des données ainsi que de facilités de navigation au sein des structures (i.e *path expressions*). Comme nous le verrons par la suite, ces aspects jouent un rôle important dans les travaux sur les bases orientées documents.

Dans le cadre de cette thèse nous nous intéressons à la structuration des données dans des systèmes *schemaless*. Ces systèmes n'ont pas de définition de schéma a priori comme cela est fait dans les SGBD relationnels notamment. Les systèmes ne supportent pas d'abstraction de schéma sur laquelle s'appuyer. La structure est implicite et partielle. La frontière entre schémas et données est effacée ou dans certains cas est très faible. Contrairement aux SGBD relationnels par exemple, nous considérons ici des systèmes qui ne suivent pas l'architecture Ansi/Sparc. Cette architecture distingue des schémas interne, conceptuel et externe en permettant une bonne séparation entre les trois niveaux d'abstraction.

Les systèmes NoSQL sont généralement classés en fonction du modèle de données utilisé, à savoir clé-valeur, orienté colonnes, graphes et orientés documents. Il n'y a cependant pas de modèle unique dans chaque famille de systèmes. Chaque système implémente la gestion de données à sa manière.

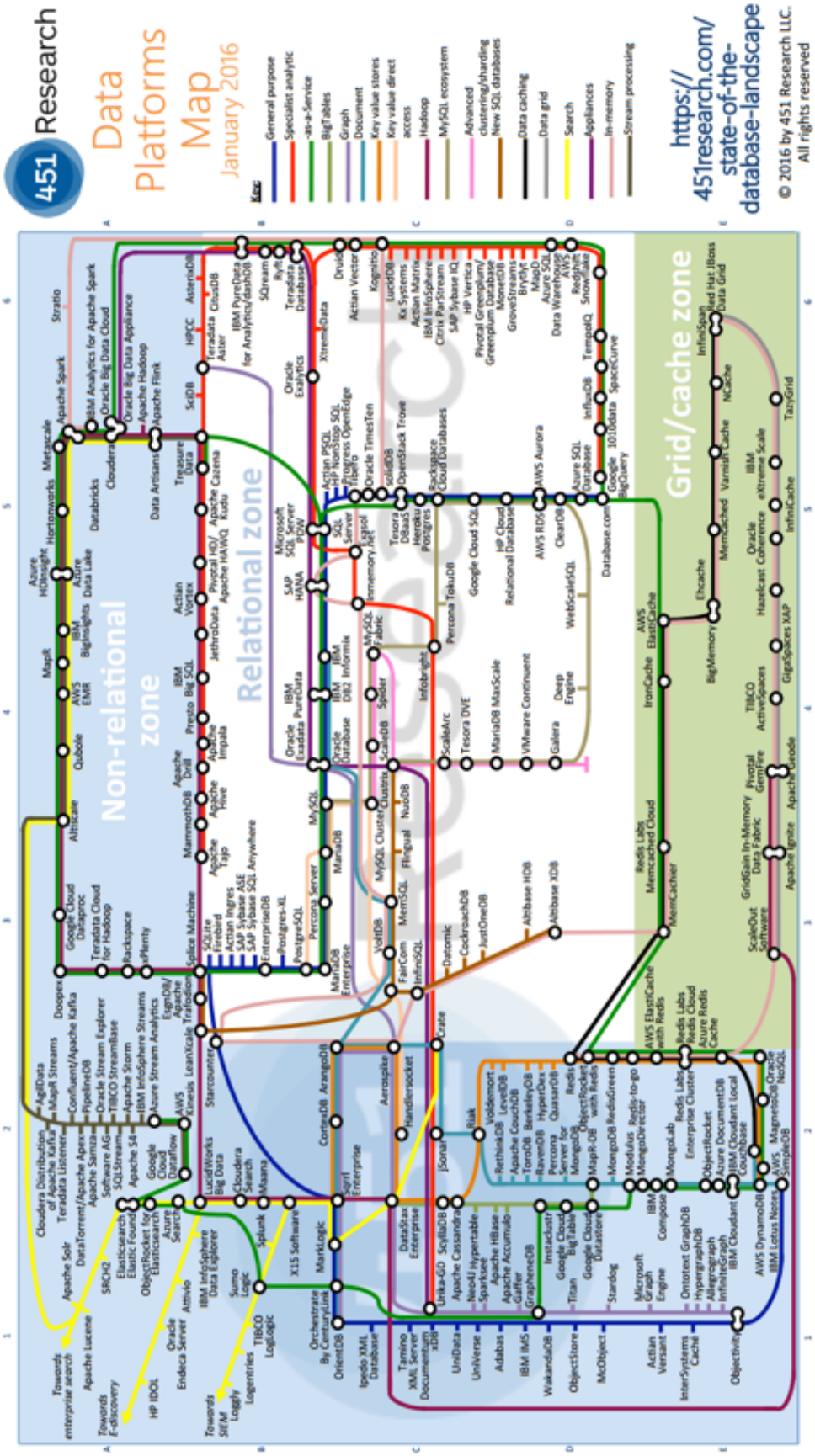


FIGURE 2.1: Carte des plates-formes de gestion de données [1]

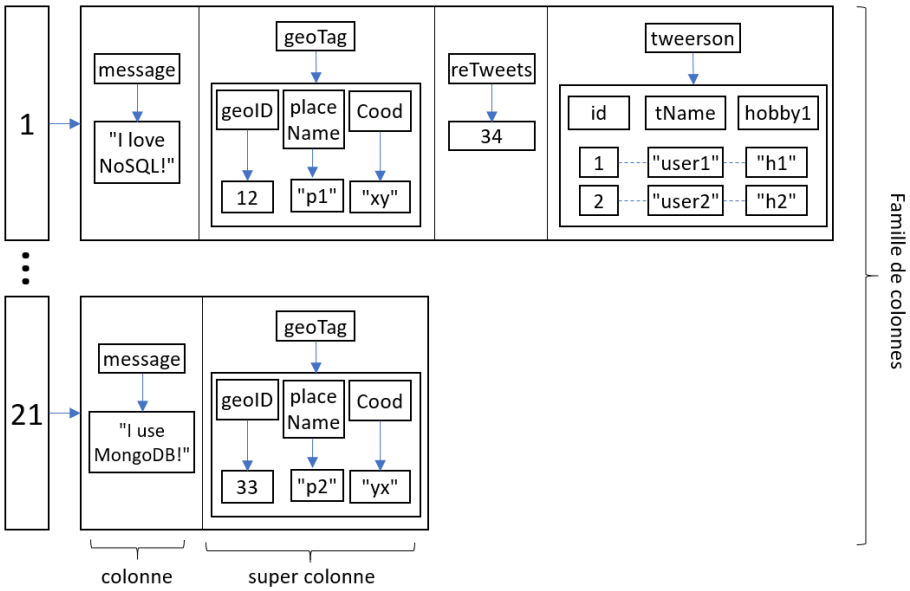


FIGURE 2.2: Exemple de données Cassandra dans un contexte Twitter

Le modèle clé-valeur, représente les données comme des couples associant une valeur à un identifiant. L'utilisateur a total liberté pour le type des valeurs. Le type n'est pas prédéfini et peut être atomique ou complexe. Comme le type est méconnu, il n'y pas de primitives d'accès de haut niveau pour le manipuler. L'interface est très basique, composée de primitives PUT et GET. En général ces interfaces ne permettent pas d'accès sémantique déclaratif. Certaines offrent des primitives de manipulation de tableaux et un support MapReduce. Les systèmes de stockage Dynamo [27], Redis [28] et Voldemort [29] utilisent un modèle clé-valeur.

Les modèles orientés colonnes offrent plus de moyens de structuration. Ils sont été lancés par Big table [30] et utilisés dans des systèmes tels que HBase [31] et Cassandra [15]. Il s'agit d'extensions du modèle relationnel pour utiliser des relations qui ne sont pas en première forme normale. Les données sont représentées avec des attributs (les colonnes) dont les types peuvent être des valeurs atomiques ou être eux mêmes plusieurs colonnes imbriquées, cf. Figure 2.2. Ainsi, le modèle permet d'avoir des tables imbriquées dans des tables. Les valeurs des attributs sont stockées avec une estampille. Lorsqu'une donnée est modifiée, le système crée une nouvelle version de la donnée avec un nouvelle estampille. Ce choix permet la historisation des données et l'accélération des mises à jour. Notons que le système Cassandra s'appuie sur Big Table pour le modèle de données et sur Dynamo pour le stockage. Les langages de requêtes pour les systèmes appartenant au modèle orientée colonne sont variés. HBase propose des opérations basiques tels que Get, Set, Scan et Delete alors que Cassandra offre le langage de requêtes CQL. Ce langage, inspiré de SQL, permet la création, l'insertion et la sélection de données.

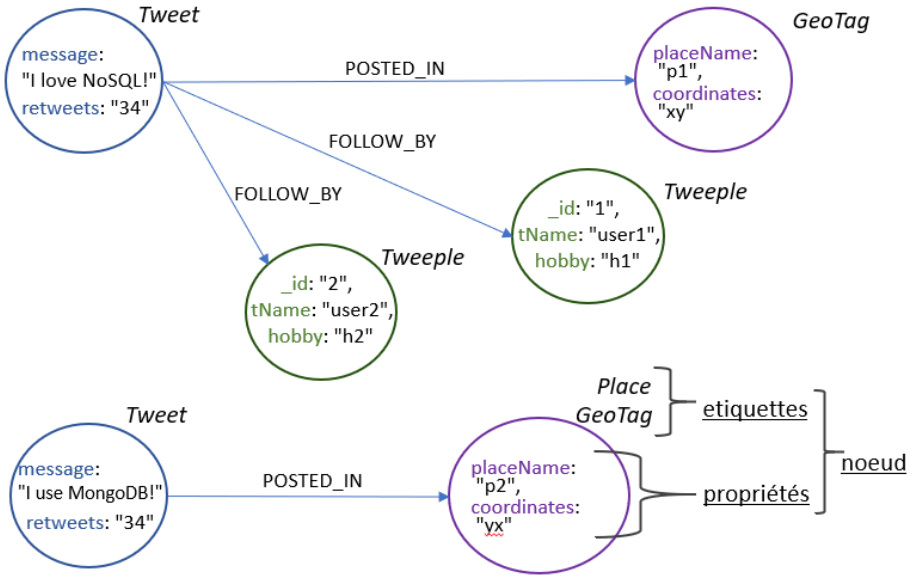


FIGURE 2.3: Exemple de données Neo4J dans un contexte Twitter

Les modèles orientés graphes, permettent beaucoup de souplesse dans la représentation des données à l'aide de nœuds et d'arcs, cf. Figure 2.3. Les graphes peuvent être construits sans contrainte et avoir des structures cycliques. Les nœuds et les arcs peuvent avoir des propriétés avec des valeurs atomiques et des tableaux de valeurs atomiques. Les nœuds peuvent porter une ou plusieurs étiquettes (dites *label*). Par exemple, un nœud peut porter les étiquettes GeoTag, Place, Endroit ... Ces étiquettes aident au regroupement des nœuds. Les arcs peuvent avoir une seule étiquette. Les systèmes Neo4J [14] et Titan [32] sont des systèmes orientés graphes. Ces deux systèmes proposent des langages de manipulation de données puissants pour interroger et mettre à jours les graphes. Les langages Cypher de Neo4J et Gremlin du système Titan sont de 2ème ordre. Les requêtes concernent à la fois la structure et les valeurs. La navigation dans les graphes se fait à l'aide de patrons, en incluant des opérateurs spécifiques tels que la récursion.

Dans la famille des systèmes orientés document, les données sont représentées dans un format semi-structuré hiérarchique de type JSON ou XML. Un "document" a une structure arborescente qui contient une liste de champs qui ont une valeur et qui peuvent à leur tour contenir une structure avec une liste de champs et ainsi de suite. Les documents sont regroupés en collections mais il n'y a pas de type imposé pour les documents d'une collection. Les systèmes orientés documents sont au coeur de cette thèse et seront développés par la suite. Certains systèmes, dits polyglottes, utilisent plusieurs modèles de données. Les systèmes OrientDB [33] et ArangoDB [34], par exemple, utilisent les modèles clé-valeur, document et graphe.

La suite de ce chapitre est consacrée à des travaux autour des systèmes orientés document. Nous détaillons d'abord les caractéristiques de ces systèmes en mettant l'accent sur la représentation et la manipulation des données. Rappelons que ces systèmes ne gèrent pas de schéma conceptuel mais le choix de structuration de données s'avère néanmoins important. Ainsi, certains travaux cherchent à adopter des approches de modélisation conceptuelle pour le développement avec des bases orientées document. Ces travaux sont présentés dans la section 2.2.2.2. Dans la section 2.3.1, nous présentons des travaux qui suivent une logique d'ingénierie inverse et proposent des solutions pour explorer des bases de documents opérationnelles et abstraire des informations sur la structuration des données.

2.2 Approche orientée document

Dans cette thèse nous nous intéressons surtout aux aspects liés à la structuration de données. Pour la plupart des systèmes orientés documents, l'information est stockée dans un format style JSON, *JavaScript Object Notation*. L'information est ainsi représentée avec des structures hiérarchiques, parfois très complexes. Les systèmes qui utilisent JSON bénéficient des outils proposés pour JavaScript, ce qui a contribué à leur popularité. Dans la suite, nous rappelons d'abord les fondements de JSON et ensuite abordons trois systèmes représentatifs des systèmes orientés documents. Il s'agit de MongoDB, CouchDB et AsterixDB. Nous avons choisi de travailler avec MongoDB car ils est très largement utilisé. Néanmoins, AsterixDB offre des choix plus intéressants en matière de modélisation des données.

2.2.1 JSON

JSON (JavaScript Object Notation) est un format d'échanges de données léger [35]. Il est dérivé de la notation des objets du langage JavaScript. Ce format permet de représenter les données de façon semi-structurée comme le permet XML [36]. Il est largement utilisé dans de nombreuses architectures d'applications, divers langages de programmation et systèmes NoSQL.

Un texte JSON est formé par un ensemble d'objets. Un objet est une structure composée de zéro ou plusieurs paires *nom* : *valeur*. Tous les champs d'un objet sont entourés d'une paire d'accolades. Une valeur JSON peut être atomique, une chaîne de caractères, un tableau ou un objet. Les éléments d'un tableau sont entourés par des crochets.

La Figure 2.4 présente un exemple du format JSON. L'information concernant l'objet nommé *Image* contient six champs. Le champ *IDs* a comme valeur un tableau d'entiers alors que *Thumbnail* a comme valeur un autre object composé de trois champs avec de valeurs simples.

```
{
  "Image":  {
      "Image":   800,
      "Height":  600,
      "Title":   "View from 15th Floor",
      "Thumbnail":   {
          "Url":  "http://www.example.com/image/45632919943",
          "Height":  125,
          "Width":   100
      },
      "Animated":   false,
      "IDs":   [116, 943, 234, 38793]
  }
}
```

FIGURE 2.4: Exemple de format JSON

2.2.2 Système MongoDB

Le système MongoDB propose de nombreuses fonctionnalités pour la gestion de masses de données. Il peut être déployé sur une architecture client-serveur mais aussi sur un cloud. Il offre des facilités pour une scalabilité horizontale et de la haute disponibilité. MongoDB permet le partitionnement des données, sharding et implante des protocoles de réplication maître-esclaves (cf. section 2.2.2.4). Les aspects liés à la modélisation de données sont abordés dans les sections 2.2.2.1 et 2.2.2.2. Les opérateurs proposés pour leur manipulation sont présentes dans la section 2.2.2.3.

2.2.2.1 Modèle de données

MongoDB ne supporte pas de schéma de base de données explicite et préalable à la création de la base. Son modèle de données, basé sur JSON, est composé de collections et de documents.

Les collections sont composées des documents dont la structure peut être identique ou différent. Un document est composé par un ensemble des paires `cle:valeur`. Tout document a l'identifiant `_id`, dont la valeur est assignée automatiquement par le système ou explicitement par l'utilisateur. Le système de types comprend les types atomiques (String, Integer, Double, Boolean), des documents et des tableaux de valeurs ou de documents. Les contraintes d'intégrité ne sont pas prises en charge par le système.

Il'y a deux façons de relier les documents, par *imbrication* ou par *référencement*. La première permet à un ou plusieurs documents d'être intégrés par un document. Concernant le *référencement*, la valeur d'un attribut peut être l'identifiant d'un document dans une autre collection.

La Figure 2.5 illustre le modèle de données MongoDB dans un contexte *Tweeter*. Il y a deux collections nommées `Tweets` et `GeoTag`. La collection `Geotag` est composée de documents homogènes. Chaque document contient des informations concernant le nom d'un endroit et ses coordonnées. La collection `Tweets` contient des documents avec l'identifiant du tweet, le message, la geo localisation, le nombre de retweets et pour certains documents, les personnes qui l'aiment. L'information des personnes qui aiment le tweet est incluse dans le tableau nommé tweerson. Les documents imbriqués contiennent l'information des personnes. Cela ajoute un niveau de profondeur dans la structure de la collection. L'attribut `geoLocation` signale l'emplacement de l'utilisateur lorsqu'il a publié le Tweet. Pour cela, une référence vers le document pertinent dans la collection `Geotag` est utilisée.

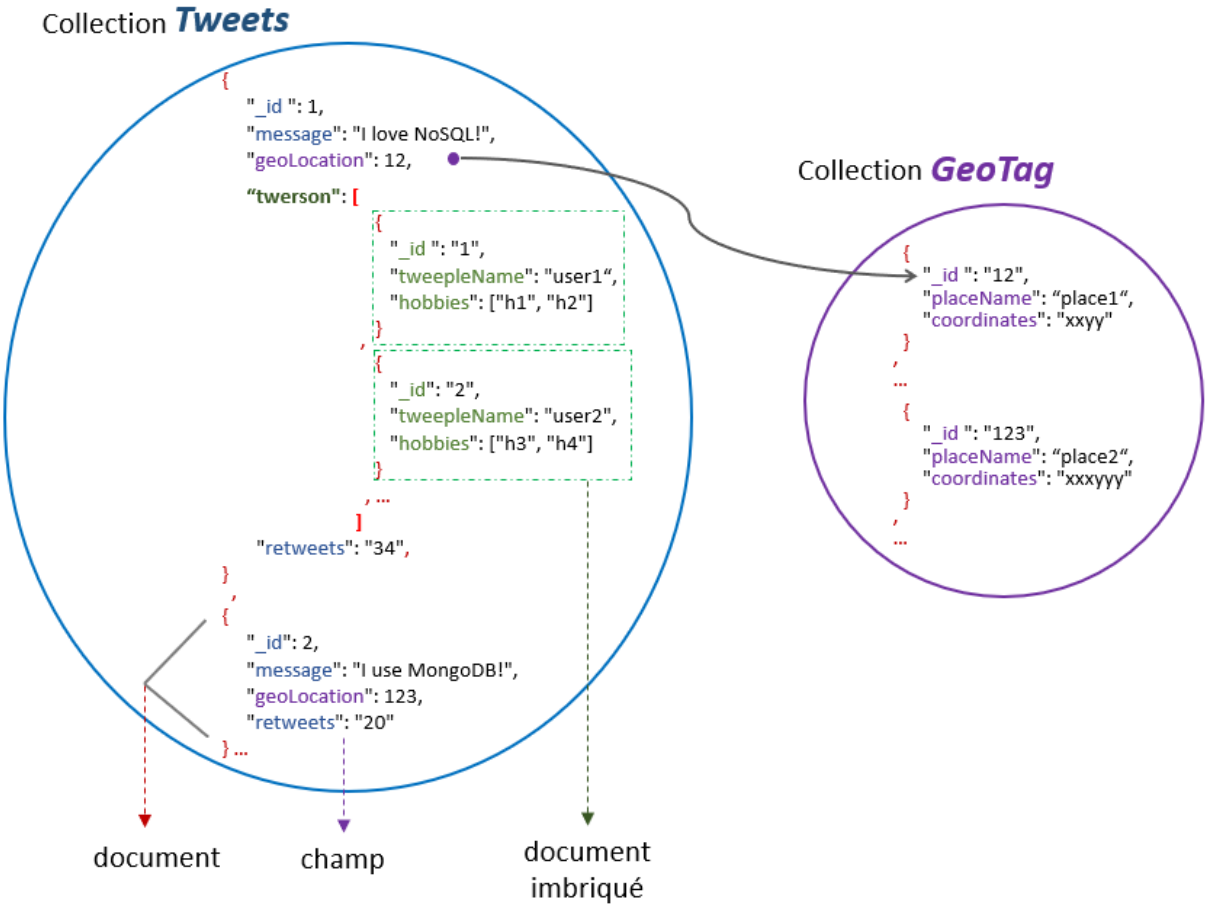


FIGURE 2.5: Exemple de données MongoDB dans un contexte Twitter

2.2.2.2 Flexibilité dans la structuration

Le système de types de MongoDB permet beaucoup de flexibilité et ouvre la porte à de nombreuses possibilités de structuration qui peuvent mélanger l'imbrication, le référencement et la duplication. Des différents alternatives de structuration sont parfois comparées aux solutions normalisées et dé-normalisés du modèle de données relationnelles [16].

La Figure 2.6 illustre trois manières de structurer les informations sur les tweets et les twerson dans notre contexte *Tweeter*. La Figure 2.6a montre une collection de Tweets et une collection de Twerson. Les documents de la collection Tweets référencent les documents dans la collection Twerson. Dans la Figure 2.6b le choix est différent, il y a une seule collection Tweets. Les

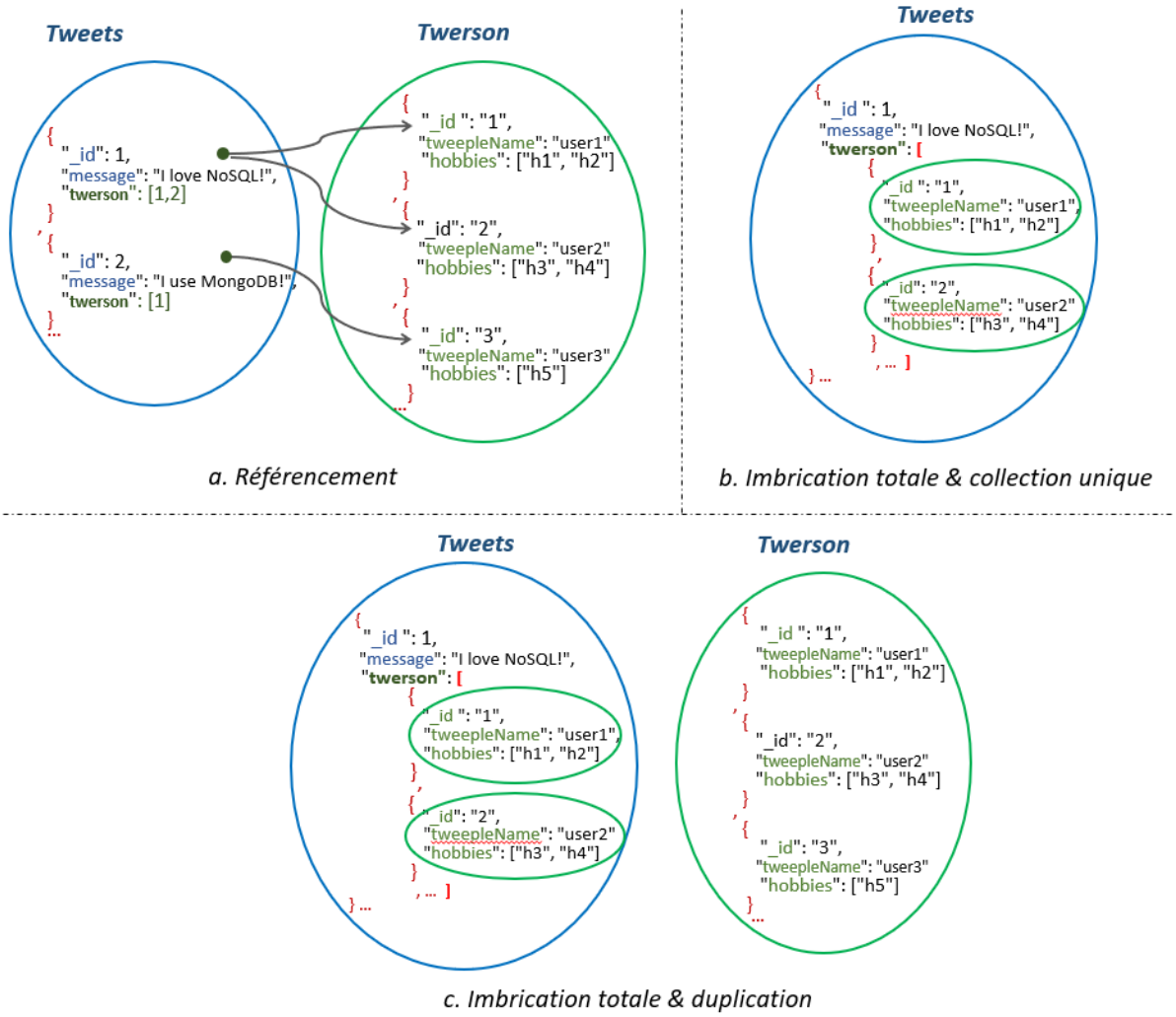


FIGURE 2.6: Flexibilité de structuration dans les systèmes orientés documents : contexte Tweets

documents de cette collection imbriquent leurs twerson respectifs. La Figure 2.6c considère aussi deux collections sans référencement. La collection Tweets existe en imbriquant les tweerson et la collection Twerson contient des documents twerson. Dans ce choix, il y duplication de l'information concernant les twerson.

En réalité, il y a de nombreuses possibilités, de manière générale, tout en garantissant la complètude des données, les collections peuvent être structurées et reliées de diverses manières, e.g. collections séparées et sans imbrication, collections complètement imbriquées, combinaison d'imbrications et de référencements ou duplication de données. Le choix de la meilleure structuration n'aura probablement pas une réponse unique ni absolue et dépendra des priorités et besoins d'accès du moment. Nous pouvons imaginer que nous faisons un choix unique, ou que pour un certain temps on peut avoir deux alternatives.

1	db.tweets.find()	Afficher tous les documents
2	db. tweets.find().pretty()	Afficher les données dans un format le plus clair
3	db. tweets.find().sort({“message”: 1})	Afficher tous les documents et trier par nom croissant
4	db. tweets.find({"retweets": 34}).pretty()	Sélection. Filtrer par un ou plusieurs champs. Afficher tous les champs pour chaque élément trouvé.
5	db. tweets.find({"retweets": 34} , {“message":1,“geoLocation": 1})	Projection Filtrer par un ou plusieurs champs (premier paramètre). Seuls les champs indiqués dans le second paramètre seront affichés
6	db. tweets.find({"retweets":{ **$gt** : 30} }, {“message": 1, "geoLocation": 1 })	Opérateur de comparaison $ gt (>) Afficher les tweets dont le nombre de retweets est supérieur à 30 Pour chaque tweet, seulement le message et la geoLocation sont affichés.
7	db. tweets.find({ “twerson.hobbies":{ $all: ["tennis" , "swimming"] }}, {"message": 1, “geolocation": 1 })	Trouver des tweets avec le twerson ayant "tennis" et "swimming" parmi leurs intérêts.
8	db.tweets.find({"twerson": {$exists : true} })	Recherche des tweets dont l'information de twerson est renseigné

FIGURE 2.7: Exemples de requêtes avec Find sur les données de la Figure 2.5

La manière dont sont structurées les données a un fort impact sur la taille de la base, les performances des requêtes et la lisibilité du code des requêtes, ce qui influence la maintenabilité et l'utilisabilité de la base ainsi que de ses applications. Cela sera vérifié de manière expérimentale dans les chapitre 4 et 8 où plusieurs patrons de comportement sont mis en évidance.

2.2.2.3 Interrogation et manipulation des données

MongoDB propose un ensemble d'opérations permettant d'insérer, de supprimer, d'interroger et de mettre à jour des documents. Concernant l'interrogation, un ensemble d'opérateurs de sélection, de projection, d'agrégation et de classement est fourni. Des opérateurs logiques, des opérateurs arithmétiques, de manipulation de dates et de chaînes de caractères sont également disponibles.

Une requête porte toujours sur une collection de documents. L'opérateur *find* permet de sélectionner des documents d'une collection. La Figure 2.7 introduit quelques exemples d'utilisation. Les lignes 1,2 et 3, sélectionne tous les documents de la collection tweets. La 3ieme requête utilise l'opérateur de tri sur un attribut. Les conditions de sélection portent sur les valeurs des attributs (e.g. ligne 4) et peuvent inclure des fonctions logiques et arithmétiques (e.g. supérieur à, $gt, utilisée dans la requête de la ligne 6). Notons un opérateur particulier, nommé *$exist*), pour vérifier qu'un attribut existe dans un document. La ligne 8 montre son utilisation.

L'opérateur *find* permet aussi d'indiquer les attributs sur lesquels on souhaite effectuer une projection. Ceci est donné comme deuxième paramètre du *find*. Les requêtes des lignes 5, 6 et 7 font des projections.

Lorsque l'on manipule des documents avec des attributs dont la valeur est un tableau, il est possible d'exprimer des conditions de sélection sur les éléments du tableau. Par exemple, à l'aide de l'opérateur $all, pour vérifier que le paramètre figure dans tous les éléments du tableau. La ligne 7 montre une requête qui met une condition sur natation et tennis. Cette requête sélectionne les tweets (en gardant le message et la geo-localisation) dont tous ses tweerson ont les hobbies natation et tennis. L'opérateur *find* peut utiliser les fonctions d'agregation *count* et *distinct*. Si l'on souhaite d'autres agrégations, on doit utiliser *aggregate*. [1]

L'*aggregate* traite les documents en plusieurs étapes correspondant à la concaténation d'instructions. Les documents sont transformés jusqu'à l'obtention du résultat global. La Figure 2.8 présente des opérateurs disponibles pour un *aggregate* ainsi que la notation SQL qui correspondrait aux opérations analogues. L'opérateur *$group* permet le partitionnement des données, *$match* la sélection, *$sort*, le tri des données, *$project* pour choisir les champs à garder dans le résultat, *$limit* pour restreindre le nombre de réponses, *$skip* pour sauter n résultats et *$unwind* pour aplatir les tableaux.

Dans MongoDB, il n'y a pas d'opérateur de jointure à proprement parler. Il fournit cependant l'opérateur *$lookup*, qui met en oeuvre une sorte de jointure externe entre collections. Il est illustré ci-après.

SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM	$sum
COUNT	$sum
JOIN	Pas d'opérateur direct $lookup pour faire une correspondance avec les documents d'une collection.
-	$unwind pour gérer les champs incorporés dans un document

```
SELECT cust_id,
      SUM (price) as total
FROM orders
WHERE status = 'A'
GROUP BY cust_id
```

```
db.orders.aggregate (
    [
        { $match: { status: 'A'} },
        { $group: { _id: "$cust_id" , total: { $suc: "$price"} } }
    ]
)
```

FIGURE 2.8: Opérateurs de l'*aggregate* et fonctions similaires dans SQL

1. Les mises en oeuvre avec mapReduce sont possibles également mais ne seront pas développées ici.

Concernent la manipulation des tableaux, l'opérateur *$unwind* joue un rôle important en permettant de les « aplatir » afin d'extraire l'information des éléments. La Figure 2.9 introduit un exemple d'utilisation de cet opérateur sur la collection `Tweets`. Nous nous intéressons aux tweets et twerson avec plus de 50 followers.

Nous sommes dans le cas de documents qui imbriquent un tableau de documents. Lorsque la condition de sélection porte sur les documents contenus dans le tableau, la réponse sont les documents dont le tableau contient des documents vérifiant la condition de sélection. Dans notre exemple, cela peut être apprécié pour la première instruction de l'*aggregate* qui fournit un document tweet avec le tableau au se trouve une personne qui vérifie la condition de plus de 50 followers. Pour accéder à ce document précis, il faut forcément aplatir le tableau pour ensuite faire à nouveau une sélection des documents souhaités. Comme dans les deuxième et troisième instructions de l'*aggregate* de l'exemple. L'opérateur $*unwind* dé-construit un tableau en produisant un nouveau document pour chaque élément du tableau (soit du type atomique, soit du type document). Les attributs du document qui contient le tableau sont aussi inclus dans chaque nouveau document.

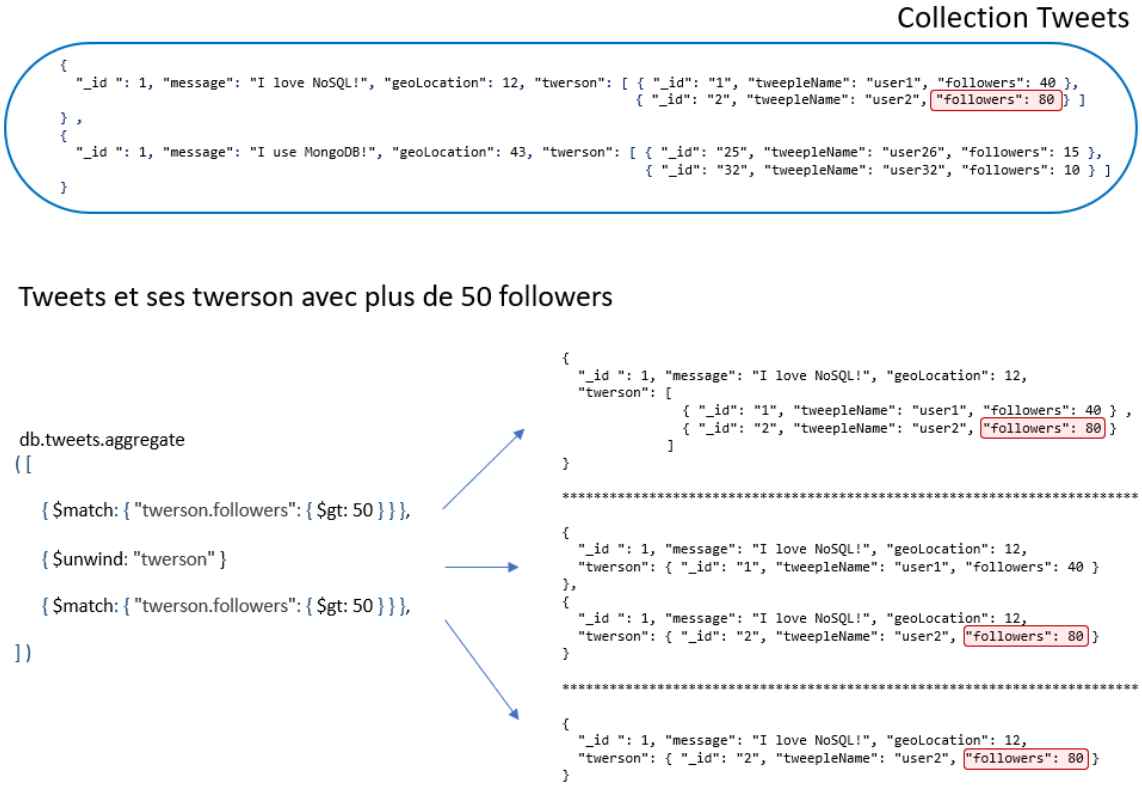


FIGURE 2.9: Exemple d'utilisation de l'opérateur *$unwind*

L'opérateur $lookup permet d'obtenir une forme de jointure externe. Il travaille sur les documents d'une collection "locale" et ajoute à chaque document un tableau contenant les documents de la collection "externe" vérifiant la condition de jointure. Si le champ local (désigné par *localfield*) sur lequel porte la condition de jointure n'existe pas dans un document de la collection locale, le document est ignoré. S'il existe mais qu'il n'y a pas de correspondance avec un document de la collection externe, le document de la collection locale est gardé dans la réponse avec un tableau vide pour le champs créé pour la jointure.

L'exemple de la Figure 2.10 établit une jointure entre la collection `Tweets` et la collection `Twerson`. Nous nous intéressons aux personnes ayant le même tweet préféré. La jointure reprend la collection de `Tweets` et le champ *message* et la collection externe `Tweerson` avec le champ *favoriteTweet*. L'opération ajoute à chaque tweet l'information des fans dans un tableau appelé *tweersonFans_docs*.

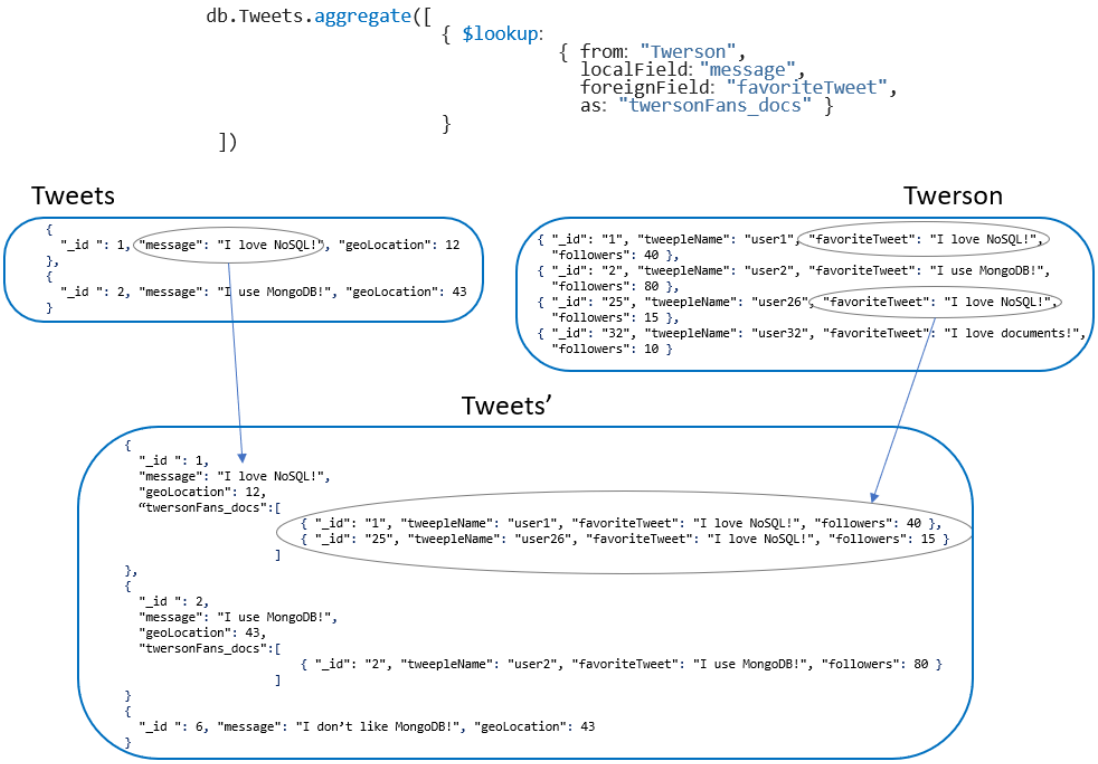


FIGURE 2.10: Exemple d'utilisation de l'opérateur *$lookup*

La condition de jointure peut porter sur plusieurs champs et peut utiliser des opérateurs de comparaison tels que l'égalité et supérieur à. Il n'est pas possible d'exprimer des conditions de jointure sur un tableau. Si cela s'avère nécessaire, les éléments doivent d'abord être extraits du tableau à l'aide de l'opérateur *$unwind*.

MongoDB permet également de consulter les résultats fournis par une requête. Il est possible de traiter chaque document de la réponse à l'aide de *JavaScript*. En conséquence, le programme d'interrogation combinera les opérateurs fournis par MongoDB et les instructions *JavaScript*.

La mise en œuvre d'une requête dépend de la structure de données et peut être effectuée de plusieurs façons différentes. La complexité d'un programme de recherche est fortement liée au nombre de collections nécessaires et à la complexité de leur structuration. La performance et la lisibilité des programmes reposent sur les compétences et les connaissances du développeur d'applications.

2.2.2.4 Fragmentation et Réplication des données

MongoDB gère la réplication à l'aide d'un ensemble de répliques (dit *replica set*) qui distingue une copie primaire et N copies secondaires (cf. Figure 2.11).

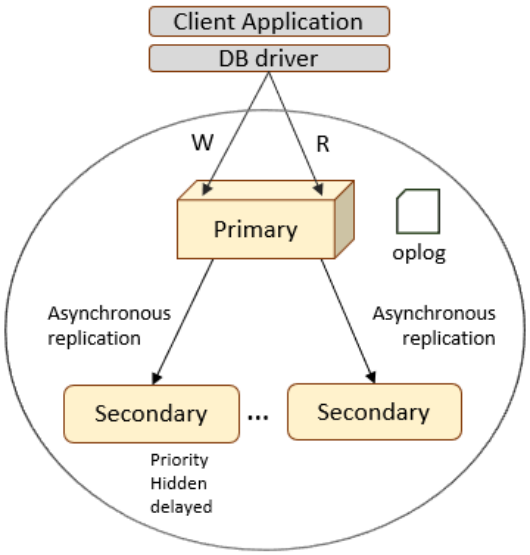


FIGURE 2.11: Réplication de données dans MongoDB

La copie primaire :

— traite toutes les opérations de lecture et d'écriture,

— gère un journal (registre *oplog*) qui conserve en continu les opérations de modification des données et

— gère la synchronisation des copies secondaires de manière asynchrone.

Les copies secondaires :

— traitent des opérations de lecture et

— répliquent le journal *oplog*

Les copies secondaires peuvent jouer des rôles différents et leur synchronisation peut être différentiée. Dans la configuration, il est possible de préciser si la copie est :

— prioritaire : dans ce cas la copie est synchronisée en priorité et peut devenir copie primaire.

— cachée : la copie n'est pas disponible en lecture et ne peut pas devenir copie primaire.

— retardée : la copie est synchronisée de manière asynchrone après un délai donné. Cela permet une forme de sauvegarde et la copie peut être utilisée pour revenir à un état précédent. La copie secondaire retardée ne peut pas devenir copie primaire.

MongoDB fournit une scalabilité horizontale en utilisant une technique de fragmentation, dite *sharding*. Il s'agit de faire une fragmentation horizontale de la (les) collections afin de répartir son stockage sur plusieurs nœuds. La fragmentation de la collection est faite à l'aide d'une clé de partitionnement. Les fragments sont appelés *chunks*. La clé de partitionnement peut être n'importe quel champ d'index (ou champ composé d'index) présent dans chaque document de la collection (cf. Figure 2.12). Il existe deux approches de partitionnement, *Hashing based partitioning*, utilisant une fonction de hachage sur la clé de partitionnement et *Range based partitioning* qui utilise des intervalles de valeurs d'un attribut pour créer les fragments.

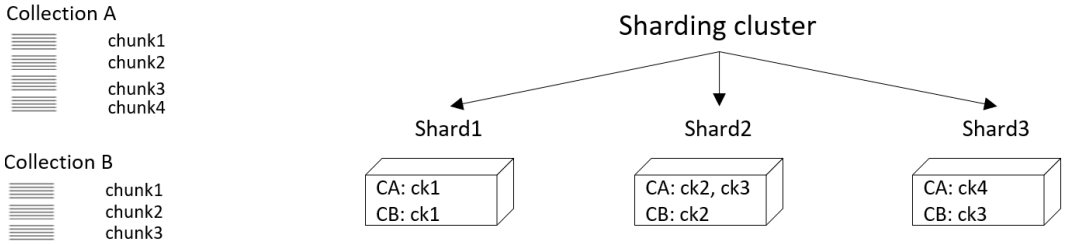


FIGURE 2.12: Fragmentation «sharding» dans MongoDB

MongoDB utilise pour le stockage un *cluster* de machines, dit *sharding cluster*. Un *shard* est un où plusieurs nœuds destinés à stocker des données. Le routeur de requêtes se charge de diriger les opérations des clients vers le ou les *shards* stockant les fragments appropriés et de renvoyer les résultats aux clients. Ce routeur s'appuie sur une configuration avec les méta-données du cluster incluant les informations de placement des données. Pour assurer la tolérance aux pannes, chaque fragment peut être répliqué (cf. Figure 2.13).

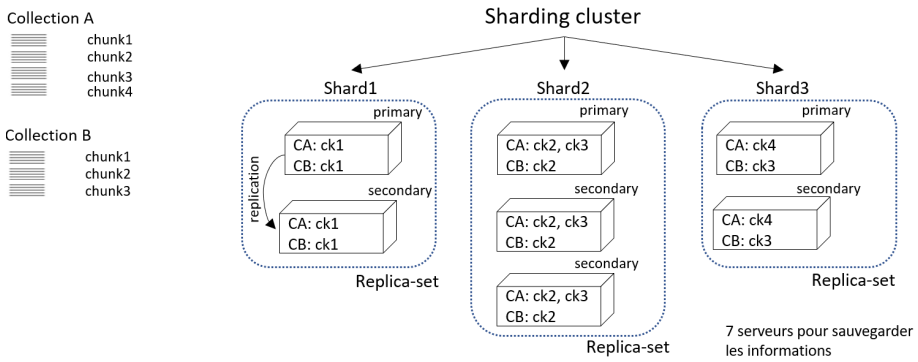


FIGURE 2.13: *Sharding* et réplication dans MongoDB

2.2.3 Système CouchDB

CouchDB [37] est un système orienté documents dont l'architecture est particulièrement motivée par le passage à l'échelle. Il n'offre néanmoins pas de fonctionalités tels qu'un support transactionnel complet ou un langage de requêtes.

Les données sont gérées sous forme de fichiers JSON avec des champs des types de données mentionnés précédemment. Chaque document possède un identifiant unique, `_id`, et un identifiant de version, `_rev` (ID de révision). Les mises à jour des documents, donnent lieu à la création d'une version avec une nouvelle valeur de `_rev`. Le versionnement des documents est utilisé dans la gestion des accès concurrents.

CouchDB n'adopte pas la notion de collection comme dans MongoDB mais une notion de vue. L'utilisateur peut créer des vues associées à des requêtes sur les documents. Il n'y a cependant pas de langage de requêtes à proprement parler. L'utilisateur doit écrire des programme en Javascript. L'exécution des programmes peut être distribuée et parallélisée avec map-reduce. La distribution des données peut donner lieu à la création de copies sur plusieurs serveurs. Les données dupliquées sont mises à jour de manière asynchrone.

2.2.4 Système AsterixDB

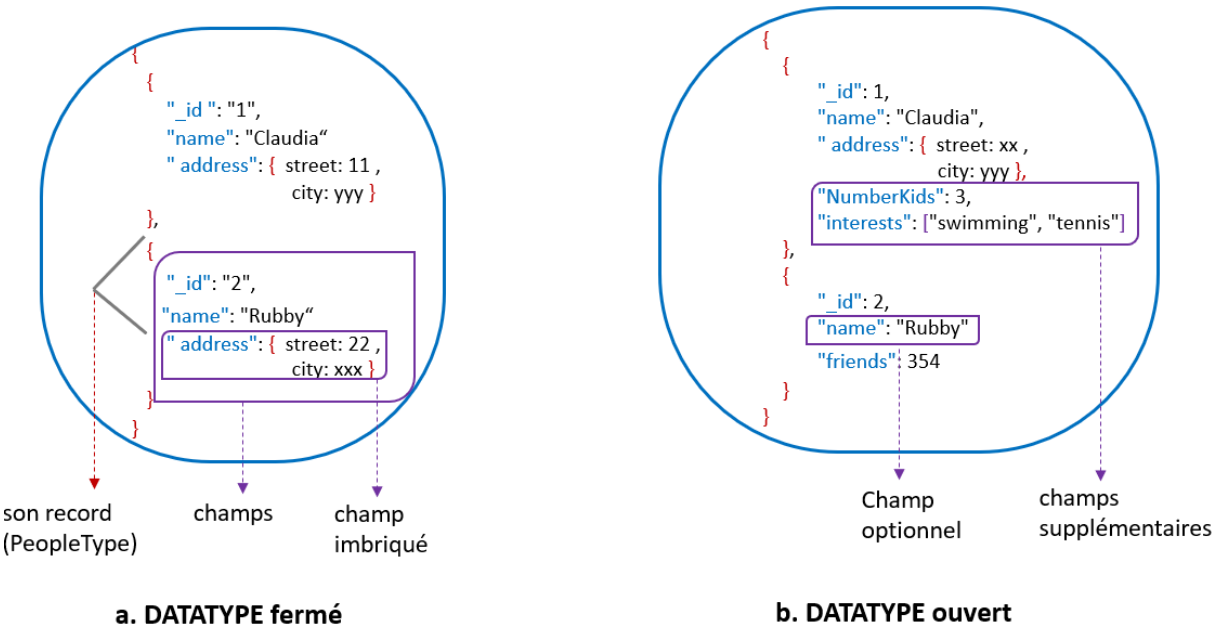
Apache AsterixDB [38, 39] se présente comme un BDMS, pour *Big Data Management System*. En effet, il offre des fonctionnalités pour gérer des masses de données en facilitant le chargement des données, en gérant un stockage distribué, un support transactionnel ACID, l'accès à des

```
create dataset People (PeopleType)
primary key id;

create type PeopleType closed {
    _id:int32,
    name: String
    address:{
                street: string,
               city: yyy
             }
}
```

```
create dataset People
(PeopleType) primary key id;

create type PeopleType open {
    _id:int32,
    name: String,
    friend: int32?
}
```



a. DATATYPE fermé **b. DATATYPE ouvert**

FIGURE 2.14: Datatypes et Datasets dans AsterixDB

systèmes de stockage externes, des mécanismes d'indexation et un langage de requêtes déclaratif performant.

AsterixDB utilise un modèle de données semi-structuré flexible orienté documents. Il se base sur JSON et introduit des aspects de gestion de schéma intéressants que nous développons ci-après. Dans la section 2.2.4.2 nous aborderons AQL, *Asterix Query Language*. Les aspects systèmes ne seront pas approfondis ici.

2.2.4.1 Modèle de données

Les bases de données d'AsterixDB sont nommées *dataverse*. Elles stockent des collections, dites *datasets*. Les *datasets* sont nommés et on leur associe un type des données, *datatype*. Les

documents, dits *registres* JSON, stockés dans le dataset doivent respecter son type de données. La Figure 2.14 illustre des déclarations de *datasets*, *datatypes* et des instances.

Un type des données permet de spécifier les champs et leur type pour les documents JSON. La particularité intéressante est qu'un type de données peut être ouvert ou fermé. Un type de données fermé est une définition stricte que doit être respectée par les instances. Les champs déclarés et leur type sont obligatoires et les instances ne peuvent pas avoir d'autres champs.

Les types ouverts ne sont pas stricts. Un champ peut être obligatoire ou facultatif. Le champ facultatif (noté avec ?) peut être présent ou non dans les instances, mais lorsqu'il est présent, il doit être conforme à la définition du type de données. Les instances peuvent avoir des champs non prévus par le type. Ainsi, les *datasets* déclarés avec un type de données ouvert peuvent stocker des instances dont la structure peut être hétérogène.

L'utilisation de types de données ouverts et fermés permet au développeur d'introduire des niveaux d'exigence dans le typage des instances d'une manière assez flexible.

AsterixDB offre un support de stockage distribué propre mais prend également en charge l'accès en lecture à des données stockées sur d'autres supports [40]. Pour cela, l'utilisateur déclare le type de données approprié et un *external dataset* pour lequel il donne le chemin d'accès aux données. La Figure 2.15 montre un example de déclarations.

```
12.34.56.78|2013-12-22T12:13:32-0800|Nicholas|GET|/|200|2279
12.34.56.78|2013-12-22T12:13:33-0800|Nicholas|GET|/list|200|5299
```

```
create type AccessLogType as closed
{
    ip: string,
    time: string,
    user: string,
    verb: string,
    path: string,
    stat: int32,
    size: int32
}
```

```
create external dataset AccessLog(AccessLogType)
   using localfs
     (("path"="{hostname}://{path}"),
      ("format"="delimited-text"),
      ("delimiter"="|"))
```

FIGURE 2.15: Exemple de données provenant d'ensembles externes

2.2.4.2 Interrogation et manipulation des données

AQL, l'*Asterix Query Language*, est un langage déclaratif permettant d'insérer, de supprimer, d'interroger et de mettre à jour des documents [41, 42]. Pour l'interrogation, il offre un ensemble

assez complet d'opérateurs tels que la sélection, la projection, l'agrégation et le tri (cf. Figure 2.16). Des opérateurs logiques, des opérateurs arithmétiques, la manipulation de dates et de chaînes de caractères sont également disponibles. Les requêtes les plus utiles dans AQL sont basées sur l'expression FLWOR, for-let-where-order by-return. Les figures 2.17 et 2.18 montre des exemples. On y voit des requêtes avec des jointures, une équi-jointure et une jointure externe gauche.

SQL	AQL
from	for
select	return (but goes at the end of a query)
with	let
where	where
order by	order by
group by	group by
limit	limit

FIGURE 2.16: Opérateurs AQL et similarités avec SQL

La Figure 2.17, l'équi-jointure porte sur l'id de l'auteur du message et de l'utilisateur. La jointure a la sémantique classique où les documents qui ne vérifient pas la condition de jointure ne sont pas retenus dans la réponse. La réponse, crée des documents avec deux champs.

La Figure 2.18 utilise un for imbriqué pour calculer une jointure externe gauche. La réponse crée un registre par utilisateur et pour chacun d'eux le champ `messages` qui imbrique l'ensemble de messages dont il est auteur.

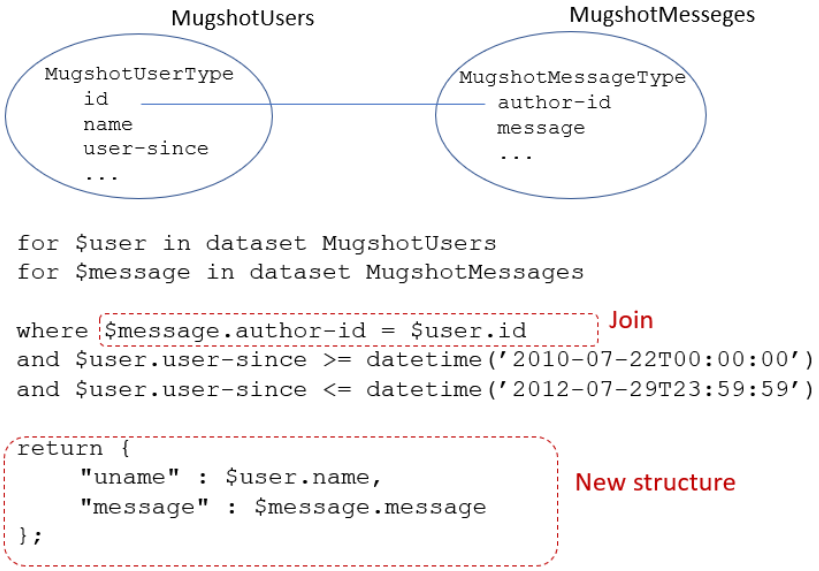


FIGURE 2.17: Exemple de requête *equijoin* en AQL

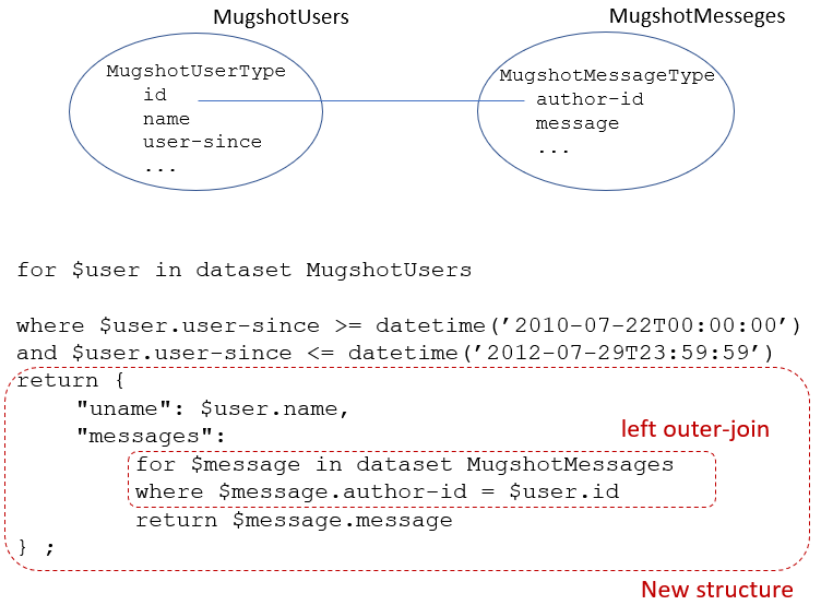
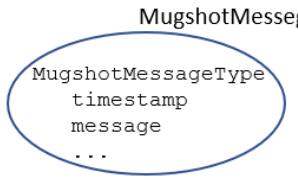


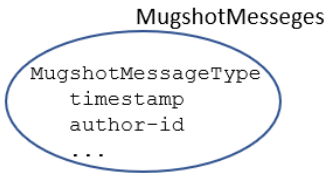
FIGURE 2.18: Exemple de requête avec une jointure gauche imbriquée en AQL

Les agrégations sont également supportées par AQL avec un opérateur group by (cf. Figures 2.19 et 2.20) et des fonctions telles que count, min, max, avg et sum.




```
avg(
     for $m in dataset MugshotMessages
     where $m.timestamp >= datetime("2014-01-01T00:00:00")
     and $m.timestamp < datetime("2014-04-01T00:00:00")
     return string-length($m.message)
)
```

FIGURE 2.19: Exemple de requête *avec la fonction avg* en AQL




```
for $msg in dataset MugshotMessages
where $msg.timestamp >= datetime("2014-02-20T00:00:00") and
      $msg.timestamp < datetime("2014-02-21T00:00:00")
group by $aid := $msg.author-id with $msg
let $cnt := count($msg)
order by $cnt desc
limit 3
return {
     "author" : $aid,
     "no messages" : $cnt
};
```

FIGURE 2.20: Exemple de requête *avec group by* en AQL

2.3 Modèles orientés document : choix de structuration

Les systèmes de gestion de données NoSQL orientés documents permettent une structuration des données avec une grande flexibilité et sans création préalable d'un schéma (contrairement aux SGBD relationnels). Dans ces solutions, il n'y a pas de séparation claire des couches logiques et physiques.

Plusieurs travaux portent sur l'analyse de cette flexibilité et ses impacts. Cette section est consacrée d'abord aux travaux, assez expérimentaux, récapitulant de bonnes pratiques de création de bases MongoDB. Nous nous consacrons ensuite à des approches plus conceptuelles qu'introduisent une démarche de modélisation.

2.3.1 Bonnes pratiques de structuration

Dans les systèmes orientés documents les possibilités de structuration des données sont assez nombreuses et le choix de structuration adopté reste crucial par son impact potentiel sur plusieurs aspects de la qualité des applications (Gómez et al., 2016).

Une des questions importantes se pose sur comment définir ou analyser si une structure est bien adaptée aux besoins des applications. En effet, chaque structuration peut présenter des avantages et des inconvénients différents en matière d'empreinte mémoire, redondance de données engendrée, coût de navigation dans les structures et d'obtention de certaines données, lisibilité des programmes, parmi d'autres.

Certains travaux fournissent un cadre directeur, des bonnes pratiques et des aspects à prendre en compte dans le choix des structures. Copeland [43] et MongoDB proposent des lignes directrices pour la création de bases MongoDB en s'appuyant sur des cas appliqués à différents domaines. Sadalage et Fowler [4] discutent divers modèles de données et produits NoSQL (MongoDB, Cassandra et Neo4j) et sur quelques problématiques de modélisation de bases orientées colonnes, orientées graphes et orientées documents. Ils notent l'importance de la structuration sans proposer d'approche.

Dans le guide de migration de MongoDB [44], les auteurs mettent en avant les « meilleures pratiques » pour créer des bases orientées documents. Notamment, ce qui concerne des aspects liés aux requêtes et à la modélisation des associations. En ce qui concerne les requêtes, il est conseillé :

— d'examiner la proportion des opérations de lecture et d'écriture

— d'identifier le type de ces opérations. Par exemple, pour une requête de recherche, identifier si elle appartient au type d'accès simple, d'agrégation ou de jointure. et,

— de considérer la volumétrie ainsi que le cycle de vie de l'information.

Même si dans MongoDB la notion de schéma n'existe pas, ce guide fournit des recommandations pour la structuration des données à partir de la modélisation des associations. Il propose l'imbrication ou le référencement selon la cardinalité de l'association.

Par rapport à l'imbrication, il est suggéré que ce soit appliqué à des relations *1-à-1* et *1-à-plusieurs*. Toutefois, cela dépend aussi s'il existe des requêtes qui demandent l'information groupée comme l'imbrication est faite.

Pour les associations *1-à-1*, il faut prêter attention car dans certains cas, il est préférable de référencer si : les données sont accédées fréquemment ou qu'une partie du document est modifiée fréquemment et que sa taille grandit.

C'est important de considérer les impacts de la volumétrie des données dans l'imbrication d'une association *1-à-plusieurs*. Imbriquer dans un document un tableau d'une centaine de documents n'aura pas le même impact que l'imbrication d'un tableau de milliers de documents.

Il peut également arriver que l'imbrication produise des documents très volumineux, sans nécessairement avoir des tableaux ou des tableaux contenant de nombreux éléments. Dans ce cas, il faut peut-être considérer qu'il est préférable de référencer. Le référencement est donc conseillé pour les associations *1-à-plusieurs* quand l'imbrication n'est pas suffisamment performante ou si les données sont référencées depuis plusieurs sources différentes [43].

Concernant le référencement, il est suggéré pour les associations *plusieurs-à-plusieurs*. Cependant, parce que cela est une approche normalisée, il faut être conscient qu'il sera nécessaire d'avoir plusieurs requêtes pour résoudre la référence. Il s'agit de l'implémentation des jointures.

Ces lignes directrices et bonnes pratiques fournissent un cadre intéressant même si nous notons constamment « cela dépend » . Les possibilités sont assez larges et difficiles à appréhender pour qu'un utilisateur prenne des décisions concernant la structuration. Ceci à cause des nombreux facteurs à prendre en compte en même temps, tel que le choix de : imbrication, volumétrie et type de requête.

2.3.2 Vers des approches conceptuelles

Certains travaux s'intéressent à la modélisation de données pour les bases NoSQL. Nous présentons ici des approches proposant de partir d'un modèle conceptuel afin de pouvoir étudier les alternatives de structuration et de trouver la plus adaptée selon un ensemble de besoins. Il s'agit de l'approche que nous préconisons dans cette thèse. Les travaux de recherche sont positionnés sur différents systèmes NoSQL, en particulier sur des systèmes orientés documents tels que MongoDB, les systèmes orientés colonnes comme Cassandra et les systèmes orientés graphes tels que Neo4j.

Nous remarquerons par la suite, que plusieurs choix de structuration vont introduire de la redondance des données. Pour la plupart, ce choix est motivé par un souci d'amélioration du temps de réponse des requêtes. Notons qu'aucun de ces travaux prend en compte la réplication des données réalisée par les systèmes sous-jascents. Cette réplication se situe plus au niveau de ce que serait un schéma physique d'une base distribuée. Elle est généralement implantée dans un souci d'amélioration de performances, de disponibilité et pour la tolérance aux pannes.

Travaux sur MongodDB
Zhao et al dans [45] propose un algorithme pour la création systématique d'une base de documents à partir d'un modèle entité-relation. Il crée, dans la base de documents, une collection par type d'entité avec imbrication récursive des entités référencées dans le modèle entité-relation. Ce

choix engendre en général de la redondance des données. En utilisant le vocabulaire du modèle relationnel, ce choix correspond à une structure dé-normalisée avec pré-calcul des jointures naturelles par imbrication de documents. Les expériences présentées pas les auteurs montrent les bonnes performances des requêtes pour lesquelles les jointures naturelles sont déjà matérialisées. La solution proposée introduit néanmoins beaucoup de redondance de données.

Travaux sur Cassandra
Des travaux [46], [47] et [48] s'intéressent aux alternatives de "modélisation" des données dans Cassandra (modèle "big table"). Les objectifs des études portent sur le stockage et les performances des requêtes. Rappelons que les requêtes sont ici implémentées avec des opérations simples, SET et GET.

Lombardo et al. proposent dans [47] un cadre qui choisit la structuration de données la plus avantageuse en déterminant les tables imbriquées les plus adaptées pour optimiser l'exécution des requêtes. À partir d'un diagramme entité-relation et les requêtes les plus utilisées, ils proposent de créer des tables redondantes dans le sens des accès nécessaires aux requêtes. Il s'agit d'une sorte de pré-calcul de certaines parties des requêtes. Les auteurs se focalisent sur le temps de réponse des requêtes mais n'abordent pas d'autres aspects tels que le stockage et encore moins des questions de qualité et "maintenabilité" des solutions.

Mior dans [49] et [46] propose un système de recommandation qui suggère un « schéma » optimisé selon une approche basée sur les coûts. Il utilise un modèle conceptuel, un ensemble de requêtes et de mis à jour paramétrées et exprimés sur le modèle conceptuel. Les fréquences d'exécution des requêtes sont également supposées connues et prises en compte. Il explore les alternatives de structuration des tables en décomposant l'ensemble des requêtes. Il propose une analyse statique de l'impact de stockage et des performances de requêtes. Une comparaison entre le schéma recommandé et deux autres schémas créés à la main est proposée.

Vajk et al. dans [48] et [50] proposent de trouver la structuration orientée colonnes la mieux adaptée à un ensemble de requêtes. Un modèle conceptuel doit être créé à l'aide d'un modèle de domaine spécifique proposé par eux et les requêtes sont élaborées à l'aide du langage OCL [51]. Ils génèrent d'abord un ensemble initial de schémas utilisant des parties dé-normalisées du modèle conceptuel. Ensuite, en utilisant l'ensemble des requêtes, l'ensemble des schémas est étendu et des variantes de requêtes sont générées afin de satisfaire les nouveaux schémas. Enfin, les coûts de chaque requête sont évalués pour chaque schéma et le meilleur est sélectionné. L'algorithme fonctionne sur des bases orientées colonnes qui permettent la recherche et le classement uniquement à l'aide d'une clé. Ceci est du par le fait que le système de stockage sous-jacent est clé-valeur (DynamoDB).

Travaux multi-modèles
Les travaux [52] et [8] s'intéressent à la manipulation d'un modèle intermédiaire qui permet de assembler les concepts de différentes familles de données non-relationnelles afin de pouvoir créer a partir de la même information, des alternatives de structuration selon le système cible.

Abdelhedi et al. [52], proposent de créer à partir d'un modèle UML plusieurs alternatives de structuration pour Cassandra, Neo4J et MongoDB. Pour cela, ils utilisent une approche MDA [53]. Ils proposent de transformer d'abord le modèle UML dans un modèle logique générique. Ce modèle est basé sur un méta-modèle qui contient les concepts communs aux trois systèmes : orientés colonnes, orientés documents et orientés graphes. Ensuite, une phase de transformation utilise le modèle générique et crée des alternatives de structuration pour chaque système NoSQL. Ils proposent des règles de transformation d'une association selon chaque système. Pour Cassandra, des solutions sont proposées : soit l'information de chaque table correspondant aux extrêmes de l'association peut être référencée soit l'association conduit à la création d'une famille de colonnes. Pour MongoDB, 5 solutions sont proposes pour structurer l'association : par classe une collection en référençant les documents de la collection partenaire, par classe une collection en imbriquant les documents correspondant la classe partenaire et une collection avec les références aux deux collections.

Atzeni et al. [8] [54] proposent l'approche NoAM comme une méthode de conception de bases de données pour divers systèmes NoSQL tels que clé-valeur (Oracle NoSQL), orientés documents (MongoDB) et orientés colonnes (DynamoDB). D'abord, une modélisation conceptuelle des données et une conception des agrégations sont mises en place. Ensuite, une analyse de haut niveau permet de décomposer les agrégations dans des éléments plus petits qui sont projetés sur un modèle intermédiaire appelé NoAM. Enfin, les éléments de ce modèle NoAM sont interprétés dans les éléments de modélisations spécifiques du système NoSQL choisi. Le modèle de données NoAM définit une base de données comme un ensemble de collections avec un nom différent. Une collection est un ensemble de blocs identifiés par une clé unique. Un bloc est un ensemble non vide d'entrées sous forme clé :valeur.

Synthèse
La table 2.1 synthétise les travaux présentés dans cette section. Notons que les auteurs se rejoignent dans le choix de l'utilisation des concepts Entité-Relation pour le modèle conceptuel. Les systèmes NoSQL prioritaires sont MongoDB et Cassandra. La motivation de la plupart de ces travaux est le choix, presque ad-hoc, d'une structure de données favorable aux performances d'exécution d'un ensemble de requêtes donné. La validation de ces travaux repose essentiellement sur le temps de réponse des requêtes. Les propositions de Abdelhedi et al. [52] et Atzeni et al. [8], proposent une approche plus riche que les autres et ciblent plusieurs systèmes avec des modèles de données différents. Ces travaux s'appuient sur un méta-modèle pour représenter les caractéristiques des modèles de données NoSQL.

Proposition	Approche conceptuelle	Système NoSQL cible	Objectif recherché	Critères pris en compte	Nb de structures proposés	Caractéristiques dominantes des structures NoSQL créées	Critère de validation de la structure proposée
Zhao [45]	ER	MongoDB (documents)	Proposer une structure qui matérialise les associations entre entités par imbrication de documents	- Modèle de données ER	1	Collection de documents par type d'entité avec imbrication récursive des entités référencées dans le modèle ER	Temps de réponse des requêtes Taille des données redondantes
Lombardo [47]	ER	Cassandra (colonnes)	Choisir une structure ad-hoc pour l'évaluation performante d'un ensemble de requêtes	- Modèle de données ER - Ensemble de requêtes (langage navigation)	1	Tables imbriquées	Temps de réponse des requêtes
Mior [49]	ER	Cassandra (colonnes)	Choisir une structure ad-hoc pour l'évaluation performante d'un ensemble de requêtes	- Modèle de données ER - Ensemble de requêtes (langage navigation) - Fréquence des requêtes	1	Tables imbriquées	Temps de réponse des requêtes
Varjk [48]	DSL	Cassandra (colonnes)	Choisir une structure ad-hoc pour l'évaluation performante d'un ensemble de requêtes	- Modèle de données ER - Ensemble de requêtes (OCL)	1	Tables imbriquées	Temps de réponse des requêtes
Abdelhedi [52]	UML et Utilisation d'un méta-modèle pour représenter 3 modèles NoSQL cibles	MongoDB Cassandra Neo4J (documents, colonnes, graphes)	Générer plusieurs structures pour des systèmes NoSQL différents	- Modèle de données UML	MongoDB : 5 Cassandra : 3 Neo4J :1	*MongoDB* : Collection de documents par classe avec référencement ou imbrication récursive des classes. *Cassandra* : Table par classe avec imbrication ou référencement *Neo4J* : Noeud par classe, les propriétés correspondent aux attributs de la classe et les arc aux associations	Temps de réponse des requêtes
Atzeni [8]	ER et Utilisation d'un méta-modèle pour représenter 3 modèles NoSQL cibles	MongoDB Cassandra (documents, colonnes, key-value)	Générer une structure de données pour de systèmes NoSQL différents afin de améliorer les performances d'un ensemble de requêtes	- Modèle de données ER - Ensemble de requêtes	MongoDB : 1 Cassandra : 1	*MongoDB* : Collections avec des documents imbriqués. *Cassandra* : Tables imbriquées	Temps de réponse des requêtes

TABLE 2.1: Synthèse des approches conceptuelles

2.4 Vers des approches d'abstraction des structures

Des travaux récents s'intéressent à l'analyse du schéma d'une base de documents déjà implémentée. Leur motivation est d'aider à comprendre la structuration des données et à expliquer des variantes. Dans [55] [56] les auteurs introduisent la gestion d'un schéma au-dessus des systèmes de documents sans schéma (schema-less).

Ruiz et al. [56] proposent une stratégie de rétro-ingénierie dans le but de déduire le schéma implicite d'une base orientée documents déjà mise en œuvre comme dans MongoDB, CouchDB et HBase. L'approche propose de découvrir les différentes versions existantes d'une entité et de ses associations. La solution considère trois étapes basées sur les techniques MDE [57]. Initialement, une collection d'objets JSON est obtenue, et chaque objet est considéré comme une version d'une entité. Ces objets sont extraits de la base à l'aide d'une opération MapReduce. Ensuite, la collection est transformée dans un modèle correspondant au métamodèle JSON construit à partir de sa grammaire. Dans la troisième étape, ce modèle est transformé dans un modèle correspondant le métamodèle nommé NoSQL-Schema. Le métamodèle NoSQL-Schema considère des concepts des bases orientées documents ainsi que des concepts pour gérer les entités, les attributs, les associations et les versions. Ce modèle peut être utilisé pour créer des utilitaires de bases de données ou pour guider les développeurs en raison de l'absence de schéma.

Gallinucci et al. [7] proposent d'étudier les structures des documents dans le but de créer des profils des documents présents dans une collection. Cela permet de donner une classification qui montre combien de structures différentes sont adoptées par les documents d'une collection. Ils proposent d'identifier des conditions d'existence d'attributs et des conditions de valeur. Les conditions d'existence d'un attribut permettent de mettre en évidence les attributs qui ne sont pas présents dans tous les documents considérés d'un certain type. Les conditions de valeur, analysent l'ensemble de valeurs d'un attribut et tentent d'inférer des caractéristiques du domaine de valeurs. Par exemple, pour un attribut entier, l'approche permet de dériver l'intervalle des valeurs utilisées. Pour des chaînes de caractères, on peut avoir un type énuméré de valeurs. Les auteurs adoptent une notion d'entropie d'une collection qui reflète la diversité des structures utilisées par les documents. Il ne s'agit pas ici de dire seulement combien de structures différentes sont utilisées par les documents mais aussi de prendre en compte la différence entre ces structures. Cette différence est basée sur la présence ou non du même ensemble d'attributs et leur divergence ou non sur leur type.

Wang et al. [55] s'intéressent également à l'extraction d'éléments permettant d'apprécier la structure de documents JSON. Ils proposent un framework de gestion de schémas qui matérialise les attributs présents dans les objets. Les attributs identifiés sont analysés pour faire une catégorisation

et mettre en évidence des noyaux d'attributs (nommés skeleton) et des chemins d'imbrication d'attributs. Le type des attributs n'est pas géré. Les informations extraites sont représentées sous forme d'arbre et stockées de manière à pouvoir être consultées.

Klettke et al. [58] ont également fait un travail d'extraction de structure de documents JSON. Ils identifient les attributs, leur nature complexe (si c'est un array) ou atomique. Pour une collection, ils identifient si elle est homogène au sens que les documents ont tous la même structure ou si elle est hétérogène.

Dans les outils existants, notons MongoDBCompass [59] et JSON schema [60]. MongoDBCompass opère sur des base MongoDB. Concernant la structuration des données, cet outil permet d'obtenir des informations sur les types des documents. Plus précisément, il permet d'afficher des statistiques sur l'apparition d'un attribut dans les documents d'une collection. L'outil infère le type de l'attribut en se basant sur le type de la valeur stockée. Tenant compte du fait que les documents peuvent avoir des structures différentes, pour un même attribut, il est possible de trouver plusieurs couples (nom d'attribut, type de la valeur).

JSON schema [61] est un format qui permet de décrire la structure d'objets JSON en utilisant sa grammaire. Avec JSON schema, il est possible de définir les attributs contenus dans les objets JSON et leurs types. Chaque attribut est défini comme une paire *nomAttribut :objet*. Cet objet peut contenir plusieurs paires *clé :valeur* définissant entre autres le type de l'attribut, exemple ("type" : "String"), une valeur par défaut pour l'instance et une description entre autres. Il est possible d'inclure des méta-données pour définir la version du schéma, son nom, les attributs obligatoires, l'approbation des attributs supplémentaires dans les instances et la définition des patrons d'accès, entre autres. Certains outils comme JSON-schema [60], quicktype [60] et Schema Guru [60] analysent des documents JSON dans le but d'abstraire la structure en suivant les lignes directrices fournies par la spécification.

Synthèse
La table 2.2 récapitule les travaux de retro-ingénierie pour obtenir des informations sur la structuration des données. Ils portent essentiellement sur des documents style JSON. Ces travaux utilisent des terminologies différentes mais les concepts sont proches. Ils extraient les attributs utilisés dans les documents et tentent d'identifier leur type. Tenant compte le caractère semi-structuré et l'hétérogénéité des documents, chaque proposition extrait des informations qui reflètent les variantes présentes dans la base.

Proposition	Type de données à analyser	Objectifs	Structure reconstruite
Ruiz [56]	Base orientée document	- Identifier les structures des données - Découvrir les différentes versions de structure des documents d'une collection	Graphe représentant la structure : - Entités et ses attributs
Gallinucci [7]	Base orientée document	- Identifier les structures des données - Créer des profils de documents	Graphe représentant la structure : - Existence ou absence d'attributs - Caractérisation du domaine de valeurs
Wang [55]	Base orientée document	- Identifier les structures des données - Découvrir les noyaux d'attributs (Skeleton)	Graphe représentant la structure : - Existence ou absence d'attributs - Noyau d'attributs - Chemins d'imbrication
Klettke [58]	Documents JSON	- Identifier les structures des données - Découvrir les variantes de structure des documents	Graphe représentant la structure : - Existence ou absence d'attributs - Caractérisation du domaine de valeurs JSON schema
MongoCompass [59]	Base orientée document	Extraire des statistiques sur les documents d'une collection	Catalogue des caractéristiques structurelles : - Existence ou absence d'attributs - Caractérisation du domaine de valeurs
JSONSchema [60]	Document JSON	Abstraire la structure d'un objet JSON en suivant les lignes directrices de la spécification	JSON schema : - Noms et types d'attributs - Chemins d'imbrication

TABLE 2.2: Synthèse des approches d'abstraction des structures

2.5 Conclusion

Nous avons abordé diverses approches NoSQL. Nous avons particulièrement étudié les bases orientés document selon l'angle de la structuration des données. La flexibilité de ce modèle de données ouvre la porte à de multiples possibilités de structuration. Cette flexibilité est accentuée par le fait que la plupart des systèmes ne gèrent pas de schéma. Maintenant qu'il y de nombreuses bases de documents implémentées, notamment sur MongoDB, et créées sans la préoccupation des impacts de la structuration des données, les difficultés de gestion des bases et des programmes commencent à apparaître.

Nous avons passé en revue certains travaux récents qui cherchent, comme nous, à améliorer les questions autour de la modélisation des données semi-structurées. Certains travaux portent sur des approches d'ingénierie inverse pour extraire des caractéristiques des structures de bases de documents existants alors que d'autres cherchent à encourager une modélisation en amont. Ces travaux se basent sur un modèle conceptuel et produisent un schéma semi-structuré unique. Les alternatives de structuration sont écartées et ne sont pas présentées aux utilisateurs. Ceci restreint l'exploitation des possibilités offertes par la flexibilité des systèmes orientés documents.

Les besoins de pouvoir compter sur des informations de structuration des bases orientés documents ressortent dans plusieurs travaux. Ainsi des systèmes tels qu'Asterix offrent la possibilité d'introduire une description de schéma et d'effectuer une vérification des types de documents sans pour autant rendre obligatoire et stricte cette structuration. Les efforts autour de JSON Schema vont aussi dans ce sens.

Tous ces travaux reflètent globalement le besoin d'améliorer et de formaliser l'analyse des données afin de mieux maîtriser leur structuration et l'impact sur les systèmes d'information qui les utilisent. Il y a très peu de propositions de critères objectifs pour faciliter l'évaluation et la comparaison d'alternatives de structuration. Dans le chapitre suivant, nous poursuivons l'étude de l'état de l'art, en abordant les travaux proposant des métriques dans divers domaines.

"La persévérance, c'est ce qui rend l'impossible possible,
le possible probable et le probable réalisé."

Léon Trotsky

3

Métriques des données et des logiciels

.

3.1 Introduction

Dans le chapitre précédant nous nous sommes intéressés aux systèmes NoSQL et plus particulièrement aux approches orientés document, i.e. JSON et MongoDB. Dans cette thèse nous nous intéressons aux questions de qualité des structurations des données dans ces bases et leurs retombées sur les logiciels. Notre approche s'appuie sur la proposition d'un cadre aidant à l'analyse des alternatives de structures. Pour cela nous avons travaillé sur la proposition d'éléments formels qui aident le développeur. Comme nous le verrons par la suite, nous proposons des métriques structurelles.

Dans ce chapitre nous passons en revue des travaux liés aux métriques dans divers cadres. Précisons que nous ne nous intéressons pas aux questions de qualité des données au sens de leur précision ou correction, mais à la qualité des structures de documents.

Dans le contexte des bases de données relationnelles, les travaux sur la normalisation ont joué un rôle extrêmement important dans la conception des schémas. Cette théorie donne un cadre formel qui permet d'identifier des anomalies d'un schéma affectant l'insertion, la suppression et la mise à jour des données.

[62] s'est intéressé aux questions de qualité dans les schémas conceptuels basés sur un modèle entité-association avec des extensions orientées objet. Bien que [62] ne présente pas des résultats très complets, cet article nous semble important par son approche pour tenter de mesurer la qualité d'un schéma conceptuel. L'auteur présente des critères tels que la pertinence et la complètude d'un schéma par rapport au monde applicatif, ainsi que sa facilité de lecture (*readability*) par la clarté du vocabulaire utilisé.

Dans la suite nous présentons, dans un premier temps, des travaux sur l'évaluation de performances opérationnelles des systèmes NoSQL. Ils concernent, entre autres, la taille des bases et le temps de réponse des requêtes. Nous nous focalisons sur les travaux qui analysent les performances et font un lien avec des aspects structurels des données. Ensuite, dans la section 3.3 nous regardons plus largement des travaux sur les métriques structurelles. La littérature étudiée concerne des propositions pour des documents JSON, XML, et des graphes. La section 3.4 aborde des travaux dans le domaine du génie logiciel où les questions de qualité ont été largement étudiées. Nous nous intéresserons particulièrement à des travaux sur les métriques statiques du code. La section 3.5 conclut ce chapitre.

3.2 Métriques de performance dans des systèmes NoSQL

Les systèmes NoSQL ont été particulièrement motivés par des questions de performances et de passage à l'échelle. De nombreux efforts ont été faits pour des évaluation de performances dans le but de mettre en avant un type de système, ou un autre, selon le profil des applications et l'infrastructure matérielle choisie. Dans cette section nous restreignons notre étude à des travaux permettant de faire un lien avec la structuration des données qui est le sujet de notre thèse.

Nos considérons ici des travaux pour des systèmes orientés colonnes et orientés documents. Les performances les plus étudiées, de manière générale, sont la taille des bases et des indicateurs de temps ou de nombre d'opérations pour l'accès aux données.

Systèmes orientés colonnes
Nous revenons ici sur certains des travaux déjà évoqués dans la section 2.3.2. Ils ont fait des propositions pour aider au choix des structures dans Cassandra. Nous rappelons que le modèle de données est Big Table et que les requêtes sont programmées avec des primitives GET et SET.

Dans les travaux de Mior [46] les métriques considérées sont la taille du stockage et le nombre d'opérations SET et GET nécessaires pour évaluer les requêtes. L'objectif recherché est des trouver un compromis de réduction de taille de la base et optimisation des requêtes en terme de nombre d'opérations.

Les travaux de Lombardo et al. [47] qui concernent également la modélisation dans Cassandra, ne proposent pas de métriques. Les travaux de [52] sont multi-plateformes, et incluent Cassandra, Neo4J et MongoDB. Leur évaluation de performances porte sur le temps de réponse des requêtes.

Systèmes orientés documents
Dans les systèmes orientés documents, nous avons déjà évoqué en section 2.3.2 les travaux de Zhao [45] pour la création de bases MongoDB. Les métriques utilisées dans ce travail concernent principalement la taille de la base et la redondance des données dans la base.

MongoDBCompass, outil proposé par MongoDB, permet de surveiller le temps d'exécution des requêtes et de connaître le volume des données. Il propose également des informations structurelles présentées ci-après.

Certains travaux [63, 64, 65]. comparent les performances, l'évolutivité et d'autres propriétés entre les systèmes relationnels et NoSQL.

3.3 Métriques de structure pour de données semi-structurées

Cette section porte sur des travaux qu'introduisent des métriques structurelles pour des données semi-structurées. Dans un premier temps nous aborderons des caractérisations des graphes puis nous présenterons des propositions faites sur des documents XML et JSON.

3.3.1 Métriques des graphes

Plusieurs modèles de données s'appuient sur des représentations sous forme de graphes et bénéficient des travaux sur la théorie des graphes [66, 67, 68, 69, 70, 71]. On retrouve des représentations avec des graphes orientés et des graphes non orientés.

Un graphe orienté, G=(N,A), est un couple formé d'un ensemble N de nœuds et un ensemble A d'arcs. Chaque arc étant un couple de nœuds (x,y). On dit que x est l'origine ou la source de l'arc et que en y est l'extrémité ou la cible. Le degré sortant d'un nœud, est le nombre d'arcs ayant ce nœud pour origine. Le degré entrant d'un nœud, est le nombre d'arcs ayant ce nœud pour extrémité. Un graphe non orienté est un graphe dont les arêtes n'ont pas d'orientation. Une arête est une paire non ordonnée de sommets.

Dans la gestion des données il est couramment nécessaire de savoir si un graphe est simple, connexe ou un arbre. Un graphe simple est un graphe non orienté sans arêtes multiples et sans boucle. Un graphe est connexe s'il existe un chemin entre tout couple de sommets. Parfois il est intéressant de déterminer les nœuds qui sont des points d'articulation. Un nœud est dit d'articulation si le sous-graphe obtenu en le supprimant n'est pas connexe. Un arbre est un graphe connexe et acyclique. Il est aussi courant de travailler avec des arborescences qui sont des graphes acycliques orientés où l'on distingue un nœud racine de degré entrant nul, et où tous les autres nœuds sont de degré entrant 1.

Dans la caractérisation des structures et les métriques, l'on manipule les degrés entrants et sortants des nœuds (par exemple pour déterminer le nombre de "feuilles") mais aussi des informations liées aux chemins entre les nœuds.

Un chemin est une suite finie non vide de sommets telle que chaque couple de sommets consécutifs de la suite soit un arc du graphe. La notion correspondante dans un graphe non orienté est celle d'une chaîne. La longueur d'un chemin est celle de sa suite d'arcs. La distance entre les nœuds x

et y est la longueur du plus court chemin (respectivement plus courte chaîne) entre ces sommets.

Un chemin est dit élémentaire si aucun des sommets ne figure plus d'une fois comme source, ni plus d'une fois comme but d'un arc du chemin. Un chemin est dit simple si il ne passe pas deux fois par un même arc (tous les arcs sont distincts). Un circuit (ou cycle) est un chemin dont les deux extrémités sont identiques.

Dans les métriques sur les graphes significatives dans la gestion de données, on trouve notamment le nombre de chemins simples, de chemins élémentaires, de circuits et leur longueur. Pour le calcul de ces métriques, les systèmes peuvent s'appuyer sur une représentation des graphes par matrice d'incidence.

3.3.2 Métriques pour XML et JSON

Dans cette section, nous nous intéressons aux métriques structurelles des documents type XML et JSON. La plupart des travaux étudiés portent sur XML, mais ils nous ont servi dans le cadre de cette thèse.

Klettke et al. [72] s'appuie sur le modèle de qualité de software ISO 9126 et adapte cinq métriques pour évaluer des documents XML. Ces travaux tirent profit de l'existence de la DTD qui est manipulée sous forme de graphe. Les métriques proposées sont le nombre d'éléments définissant la DTD, le nombre d'arcs sortant d'un nœud (*fan-in*), le nombre d'arcs entrant d'un nœud (*fan-out*) et la profondeur maximale. Une métrique de complexité de la DTD est utilisée en se basant sur la complexité du graphe en termes de nombre de nœuds et d'arcs.

Les travaux de Pušnik et al. [73] portent aussi sur XML. Ils proposent six métriques associées chacune à un aspect tels que la structure, la clarté, l'optimalité, le minimalisme, la réutilisation et la flexibilité. Ces métriques utilisent 25 variables qui mesurent, entre autres, le nombre d'éléments, d'annotations, de références et de types.

MongoDBCompass travaille sur des bases MongoDB opérationnelles. En plus des informations sur la taille des données déjà mentionnées, cet outil permet d'afficher des informations sur les types des attributs des documents. Il donne notamment la liste des attributs présents dans les documents, leur type et permet de visualiser une synthèse qui précise le pourcentage de documents qui possède un certain attribut. Cette dernière information peut être complétée par l'information sur l'utilisation de différents types pour un même attribut. Par exemple, un attribut X qui apparaît dans certains documents comme un integer et dans d'autres comme une chaîne de caractères.

MongoCompass calcule quel pourcentage de documents a l'attribut X et quelle partie est d'un type ou de l'autre.

Dans les travaux de recherche sur les approches orientées documents, certains efforts déjà mentionnés, suivent une approche d'ingénierie inverse pour inférer un schéma, avec les types de données, à partir de bases de documents opérationnelles. Dans le travail de Ruiz et al. [56], l'utilisateur peut ensuite savoir si un attribut existe et son type. Ils utilisent une notion de version pour gérer les cas où divers types sont utilisés. Un type "null" est introduit si certains objets ne possèdent pas l'attribut alors qu'il est supposé exister.

Wang [55] travaille aussi dans l'abstraction de schémas. Pour caractériser les éléments manipulés, ils introduisent une notion de type d'objet et identifient l'existence de collections d'un certain type d'objet. La présence ou l'absence d'un attribut supposé être présent est indiquée.

Gallinucci et al. [7] proposent de créer des profils des documents présents dans une collection. Les auteurs adoptent une notion d'entropie d'une collection qui reflète la diversité des structures utilisées par les documents.

Suite à l'abstraction du schéma, Klettke et al. [58] fournissent des statistiques et calculent des métriques. Ils proposent la détection des erreurs possibles telles que des structures de documents créés par erreur dont elles apparaissent très peu et des propriétés avec des noms similaires qui peuvent faire référence aux même informations. Ils proposent également des informations statistiques par rapport au pourcentage de documents contenant une propriété et aux propriétés perdues. Ces dernières font référence aux propriétés absentes dans un tout petit nombre de documents de la collection. Afin de mesurer la variabilité, ils se sont intéressés par le caractère optionnel d'une propriété et à l'homogénéité des documents. Tel que Gallinucci avec sa métrique d'entropie [7], les auteurs proposent de mesurer le degré de couverture des documents *cov*. Ceci détermine l'homogénéité des documents d'une collection, *cov=0* indique que les documents sont complètement hétérogènes et *cov=1* que tous les documents ont les mêmes propriétés. Concernant le caractère optionnel, les auteurs proposent des métriques pour identifier les propriétés facultatives et obligatoires des documents d'une collection.

3.4 Métriques en génie logiciel

De nombreux efforts sont faits dans le domaine de génie logiciel pour aider à la qualité des logiciels. Ces travaux couvrent un spectre large de la vie des logiciels en incluant leur conception,

développement, mise au point et évolution. Divers outils aident au développement de logiciels de qualité dont les métriques qui nous occupent ici. Parmi les métriques proposées, notons les métriques dites statiques ou structurelles, les métriques dynamiques et celles qui caractérisent les processus des utilisateurs. Le lecteur peut se référer à [74, 75] pour une présentation globale.

Dans cette section nous abordons essentiellement les métriques statiques qui concernent la structure des programmes. Ces mesures sont très importantes car elles montrent des caractéristiques du code qui peuvent avoir une incidence sur leur clarté et la maintenance. L'objectif recherché est de produire des programmes qui pourront être compris correctement par les développeurs afin de les faire évoluer avec un effort adéquat. Un code clair, bien structuré et facile à lire pourra être modifié plus facilement et réduira les risques d'erreurs.

Il y a des métriques standard telles que la complexité cyclomatique ou la mesure de McCabe [76] mais leur calcul est dépendant du langage de programmation. Divers outils existent aujourd'hui destinés à calculer des métriques statiques pour des programmes écrits dans certains langages. La complexité du code est mesurée selon divers aspects. Parmi eux, l'imbrication des instructions de contrôle, la complexité des hiérarchies des classes et la cohésion d'un composant (ou classe). Au sujet des instructions de contrôle, il s'agit de voir le nombre d'imbrications de branchements du flux d'exécution dans un programme. Lorsque ce nombre est très élevé, les performances du programme s'en ressentent ainsi que la clarté du code. Un tel code sera plus difficile à maintenir et à faire évoluer.

Chidamber [77] propose de mesurer la cohésion de chaque « module » et le couplage entre les « modules » . Selon le contexte du logiciel considéré, un « module » peut désigner une méthode, une classe, un fichier ou encore un « composant ». La cohésion d'un module, cherche à répondre à la question du pourquoi ces éléments sont groupés dans ce module. Alors que le couplage porte sur les liens entre les modules. Une grande cohésion facilite la recherche du module à utiliser (ou à modifier) et concentre l'ensemble des opérations dans un minimum de modules. Un faible couplage entre deux modules limite l'effet de l'évolution d'un module sur l'autre.

Dans [78, 77, 76], les auteurs travaillent sur des métriques de logiciel avec le paradigme procédural et le paradigme orienté objet. Plusieurs métriques sont proposées pour refléter notamment les niveaux de couplage entre composants et classes, la taille des objets, des hiérarchies de classes et le nombre de méthodes.

Il serait donc intéressant de maximiser la cohésion et de minimiser le couplage afin de faciliter la compréhension et l'évolution des modules. Néanmoins, un couplage trop faible pourrait induire des problèmes de performance à l'exécution. Il est donc important d'analyser les métriques de manière globale en considérant de multiples métriques statiques mais aussi des métriques

dynamiques.

Comme nous verrons dans la suite, ceci est également vrai pour les applications utilisant des bases orientées documents. Cela apparaît clairement dans notre étude expérimentale et nous a conduit à la proposition de métriques structurelles. Nous avons étendu et adapté des propositions existantes pour notamment prendre en compte les particularités d'imbrication de documents et types d'attributs JSON en vue d'une utilisation dans une base de documents telle que MongoDB.

Notons, pour finir, que l'analyse statique du code a pris beaucoup d'importance grâce aux outils de qualité qui sont disponibles. C'est le cas de SonarQube [1] [79] qui analyse statiquement le code et produit des tableaux de bord récapitulant de nombreux indicateurs de qualité. Parmi eux, la vérification de normes de codage, des métriques du code (par exemple la complexité de McCabe), des indicateurs de points susceptibles d'être problématiques pour le maintien du code ainsi que des vulnérabilités de sécurité. SonarQube estime une *Technical Debt* qui indique le temps nécessaire pour corriger les problèmes de maintenabilité repérés dans le code.

3.5 Conclusion

Dans ce chapitre nous avons étudié des propositions des métriques dans le domaine de la gestion de données et du génie logiciel. Dans le domaine du génie logiciel les travaux sur les métriques sont très riches. Nous nous sommes particulièrement intéressés aux métriques statiques étant donnée l'orientation de nos recherches sur l'impact des structures des données. Nous notons particulièrement les métriques de cohésion et de couplage, largement adoptées en génie logiciel et qui nous semblent intéressantes pour les approches orientées document.

Concernant les données semi-structurées nous avons étudié des propositions sur les graphes et les documents aux structures arborescentes tels que XML. Ces métriques considèrent des aspects tels que l'existence d'éléments et l'identification de la complexité des structures. Ces métriques de complexité de structures sont un point de départ pour nos propositions.

Dans le domaine des systèmes de gestion de données l'accent est souvent mis sur les performances de l'exécution des requêtes. Ceci est particulièrement notable pour les systèmes NoSQL pour cause de l'absence de schéma de données. Nous avons néanmoins noté certains efforts pour mesurer des caractéristiques des structures utilisées dans des bases MongoDB opérationnelles. Les propositions portent sur l'analyse des structures afin d'extraire des informations sur la présence

1. Précédemment appelé Sonar

d'un attribut dans un document et sur des inférences de type des attributs. Quant aux travaux orientés modélisation conceptuelle, leur validation est ramenée à des études de performances de l'exécution des requêtes sans donner des indicateurs d'évaluation des structures. La validation des performances nécessite bien entendu la création et la population des bases de documents.

Notre étude montre qu'il n'y pas de proposition d'un ensemble formel de métriques qui aide à refléter la complexité globale de la structure de la base. Il n'y a pas non plus des propositions qui facilitent la comparaison de structures dans le but d'assister le développeur dans ses choix et de soutenir les analyses en amont, avant la création des bases.

"La vie, c'est comme une bicyclette,
il faut avancer pour ne pas perdre l'équilibre."

Albert Einstein

4

De l'importance de la structuration de données dans les systèmes NoSQL, un regard expérimental

.

4.1 Introduction

La modélisation des données a un impact sur la performance de l'interrogation, la cohérence, la facilité d'utilisation, la mise au point du logiciel et la maintenabilité, et bien d'autres aspects [16]. Quelle serait la *bonne* structuration des données dans les systèmes NoSQL ? Quel est le prix à payer par les développeurs et les utilisateurs pour une structuration de données flexible ? L'impact de la structuration des données sur la complexité de l'interrogation est-il élevé ? Afin de mieux aborder ces questions et de bien apprécier l'utilisation des bases orientées document, nous avons mené une étude expérimentale.

Ce chapitre présente une analyse de l'impact de la structuration des données sur la *taille des bases*, les *performances* et la programmation des requêtes. Cette analyse s'appuie sur une expérience que nous avons menée pour comparer des alternatives de structuration de données semi-structurées en utilisant un ensemble de requêtes représentatives. Le modèle de données choisi est orienté document et les alternatives de structuration varient donc principalement sur leurs structures imbriquées. Par rapport aux requêtes, nous considérons des chemins d'accès différents et une complexité croissante. Au cours de l'expérience, nous analysons aussi l'impact de l'utilisation des index dans les collections. Nous avons effectué l'expérience et l'analyse en utilisant le système MongoDB.

Comme nous l'avons évoqué dans le chapitre 2, le système MongoDB est basé sur le format JSON. Ce système ne nécessite pas la définition d'un schéma explicite de base de données et les contraintes d'intégrité ne sont pas supportées. Il offre une grande flexibilité de modélisation des données, car de nombreuses alternatives de structuration avec les collections et ses documents sont possibles. Le système des types permet deux modes de mise en relation des documents, *intégration* ou *référencement*. Cela permet de nombreuses possibilités de « modélisation ».

Dans ce chapitre, nous verrons que si la tendance NoSQL implique et pousse l'utilisation de bases de données semi-structurées sans schéma, il existe néanmoins une structuration «implicite» et son choix est important car les implications à divers niveaux peuvent avoir des coûts inattendus ou inconnus.

Dans la section 4.2, nous définissons des différentes alternatives de structuration représentatives pour les mêmes données. Une étude des performances liées au choix de structuration de donnes est réalisée dans la section 4.3. Enfin, dans la section 4.4 nous discutons les résultats des expériences en mettent en évidence si ces résultats confirment ou non l'impact attendu des choix des structures.

4.2 Alternatives de structuration des données semi-structurées

Cette section explore les possibilités de structuration des données qui sont possibles dans MongoDB afin d'analyser leurs avantages et leurs inconvénients possibles en fonction des besoins de l'application. Nous utilisons un exemple simple nommé *l'entreprise*. La Figure 4.1 montre le modèle UML qui implique des entreprises, des départements et des employés. Les associations sont 1 .. * avec la sémantique habituelle. Une entreprise a plusieurs départements. Un département appartient à une entreprise et il a plusieurs employés. Chaque employé travaille pour un seul département. L'exemple choisi n'a aucune situation d'associations multiples entre deux classes. Ce choix nous permet néanmoins de mettre en évidence un grand nombre d'alternatives de structuration.

Nous avons donc créé six bases de données MongoDB pour le cas `l'entreprise`. Naturellement, il est toujours possible de créer d'autres bases de données pour cet exemple en considérant le regroupement des documents selon les associations du modèle UML.

Comme il n'y a pas de définition de schéma d'une base de données dans MongoDB, chaque base de données est un ensemble de collections qui contiennent des documents en fonction de différentes structurations des données. La Figure 4.2 illustre les alternatives de structurations. Les structurations de S1 à S6 correspondent aux six bases de données, appelées bS1 à bS6, sur lesquelles porte notre expérimentation. Un cercle extérieur représente les documents au premier niveau d'une collection alors que les rectangles internes correspondent à une hiérarchie des documents ou tableau de documents. Seulement les identifiants des documents sont présentés dans les cercles. La Figure 4.3 introduite un exemple de données contenues dans les base bS2 et bS5 correspondant les représentations S2 et S5 de la Figure 4.2. Ici, toutes les données sont incluses.

En considérant l'imbrication des documents et des ensembles de documents comme des caractéristiques particulières, nous avons proposé des structurations de documents différentes pour chaque base dans lesquelles on mélange l'existence de collections, l'imbrication, le référencèrent et la duplication en conduisant ainsi à différents niveaux d'accès et plusieurs copies d'un document. Nous présentons ensuite une caractérisation des structurations des données considérées en tenant en compte ces aspects.

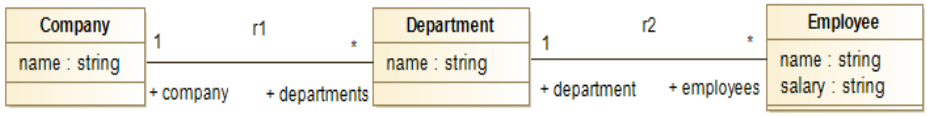


FIGURE 4.1: Diagramme UML dans un contexte d'entreprise

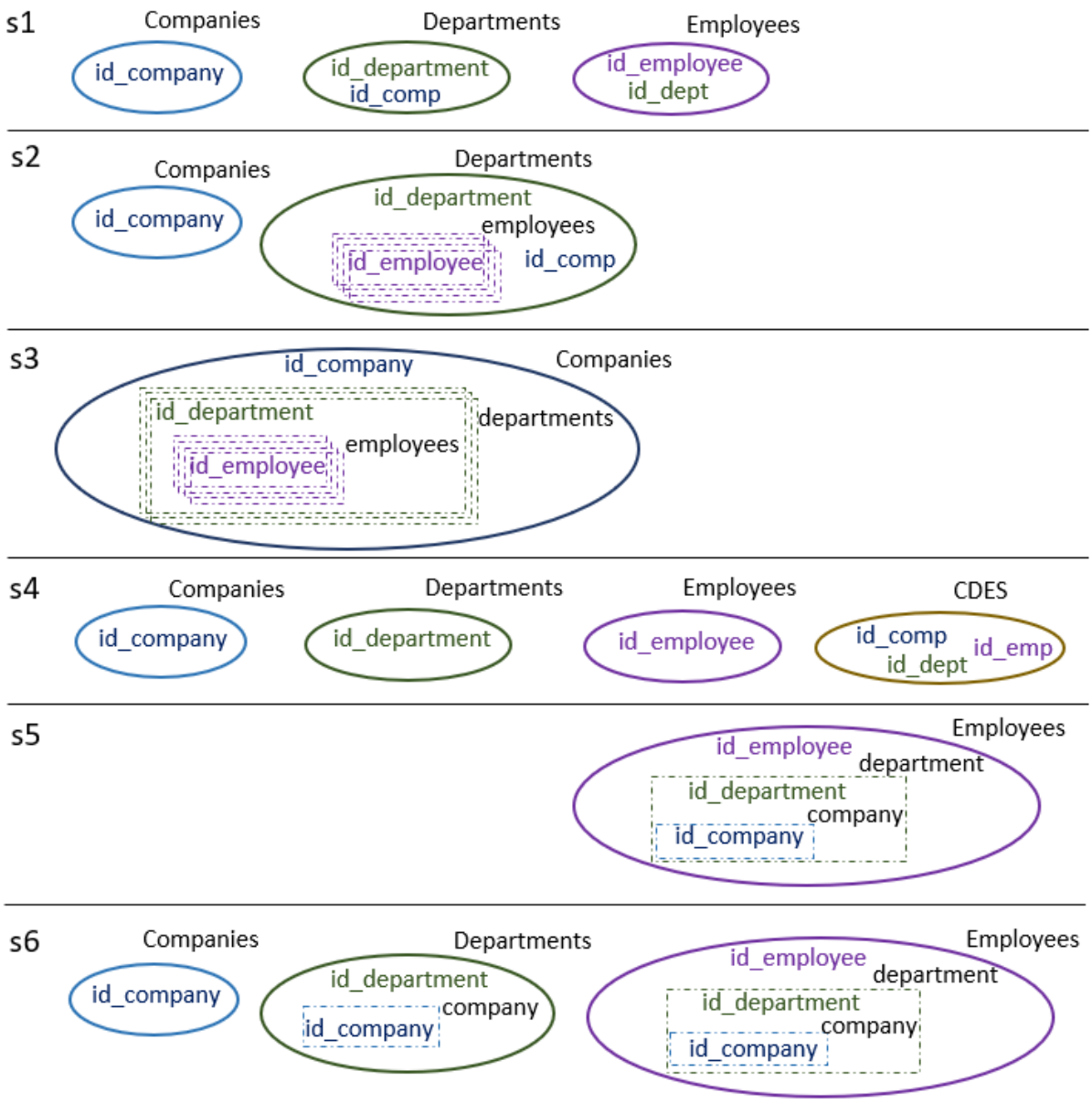
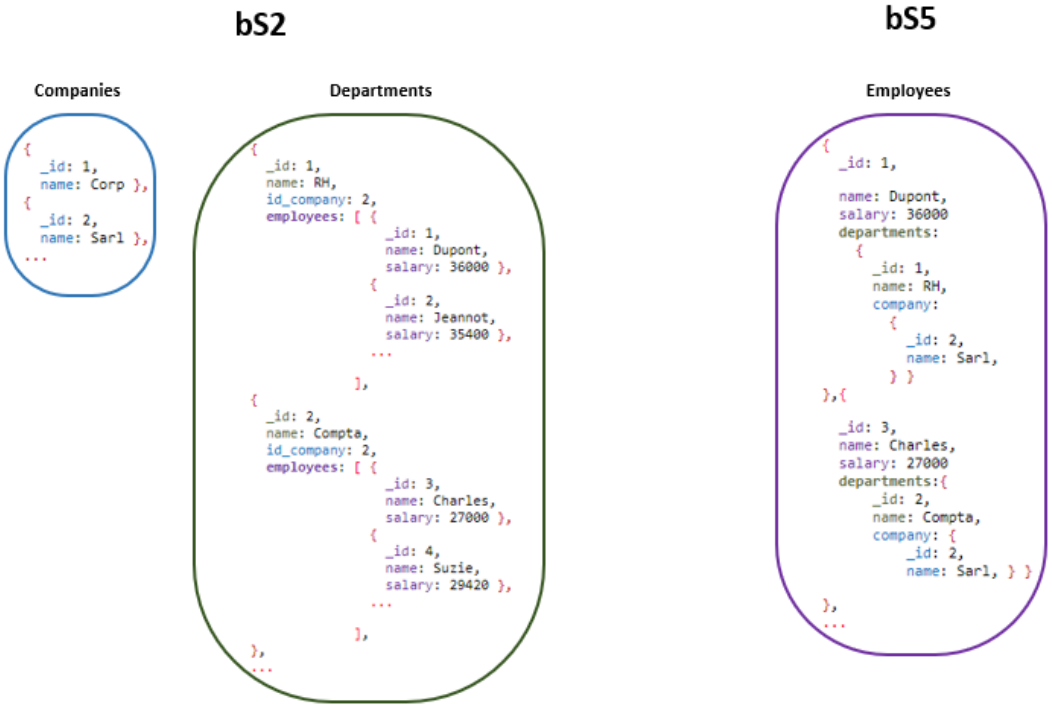


FIGURE 4.2: Représentation des collections créées dans 6 bases MongoDB



a) Données dans la base bS2 b) Données dans la base bS5

FIGURE 4.3: Exemple des données pour les bases MongoDB bS2 et bS5 correspondant aux représentations S2 et S5 de la Figure 4.2

Absence d'imbrication & référencement

Dans les bases bS1 et bS4, chaque classe du modèle UML est une collection séparée. Chaque document a un champ additionnel qui est l'identifiant du document imposé par MongoDB. Ces identifiants sont utilisés pour les références entre documents afin de représenter les associations, ceci est similaire aux systèmes relationnels normalisés avec les clés primaires et les clés étrangères. Dans bS1, les références sont utilisées dans les collections dont la classe correspondante dans le modèle UML a une association avec une cardinalité *plusieurs* de son côté. Dans bS4, les références se placent dans une collection supplementaire nommée `CDES`. Dans ces bases, les documents ne sont pas imbriqués.

Imbrication totale & collection unique

Dans les bases bS3 et bS5, toutes les associations sont matérialisées avec des documents imbriqués. En considérant le modèle UML de notre cas de étude, la traversée des associations conduit à une collection unique avec des documents imbriqués à 3 niveaux. Dans bS3, l'information

concernant les entreprises se trouve au premier niveau de la collection `Companies` (niveau 0). Chaque document imbrique *n* documents `departement` (niveau 1). À son tour, chaque document `departement` imbrique *n* documents `employee` (niveau 2). Le choix de l'imbrication dans bS3 reflète la direction *1 à plusieurs* des associations. Dans bS5, le raisonnement est similaire, mais il utilise la direction *plusieurs à 1* des associations. Ici, les documents `employee` se trouvent au niveau 0 de la collection alors que les documents `company` sont imbriqués au niveau 2. Ce choix dans bS5 introduit la redondance des documents imbriqués dans la collection (par exemple, entreprise et département).

Imbrication & référencement

Les bases bS2 et bS6 combinent des collections avec des documents imbriqués et des références. Dans la base bS2, la classe `Department` est représentée comme une collection et l'association avec `Employee` est matérialisée en imbriquant des documents `employee` (comme un ensemble de documents) dans le document `département` correspondant. Cela représente la direction *1 à plusieurs* de l'association. L'association entre `Department` et `Company` est traitée par référencement de `l'entreprise` dans chaque `département` en utilisant l'identifiant `id_company`. La collection `Companies` existe comme une collection separée.

Imbrication & duplication

La base bS6 a été créée avec l'approche présentée par Zhao [45]. Il utilise la direction *plusieurs à 1* des associations pour imbriquer les documents (comme dans bS5), mais en plus, il duplique les documents de façon à avoir une collection de « premier niveau » pour chaque classe. La Figure 4.2 montre les trois collections de bS6, où `Company` existe en tant que :

1. une collection séparée,
2. des documents imbriqués dans la collection `Départements`, et
3. intégrée au troisième niveau dans la collection `Employees`.

Il y a aussi une redondance pour les documents imbriqués, comme cela a été mentionné précédemment pour bS5.

4.3 Étude de performance

Cette section est consacrée à l'expérience menée avec les alternatives de structuration des données introduites dans la section 4.2. La section 4.3.1 présente la configuration de l'expérience. Dans

la section 4.3.2, nous discutons des exigences de mémoire en ce qui concerne les choix de structuration des données. Dans la section 4.3.3 nous présentons les requêtes mises en œuvre pour les différentes bases de documents.

4.3.1 Configuration

Pour notre étude, nous avons d'abord créé les six bases de données MongoDB introduites dans la section précédente (bS1 à bS6). Les six bases ont été remplies avec les mêmes données : 50 entreprises, 10 départements par entreprise et 1000 employés par département. Les données sont cohérentes.

Nous avons programmé les requêtes introduites dans la section 4.3.3 pour toutes les bases de données. Dans certains cas, la mise en œuvre d'une requête diffère beaucoup d'un schéma à un autre à cause du nombre de collections concernées et des niveaux d'imbrication où se trouvent les données requises.

Nous avons également créé six bases de données nommées bS1-ix à bS6-ix. Ces bases sont une réplique des informations contenues dans les bases bS1 à bS6 mais en considérant les index nécessaires pour l'exécution du même ensemble de requêtes. Les détails de ces index sont présentés dans la section 4.4.2.

Pour chaque base de données, sans et avec index, chaque requête a été exécutée 31 fois. La première exécution de chaque séquence, à froid, a été séparée.Cette première exécution, à froid, est beaucoup plus longue car les données ne sont pas en mémoire. L'expérience a été exécutée sur un poste de travail avec processeur Intel Core i7 avec 1,8 GHz et 8 Go de RAM. Nous avons utilisé la version MongoDB 3.2.6

4.3.2 Évaluation de la taille des bases de documents

La Figure 4.4 montre la taille des six bases de données. La Figure 4.5 montre la taille de chaque collection dans chaque base de données. Notez la grande taille des bases de données bS5 et bS6 par rapport aux autres options. Les tailles de bS5 et bS6 sont principalement dominées par la collection `Employees`, qui a une structure entièrement imbriquée en suivant les directions plusieurs-à-un des associations. Les documents `Company` et `Department` sont dupliqués dans `Employees`. La taille de la base bS6 est un peu plus grande que celle de bS5 en raison des collections `Companies` et `Departments`, lesquels ont des copies supplémentaires en bS6.

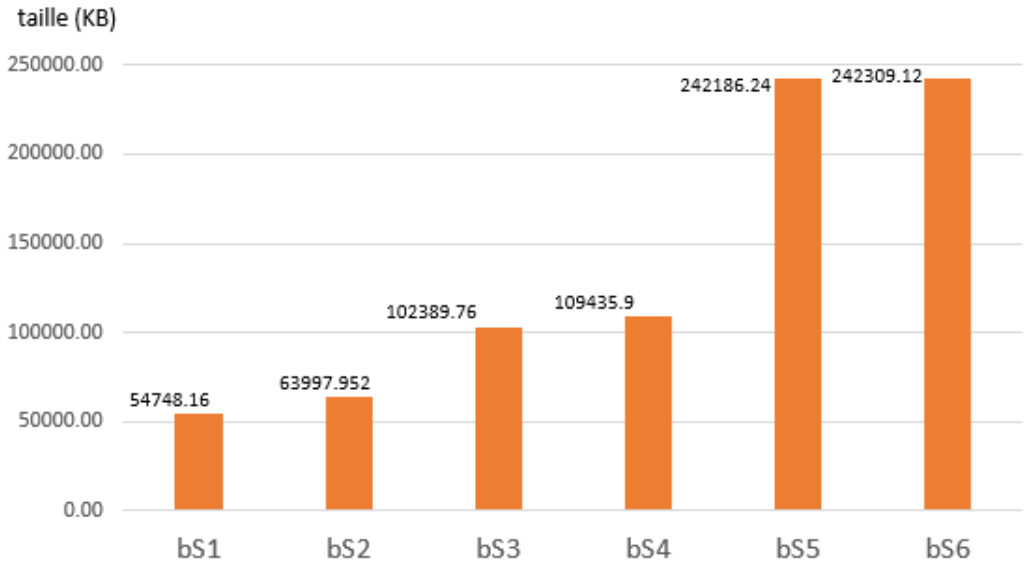


FIGURE 4.4: Taille des bases de documents

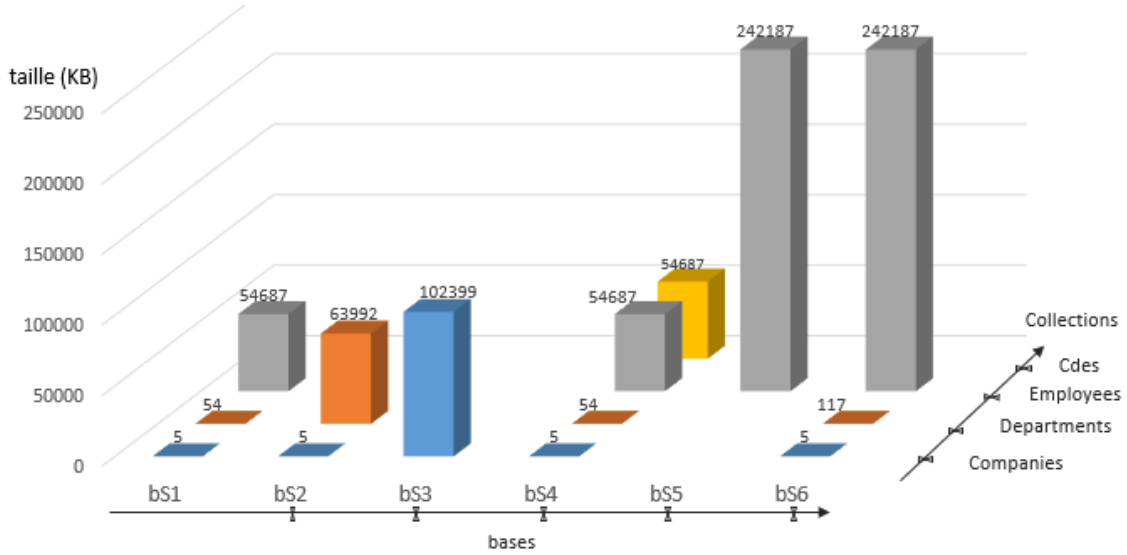


FIGURE 4.5: Taille des bases de documents par collection

Les bases bS4 et bS1 n'imbriquent pas des documents. Notons sur la Figure 4.4 que la taille de bS4 est deux fois la taille de bS1 à cause de la taille de la collection `CDES` (représentant les associations), ce qui est significatif dans notre cas d'étude. La taille des collections imbriquées a une tendance croissante, même si elles n'ont pas de duplication.

En ce qui concerne les bases bS2 et bS3, elles présentent des structures plus complexes avec des tableaux de documents imbriqués à des différents niveaux. Concernant bS2 et bS3 par rapport à bS1, on a constaté les faits suivant :

- $taille(\texttt{Departments@bS2}) > (taille(\texttt{Employees@bS1}) + taille(\texttt{Department@bS1}))$,

- $taille(\texttt{Companies@bS3}) > (taille(\texttt{Employees@bS1}) + taille(\texttt{Department@bS1}) + taille(\texttt{Companies@bS1}))$

4.3.3 Évaluation de requêtes

Pour analyser l'impact de la structuration des données, nous avons établi un ensemble Q de requêtes à exécuter et à évaluer sur les six bases de données de documents créées avec les structurations S1 à S6. Q comprend des requêtes avec une complexité croissante : une sélection avec un simple prédicat d'égalité/inégalité sur un seul type de classe (cf. Q1, Q2, Q6), des requêtes nécessitant de grouper des données (cf. Q4), des requêtes portant sur l'information de plusieurs classes (cf. Q5, Q7) et des requêtes qui nécessitent des fonctions d'agrégation (cf. Q3).

[Q1] Les employés ayant un salaire égal à $1003.
[Q2] Les employés ayant un salaire plus élevé que $1003.
[Q3] Les employés ayant le salaire le plus élevé.
[Q4] Les employés ayant le salaire le plus élevé par entreprise et l'identifiant de l'entreprise.
[Q5] Les employés ayant le salaire le plus élevé par entreprise et le nom de la société.
[Q6] Le salaire le plus élevé.
[Q7] Informations sur les entreprises, y compris le nom de leurs départements.

Ces requêtes ont été utilisées pour étudier le développement et les performances de celles-ci en fonction de chaque structuration. Dans le système MongoDB, l'implémentation de ces requêtes est réalisée avec un langage non déclaratif. Il s'agit d'une programmation de requêtes très algorithmique et mois algébrique. Cela introduit une dépendance très forte avec la structure des données.

Par conséquent, les bases bS1 et bS3, structurées différemment conduisent a deux programmes différents pour une requête telles que Q1. Le code peut être extrêmement simple comme dans le cas de la base bS1 ou au contraire, nécessiter des algorithmes plus complexes.

4.4 Analyse des résultats

Dans cette section, nous analysons les résultats de l'expérience du point de vue de la structure des données et des performances des requêtes (Section 4.4.1). Dans la section 4.4.2, nous nous concentrons sur l'impact de l'indexation.

4.4.1 Bases de documents sans index

Comme nous l'avons mentionné précédemment, l'expérience comprend l'exécution des requêtes Q1 à Q7 sur les bases de documents bS1 à bS6. Afin de faciliter l'analyse, la Figure 4.6 indique le temps d'exécution médian pour chaque requête sur la base correspondante. En outre, la Figure 4.7 représente la performance relative pour chaque requête par rapport à toutes les bases. Il y a une ligne par requête. Les bases sont classées de gauche à droite, en commençant par celles où les performances sont les meilleures. Par exemple, pour Q7, la meilleure performance a été obtenue sur la base bS3 et la pire sur la base bS5, qui correspondent aux structures S3 et S5. Pour Q2, les performances sur les bases bS1, bS4, bS5 et bS6 sont très proches et nettement meilleures que la performance sur les bases bS3 et bS4.

Nous nous intéressons d'abord aux structurations des bases avec des *collections uniques complètement imbriquées*. Avec ce genre de structuration, les performances sont très bonnes lorsque le patron d'accès de la requête suit la direction de la structure imbriquée. Cela peut être apprécié pour les bases bS3 et bS5 par rapport aux requêtes Q4, Q5 et Q7. Lorsque Q4 et Q5 recherchent les données dans `Employees` par rapport à `company`, la structure S5 de la base bS5 n'est pas bonne car les données sont dispersées aux niveaux 0 et 2, respectivement. Cela implique de traverser à partir du bas de la hiérarchie et de remonter à plusieurs reprises dans la structure imbriquée et en supprimant les informations inutiles impliquées dans le niveau intermédiaire. Désormais, nous utiliserons le terme *intrajoins* pour faire référence à ce processus de traversée vers le bas et de remonter des structures imbriquées d'une collection afin de trouver et de relier les données.

Le cas est différent pour la structuration S3 de la base bS3, avec une seule collection `Company`. Dans celle-ci, les `entreprises` sont au niveau 0, les `départements` au niveau 1 et les `employées` au niveau 0. Les requêtes Q4 et Q5, qui s'intéressent à l'information des `employées` nécessitent par `entreprise`, nécessitent de traiter un niveau intermédiaire comme dans S5, mais contrairement à ce dernier, les *intrajoins* son inutiles. Dans ce cas, même si les informations requises pour Q4 et Q5 ne se trouvent pas dans des niveaux imbriqués contigus, ils suivent le patron d'accès par rapport à l'ordre, d'abord les `enterprises` et après les `employées`. Dans ce cas, les performances sont meilleures sur la base bS3.

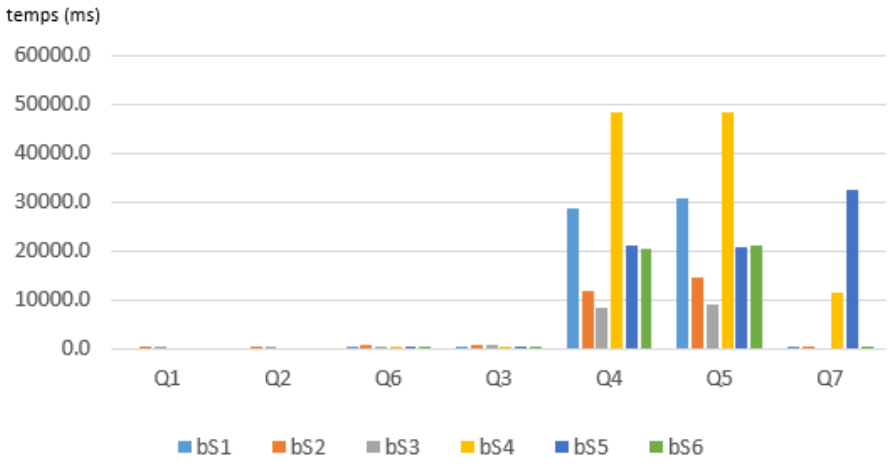


FIGURE 4.6: Synthèse des exécutions : médiane du temps d'exécution des requêtes par base

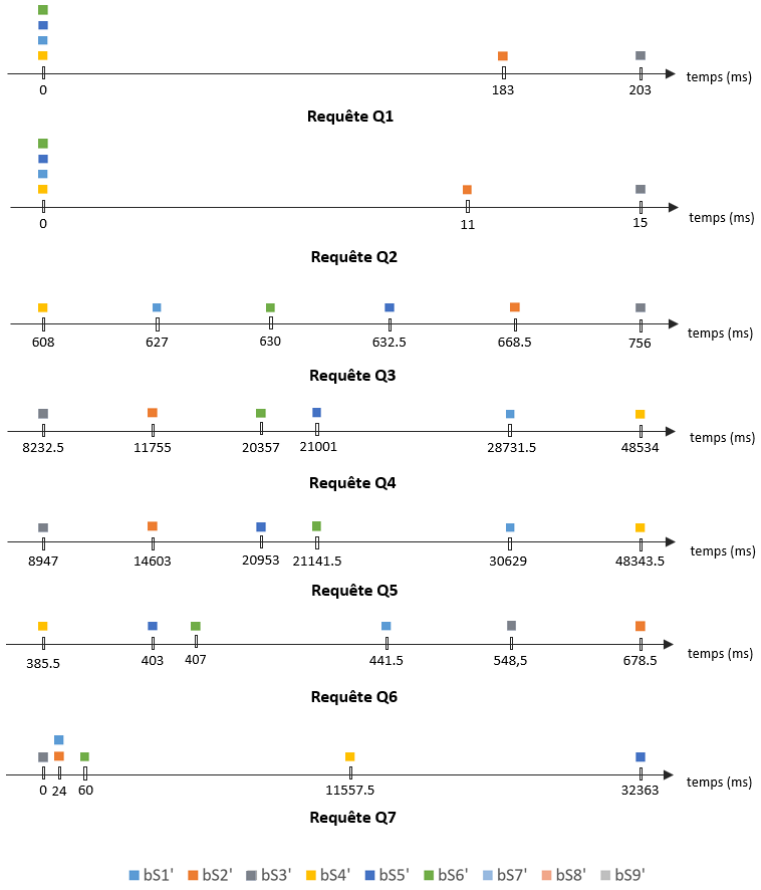


FIGURE 4.7: Synthèse des exécutions : positions relatives de bases

Un comportement similaire se produit avec Q7, qui nécessite l'information des `entreprises` et les noms de leurs `départements`. La base S5 dispose des données nécessaires aux niveaux 2 et 1 respectivement, l'ordre inverse à celui requis, et les données ne sont pas groupées. Cela implique d'abord, de descendre au niveau 1 de chaque document en supprimant l'information inutile du niveau 0 lié aux employées, et se déplaçant après entre les niveaux 1 et 2 en utilisant des *intrajoins* pour obtenir l'information dans l'ordre requis. Tout cela pour regrouper les données situées au niveau 1 basées sur les données du niveau 2. Cette combinaison de contraintes : niveau supérieur inutile, un ordre inverse et pas de regroupement, font que les plus mauvaises performances de la requête Q7 apparaissent lorsqu'elle est exécutée sur bS5.

En ce qui concerne la base bS3, le patron d'accès de la requête Q7 correspond parfaitement à sa structure imbriquée S3. Ici, l'information des `entreprises` est au niveau 0 et les `départements` sont regroupés au niveau 1. Par rapport a la requête Q7, cette structure est avantageuse car les informations requises se trouvent : dans le même ordre, à des niveaux contigus et à partir du niveau 0. Cela évite de supprimer de l'information intermédiaire, de traiter des *intrajoins*, et de descendre des niveaux pour trouver la première information requise, `entreprises` dans ce cas-là. À nouveau, S3 présente la meilleure performance. Toutefois, cela ce n'est pas vrai pour toutes les requêtes. La requête Q3 sur bS3 a la plus mauvaise performance car les données requises sont imbriquées au niveau 2.

Afin de favoriser un certain accès, plusieurs copies des mêmes données peuvent être créées en utilisant des collections avec des structures différentes, comme cela se fait dans S6. S6 étend S5 en incluant des collections supplémentaires comme `Departaments`, où `company` est imbriquée. Cette collection correspond exactement aux données requises par les requête Q7. Le requête Q7 n'utilise pas `Employees`. Dans ce cas, même si des *intrajoins* sont nécessaires, le modèle d'accès de requête est dans l'autre sens, la performance de Q7 est bien meilleure dans bS6 que dans bS5. La principale raison est que les données requises sont aux niveaux 0 et 1 dans bS6, évitant ainsi des niveaux intermédiaires ou supérieurs. Dans bS5, les employés doivent être parcourus même si leurs données ne sont pas utiles pour Q7.

En considérant des structurations avec *collections séparées sans imbrication* telles que S1 et S4, il est évident que les meilleurs résultats pour les requêtes Q1, Q2 et Q3. Ceci est expliqué par le fait que l'ensemble de données à lire par ces requêtes correspond parfaitement à une collection. Aucune donnée inutile n'est lue, et aucune structure complexe doit être manipulée. Ce n'est pas le cas pour les requêtes Q4 et Q5, qui présentent de mauvaises performances dans bS1 et bS4. Dans les deux cas, les requêtes ont besoin de données concernant les employés et les entreprises, qui sont stockées dans des collections distinctes — par exemple, `Cdes` dans bS4 et `Departments` dans bS1 —. Par conséquent, dans les bases bS1 et bS4, l'évaluation de ces requêtes nécessite des opérations de jointure et de groupement, alors que dans S3, la collection `Company` reflète déjà cette information avec la structure appropriée.

Le requête Q7 est également très inefficace dans bS4. Le surcoût est du à la recherche de la association departments-company dans la collection `Cdes`.

Il convient de souligner que l'hypothèse sur *la cohérence des données* est importante pour la mise en œuvre des requêtes. Par exemple, pour le requête Q7 sur bS5 et bS6, cette hypothèse permet l'extraction le nom du premier `département` trouvé sans scanner toute l'extension de `départements`.

4.4.2 Impact des index

Nous terminons notre étude par la création des index dans toutes les bases de documents et l'analyse de l'avantage sur les performances de requêtes Q1 à Q7. Nous avons créé des index sur les identifiants (par exemple, l'id de l'entreprise) et sur le salaire aux différents niveaux d'imbrication selon la base.

Le Tableau 4.1 présente un tableau récapitulatif des index créés dans chacune des six bases documentaires qui considèrent les index (bS1-Ix à bS6-Ix). Pour chaque index, le tableau présente la collection, l'attribut avec son chemin d'imbrication, le type d'index (single ou multikey) et le numéro du niveau où se trouve l'attribut. Rappelons que ces six bases bS1-Ix à bS6-Ix contiennent les mêmes informations que celles contenues dans les bases bS1 à bS6. La Figure 4.8 montre pour chaque base de données la taille totale divisée par la taille de données et la taille des index.

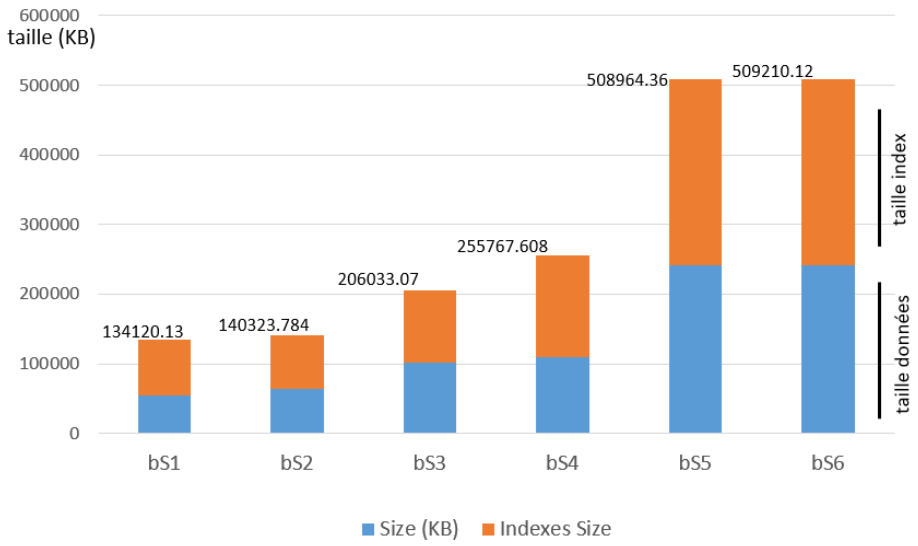


FIGURE 4.8: Taille des bases de documents avec index

Base	Collection	Indice	Type	Niveau imbrication
bS1-lx	Employees	salary	Single field	0
	Employees	id_dept	Single field	0
	Departments	id_comp	Single field	0
	Companies	id_company	Single field	0
bS2-lx	Departments	employees.salary	Multikey	1
	Departments	id_comp	Single field	0
	Companies	id_company	Single field	0
bS3-lx	Companies	departments.employees.salary	Multikey	2
	Companies	id_company	Single field	0
	Companies	departments.name	Single field	0
bS4-lx	Cdes	id_comp	Single field	0
	Employees	id_employee	Single field	0
	Employees	salary	Single field	0
	Departments	id_department	Single field	0
	Companies	id_company	Single field	0
bS5-lx	Employees	salary	Single field	0
	Employees	departments.company.id_company	Single field	2
	Employees	departments.id_department	Single field	1
bS6-lx	Employees	salary	Single field	0
	Employees	departments.company.id_company	Single field	2
	Departments	company.id_company	Single field	1

TABLE 4.1: Tableau des index par base de documents

Les Figures 4.9 et 4.10 fournissent une synthèse des résultats des exécutions des mêmes requêtes exécutées sur les bases bS1 à bS6. La Figure 4.11 montre l'amélioration obtenue à l'aide des index par rapport à la configuration sans index. Une amélioration de X% signifie que le temps de réponse d'une requête est réduit de X%.

Comme prévu, les index améliorent les performances. En particulier, l'amélioration de Q4 et Q5 est élevée dans tous les schémas, et atteint 96 % dans S1 (cf. Figure 4.11). Certaines structurations, telles que S1 et S5, avec les pires performances sans index, fonctionnent très bien avec des index (cf. Figure 4.10). Il est intéressant de réaliser que le bénéfice des index dans la base bS3 est moins important que pour les autres schémas. Pour les cas des requêtes Q4 et Q5, la base bS3 est reléguée à la dernière position de la performance par rapport aux autres bases.

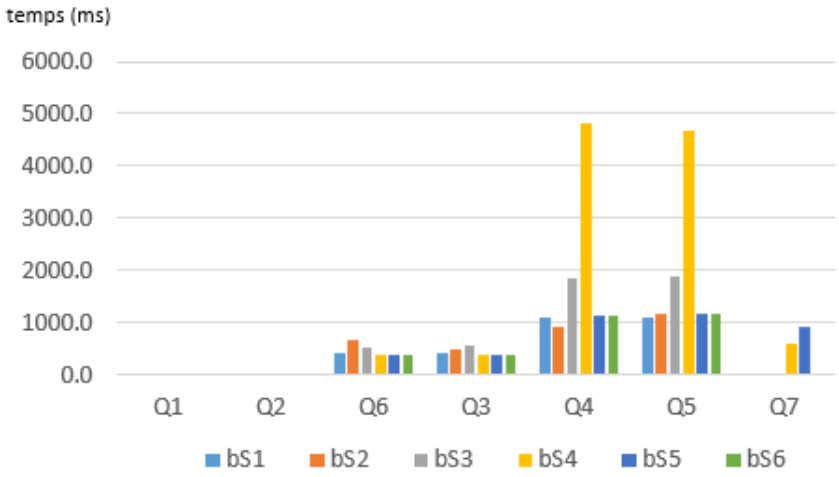


FIGURE 4.9: Synthèse des exécutions avec index : médiane du temps d'exécution des requêtes par base

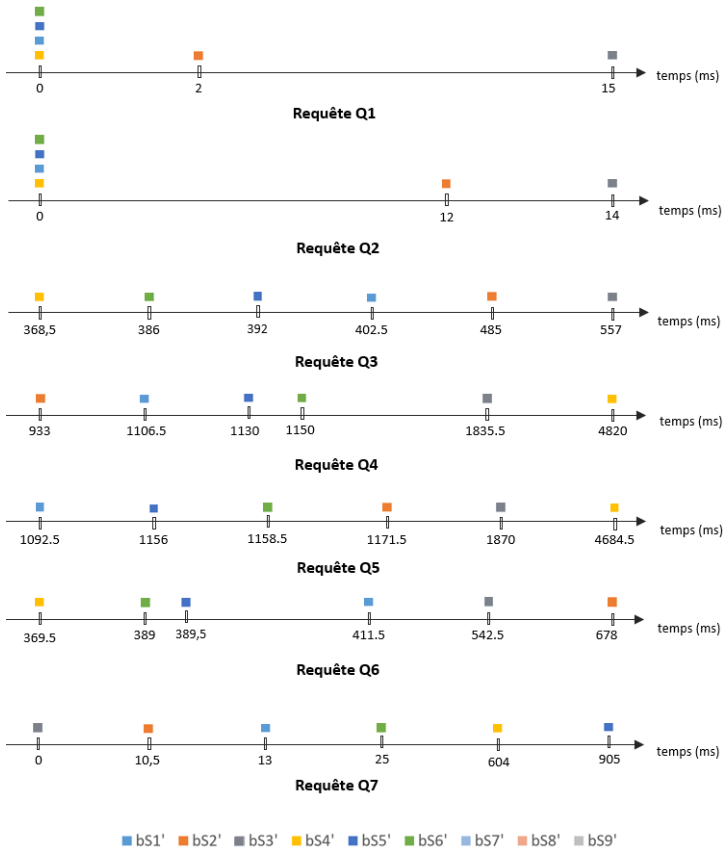


FIGURE 4.10: Synthèse des exécutions avec index : positions relatives de bases

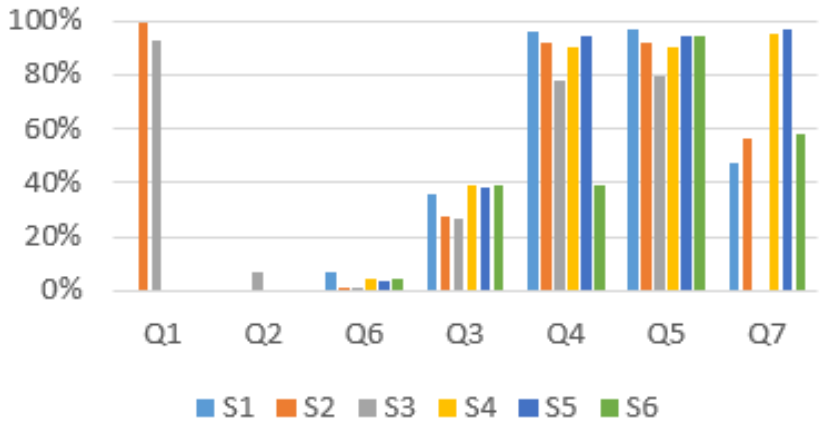


FIGURE 4.11: Impact des index : gain en pourcentage

De plus, les index améliorent les performances de la requête Q7 sur toutes les bases. Néanmoins, l'amélioration obtenue sur les bases bS4 et bS5 ne suffit pas pour dépasser celle sur la base bS3. Cela montre que même si les index sont efficaces, il y a des cas où ces index sont peu utiles si la structure est inconvenante.

4.5 Discussion et conclusion

Cette expérimentation nous a permis de confirmer certaines idées intuitives sur la modélisation des données avec MongoDB, mais il a également fait ressortir des aspects inattendus. Ces aspects concernent les performances, mais aussi la lisibilité du code utilisé pour la mise en œuvre des requêtes.

Le niveau d'imbrication des données a un impact sur les performances. L'accès aux données au premier niveau d'une collection est plus rapide et plus facile que l'accès aux données à des niveaux imbriqués.

Interroger les données stockées à différents niveaux d'imbrication dans une collection peut nécessiter des manipulations complexes. Par exemple, lorsque la structure imbrique des tableaux de documents, les algorithmes pour les gérer sont similaires à des jointures mais réalisées au sein de la collection. Ils affectent les performances, mais nécessitent également une programmation plus élaborée.

La structuration attendue, dans la réponse, impact les performances. Ce point peut être un

problème lorsque les données sont complexes. La structure de données de l'information trouvée est pré-formatée avec la structure de la collection interrogée. Par exemple, lors de l'extraction d'un champ A figurant dans des documents imbriqués au niveau k, le résultat maintiendra la structure imbriquée si aucune opération de restructuration n'est effectuée. Cela signifie que la réponse peut avoir des niveaux d'imbrication inutiles avec des informations non nécessaires présentes dans l'un des k-1 niveaux qui ont été parcourus pour accéder au champ A. La modification d'une telle structure pour fournir les données dans un autre format nécessite un traitement supplémentaire. Le coût de ce traitement supplémentaire ne doit pas être négligé.

En ce qui concerne les besoins de stockage, notre expérience a révélé que l'utilisation des collections avec des documents imbriqués a tendance à exiger plus de stockage par rapport à l'utilisation des collections séparées et « plates », et des références aux données.

Dans ce chapitre nous avons présenté une étude sur l'impact de la structuration des données dans les systèmes basés sur des documents tels que MongoDB. Ce système, comme plusieurs systèmes NoSQL, n'exige pas la création d'un schéma préalable et offre une grande flexibilité de modélisation. Nous avons travaillé avec plusieurs alternatives de structuration des données afin d'évaluer l'impact sur la taille de la base de données, et avec plusieurs requêtes afin d'analyser le temps d'exécution sur chacune des bases. Rappelons que les 6 bases contiennent les mêmes données de départ mais sont structurées de manières différentes.

Les expériences ont montré que les choix de la structure de données sont importants. Les collections avec des documents imbriqués ont un impact positif sur les requêtes qui suivent l'ordre d'imbrication. Cependant, il n'y a aucun avantage — ou il y a la possibilité de mauvaises performances — pour les requêtes qui accèdent aux données dans un autre ordre d'imbrication ou à des données imbriquées à différents niveaux dans la même collection. Les mauvaises performances sont liées à la nécessité de manipulations d'une complexité similaire à celles de jointures avec plusieurs collections. En outre, les collections avec des documents imbrication — même sans duplication — ont tendance à exiger plus de stockage que les mêmes données représentées sur les collections distinctes.

Les propositions de cette thèse cherchent à aider les développeurs dans le choix des structures pendant une phase de conception afin de limiter le coût de la création des bases.

"L'avenir, tu n'as pas à le prévoir, mais à le permettre."
Antoine de Saint-Exupéry

5

SCORUS : approche et système

.

5.1 Introduction

Notre travail se situe dans le contexte NoSQL où l'une des particularités est qu'il y a beaucoup de liberté pour représenter les données et souvent ils ne manipulent pas de schémas. Tenant compte qu'il n'est pas nécessaire de créer un schéma de données préalable à la création des données la phase de conception de schéma n'est pas indispensable. Les données peuvent être redondantes, fragmentées et/ou incomplètes. En plus, il n'y a pas une séparation claire des couches logiques et physiques.

Ainsi, la plupart de ces systèmes offrent beaucoup de flexibilité. Les modifications sur les structures et les requêtes associées peuvent avoir plusieurs impacts et difficiles à découvrir, comme nous l'avons constaté avec notre étude expérimentale introduite au chapitre 4. En particulier, lorsque les choix de structuration sont nombreux et les avantages et les inconvénients d'une structure à l'autre peuvent varier en fonction de plusieurs facteurs.

Le choix de la structuration reste donc important et critique parce qu'il va conditionner l'empreinte mémoire, les programmes écrits pour traiter ces données qui seront plus ou moins faciles à utiliser, à lire, à maintenir selon la structure choisie, et bien sûr la performance qui dépendra des structurations. Cela implique que plusieurs alternatives sont des candidats potentiels à être choisies comme un schéma unique, un schéma temporel, un schéma parallèle, etc. selon le cas.

Cependant, il est difficile de choisir des structures semi-structurées appropriées parce que d'une part il y a plusieurs options à gérer et de l'autre part il n'y a pas de critères objectifs qui permettent de les évaluer et de les comparer. Il devient donc important d'aider l'utilisateur à comprendre, évaluer, maintenir, et potentiellement faire évoluer les structures de manière plus avertie, en clarifiant les possibilités et en fournissant des critères objectifs pour prendre des décisions avec les avantages et les inconvénients que cela implique.

Notre proposition est une contribution dans cette direction. Nous proposons *SCORUS*[1], un système pour l'analyse et l'évaluation de structures orientées documents. SCORUS vise à aider les utilisateurs dans un processus de modélisation de données orientée document en utilisant une approche de recommandation. Malgré que les systèmes orientés documents ne supportent pas de schéma de base de données, nous proposons de représenter et de travailler avec un «schéma» pour faciliter la compréhension, l'évaluation et la comparaison des structures de données orientées document. Nous proposons une étude de plusieurs possibilités de structuration des données en contrôlant les variantes et l'explosion de combinaisons de structurations en se basant sur un modèle UML et un ensemble de métriques structurelles objectives et nouvelles. De telles métriques n'ont pas été formellement considérées jusqu'à présent. Elles permettent d'évaluer les structures et de

1. Semi Structured SChemas Recommendation System

les analyser pour mieux faire ressortir les avantages et les inconvénients de chacune par rapport aux besoins des utilisateurs.

Dans la section 5.2, nous introduisons le système SCORUS en décrivant ses trois fonctionnalités : la génération d'un ensemble d'alternatives de structuration, leurs évaluations avec un ensemble des métriques structurelles et l'analyse de ces alternatives. Ces trois fonctionnalités peuvent être utilisées de manière indépendante. Dans la section 5.3, nous décrivons notre représentation AJSchéma qui vise à faciliter le raisonnement sur les choix de structuration des données dans les systèmes orientés documents. Dans la section 5.4, nous introduisons les éléments de traduction d'un modèle UML pour obtenir un schéma semi-structuré centré sur les types d'information et pouvant être exprimés sous la forme de AJSchéma.

5.2 *SCORUS*, un système pour l'analyse et l'évaluation des structures orientés document

La flexibilité et l'absence de schéma dans les systèmes NoSQL orientés documents, telle que MongoDB, permettent d'explorer des nouvelles alternatives de structuration sans faire face aux contraintes. Le choix de la structuration reste important et critique parce qu'il y a plusieurs impacts à considérer et il faut choisir parmi des nombreuses d'options de structuration. Nous proposons donc de revenir sur une phase de conception dans laquelle des aspects de qualité et les impacts de la structure sont pris en compte afin de prendre une décision d'une manière plus avertie.

Dans ce cadre, *SCORUS* vise à faciliter l'étude des possibilités de semi-structurations orientées documents, telles que MongoDB, et à fournir des métriques objectives pour mieux faire ressortir les avantages et les inconvénients de chaque solution par rapport aux besoins des utilisateurs. Pour cela, une séquence de trois phases peut composer un processus de conception. Chaque phase peut être aussi effectuée indépendamment à des fins d'analyse et de réglage. La Figure 5.1 illustre la stratégie générale de *SCORUS*, composée par :

1. Génération d'un ensemble d'alternatives de structuration : dans cette phase nous proposons de partir d'une modélisation UML des données et de produire automatiquement un large ensemble de variantes de structuration possibles pour ces données.
2. Evaluation d'alternatives en utilisant un ensemble de métriques structurelles : cette évaluation prend un ensemble de variantes de structuration et calcule les métriques au regard des données modélisées.
3. Analyse des alternatives évaluées : utilisation des métriques afin d'analyser l'intérêt des alternatives considérées et de choisir la ou les plus appropriées.

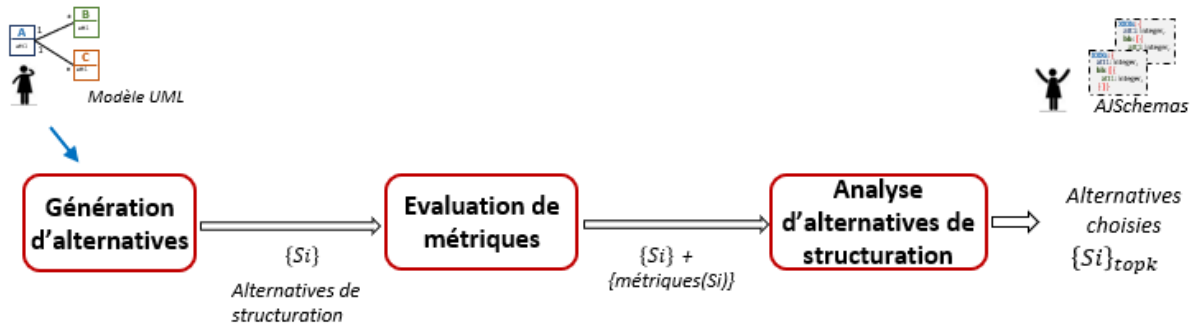


FIGURE 5.1: Organisation générale de *SCORUS*

Pour faciliter le raisonnement sur les structures, nous proposons de mettre l'accent sur les types de données contenues dans les documents. Les types de données peuvent être représentés sous forme de structures de documents qui pourraient être assimilés à des éléments d'un schéma d'une base. Pour la réalisation de SCORUS nous manipulons deux représentations, l'une arborescente, nommée *AJTree* et l'autre textuelle, nommée *AJSchéma*. Cette représentation est basée sur l'approche JSON schema et est centrée sur une présentation finale claire et textuelle des structures.

Les sous-sections suivantes sont consacrées à la description générale et à la motivation des trois étapes du processus de conception de *SCORUS*. La représentation *AJSchéma* ainsi que les principes de création de variantes à partir d'un modèle UML sont discutés dans les sections 5.3 et 5.4.

5.2.1 Génération d'alternatives de structuration

L'une des conséquences de la flexibilité dans la représentation de données dans les bases orientées documents est le grand nombre d'alternatives semi-structurées qui peuvent apparaître et satisfaire les besoins des métiers et des applications. Nous voulons donc fournir un ensemble d'alternatives, parmi lesquelles plusieurs options peuvent être envisagées ou choisies. Cependant, le nombre d'alternatives est très important et il suffit de considérer ou d'ajouter un peu d'information pour déclencher une explosion de possibilités.

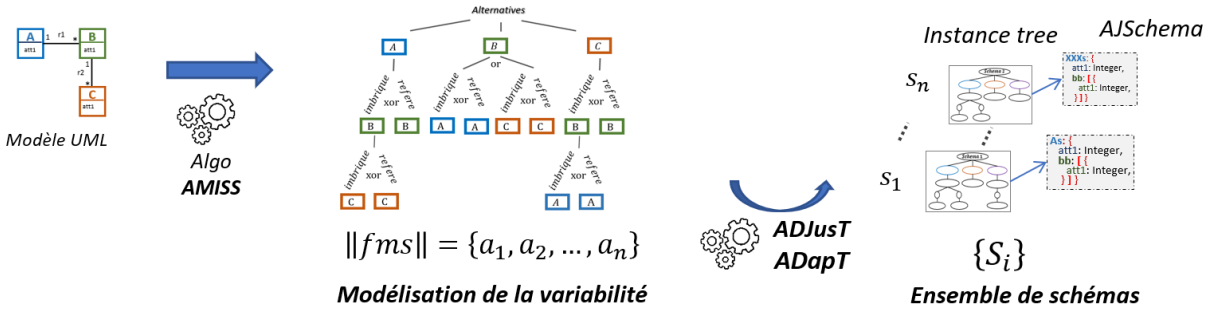


FIGURE 5.2: Phase de génération d'alternatives

Dans cette étape, nous proposons de modéliser différentes alternatives de semi-structuration à partir de l'information fournie par un modèle UML. Nous mettons l'accent sur les types d'information (données) et la façon de les structurés. Nous cherchons aussi à contrôler de façon simple et automatique l'explosion des possibilités qui se présentent lorsque l'on considère les combinaisons entre les classes et les relations présentes dans ce modèle UML. La Figure 5.2 introduit la organisation générale de cette étape de génération.

Nous proposons générer un ensemble des alternatives de structuration d'une manière compacte, en établissant leurs variations et points communs afin d'identifier facilement les choix de chaque structuration. Pour cela, nous utilisons les modèles de caractéristiques, une stratégie à la fois simple et puissante utilisée dans le domaine des lignes de produits logiciels et qui permet d'exprimer la variabilité entre les produits à construire en contrôlant l'explosion de combinaisons [80] [81]. Un modèle de caractéristiques, représenté par une structure arborescente, spécifie les similarités et les différences d'une famille de produits, les alternatives de structuration dans notre cas.

Notre modèle de caractéristiques fms, considère les alternatives concernant l'inclusion et la conception d'une collection à partir d'une classe en considérant les chemins de ses relations, et les possibilités d'imbriquer ou référencer l'information des classes incluses dans ces chemins. Un ensemble de contraintes est défini afin de garantir la complétude de chaque alternative de structuration et à éviter les boucles entre les références et l'imbrication. La construction automatique de ce modèle est réalisée par notre algorithme *AMISS*. Cet algorithme est introduit en détail dans le chapitre 6.

Chaque instance de ce modèle est composée d'une série de caractéristiques choisies et valides qui définissent une alternative de structuration. Chacune de ces instances ou alternatives est interprétée automatiquement par des algorithmes, nommés *ADJusT* et *ADApt*, pour ajuster sa structure arborescente, afin de l'utiliser ultérieurement, et construire un schéma sous forme AJSchéma.

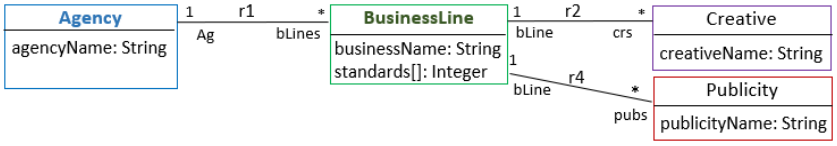


FIGURE 5.3: Diagramme UML dans un contexte de Marketing

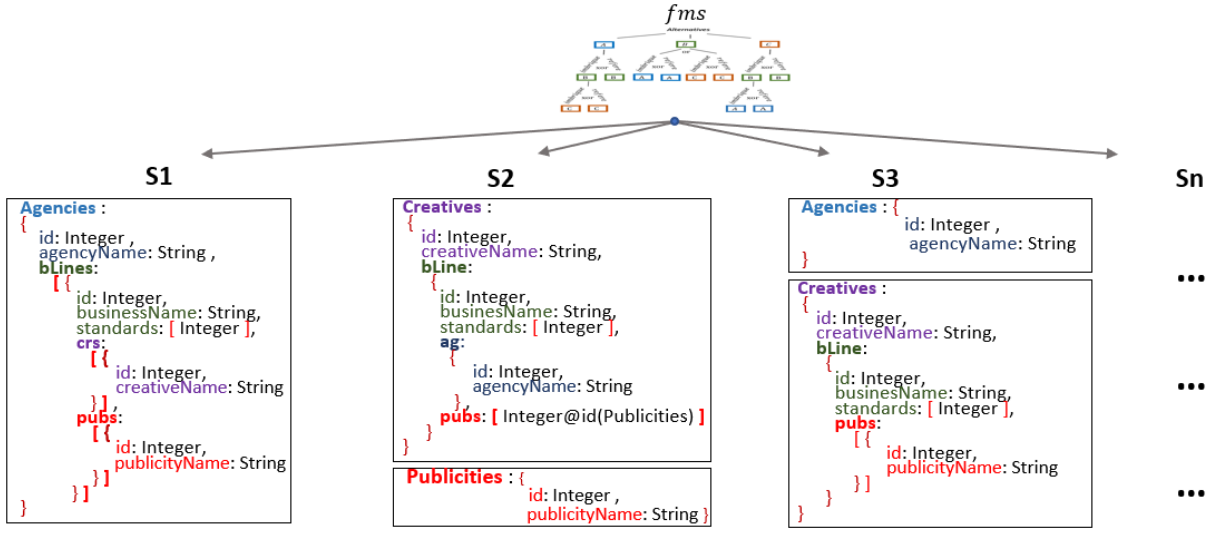


FIGURE 5.4: Schémas sous forme AJSchéma basés sur le modèle UML de la Figure 5.3

La Figure 5.3 développe un modèle de classes UML dans un contexte de Marketing. Le modèle considère quatre classes et des associations 1 .. *. La classe `Agency` représente une agence qui gère plusieurs secteurs d'activité, représentés par la classe `BusinessLine`. Chaque secteur d'activité est en charge de plusieurs designers (classe `Creative`) et lié à plusieurs publicités (classe `Publicity`). Chaque designer et publicité appartient à un seul secteur d'activité et un secteur d'activité est géré par une seule agence.

La Figure 5.4 introduit trois alternatives de structuration, parmi tant d'autres qui peuvent exister, basées sur le modèle UML de la Figure 5.3. L'alternative *S*1 contient une seule collection `Agencies`, déclenchée par la classe `Agency`. Ses documents sont constitués pour l'ensemble d'attributs propres correspondant à cette classe ou type *tAgency*. Un attribut est ajouté sous le nom `bLines`. Celui-ci correspond à l'association avec la classe `BussinessLine` et imbrique ses attributs à l'aide d'un tableau. Chacun de ces documents imbrique à son tour les attributs de `Creative` et `Publicity`.

L'alternative *S*2 contient deux collections. La collection `Publicities` dont les documents sont constitués par des attributs correspondant au type *tPublicity*. La deuxième collection, `Creatives`, contient des documents avec des attributs de type *tCreatives* et un attribut supplémentaire qui

imbrique les attributs de `BussinessLine`. Ce dernier imbrique un document, nommé `ag`, avec les attributs de *tAgency* et référence les documents de la collection `Publicités`. D'autres alternatives son présentes (*S*3 ... *Sn*)

L'utilisateur peut intervenir dans cette phase en fournissant des contraintes structurelles pour forcer ou interdire des choix et ainsi réduire l'ensemble des alternatives générées par l'algorithme *AMISS*. Il peut choisir entre un ensemble de restrictions permettant de contrôler divers aspects tels que l'imbrication, les références et la duplication.

Une fois les alternatives générées, l'utilisateur peut également en choisir un sous-ensemble pour les évaluer ou simplement les préserver pour des utilisations ultérieures. De même, l'utilisateur peut traduire les instances des alternatives générées en utilisant un autre format, mais dans ce cas, il doit fournir le composant approprié.

5.2.2 Évaluation de métriques structurelles

Comme nous l'avons souligné, le choix de la structuration reste important et critique parce qu'il y a plusieurs impacts difficiles à découvrir. Actuellement, il existe des lignes directrices pour guider la modélisation basée sur de bonnes pratiques qui permettent d'évaluer qualitativement certains aspects intéressants de la semi-structuration. Il existe également des évaluations de la performance des requêtes par rapport à une base déjà implémentée et de mesures de sa taille. Cependant, il n'existe pas de critères objectifs et formels qui nous permettent d'effectuer une analyse sur la structure orientée documents d'une base style MongoDB.

En raison de cette absence, nous avons fait une analyse du point de vue structurel à partir de notre expérience de l'impact et des travaux connexes afin d'établir les caractéristiques qui sont en relation. Nous avons analysé un certain nombre de caractéristiques critiques sur les structures et nous nous sommes aussi inspirées des métriques proposées pour les données complexes, les documents XML et dans d'autres domaines tels que l'ingénierie logicielle. Notre proposition pour cette phase d'évaluation est donc nouvelle car nous fournissons des outils quantitatifs pour évaluer les structures en proposant de métriques structurelles, cf 5.5.

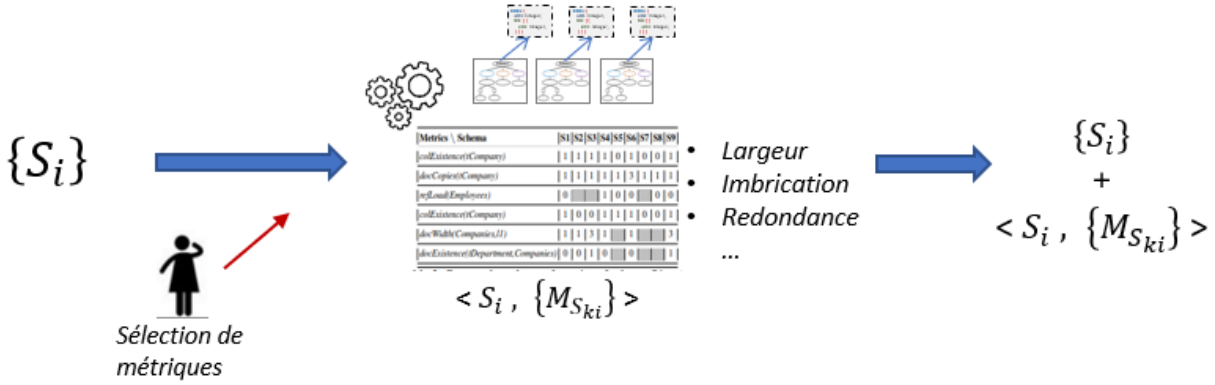


FIGURE 5.5: Phase d'évaluation d'alternatives

Nous proposons un ensemble de métriques mesurables sur les structures de données indépendant des instances. Lorsqu'elles sont disponibles, des informations statistiques sur les données peuvent également être utilisées. Nous proposons des métriques couvrant des aspects tels que l'existence, la profondeur, la largeur, le référencement, la redondance et la taille. Ces métriques reflètent la complexité des éléments des structures qui jouent un rôle sur les aspects de qualité et les impacts sur les applications et requêtes.

On pourra par exemple être intéressé à savoir où se trouve une information à l'intérieur d'une collection ou si elle existe. En reprenant les alternatives de structuration de la Figure 5.4 basées sur le modèle UML de la Figure 5.3, nous notons que l'alternative $S2$ est composée de deux collections, `Creatives` et `Publicities`, et l'alternative $S1$ possède une seule collection `Agencies` contenant toutes les informations. Maintenant, si nous nous intéressons uniquement aux informations des agences, elles se trouvent pour $S2$, dans la collection `Creatives`, et pour $S1$ dans la collection `Agencies`.

Si en plus de rechercher l'existence d'une information, nous avons besoin d'estimer, par exemple, le coût des navigations dans la structure ou la restructuration des réponses, on pourrait avoir un intérêt particulier pour un type d'information et avoir connaissance de la profondeur où il se trouve. Dans notre exemple, les informations des agences, sont à un deuxième niveau de la collection `Creatives` dans $S2$ et au premier niveau de la seule collection de $S1$

Pour accélérer les accès et limiter des opérations coûteuses, où estimer les impacts de la maintenabilité et la cohérence, l'utilisateur peut considérer la redondance. Dans notre exemple, il est possible d'identifier pour la structure $S2$, que l'imbrication, engendre 6 copies de la même agence pour chaque instance de designer.

En détaillant une structure, des aspects à identifier semblent si évidents qu'ils ne sont pas mesurés, d'autres sont entremêlés et ressemblent à un seul, et d'autres ne sont pas évidents à première vue. Nous devons en tenir compte au moment de créer, par exemple, une requête. Nous proposons de les identifier, les isoler et les formaliser afin de fournir des éléments plus précis pour évaluer la structure.

Afin de réaliser une évaluation automatique des métriques, nous proposons une représentation des structures sous forme arborescente axée sur les types d'informations, les niveaux, l'imbrication et les références. Cette forme correspond à la représentation textuelle AJSchéma utilisée dans l'exemple de la Figure 5.4. Les motivations et les définitions des métrique proposées sont présentés dans le chapitre 7. Un positionnement de nos métriques par rapport à celles utilisées dans d'autres types de structures et de domaines est introduit également dans le chapitre 7.

Cet ensemble de métriques peut être évalué sur chacune des alternatives lors de la phase de génération d'alternatives. Une sélection de métriques peut être établie par l'utilisateur soit parce qu'il choisit spécifiquement l'ensemble disponible, soit parce qu'il déclare un critère ou une préférence pouvant conduire à choisir une métrique appropriée à cet effet.

Il est possible d'évaluer ces métriques sur une ou plusieurs structures non générées par SCORUS, résultat de l'application de l'ingénierie inverse à une base ou à une structure non prise en compte par la génération automatique. Dans ce cas, la représentation de la structure ou des structures doit correspondre à la représentation requise, ce qui implique l'utilisation d'un outil externe qui garantit cette exigence.

L'évaluation de métriques peut être utilisée comme un bloc indépendant, qui délivre les valeurs des évaluations à traiter en tant que statistiques sans nécessairement prendre de décisions à leur sujet.

5.2.3 Analyse d'alternatives de structuration

Dans cette phase, on utilise les évaluations des métriques de chaque alternative de structuration pour les analyser et les comparer afin de choisir une ou plusieurs alternatives. Il s'agit en priorité de faire émerger le ou les schémas les plus favorables selon certains critères mais aussi d'écarter des choix très défavorables ou encore, d'envisager des schémas alternatifs qui n'étaient pas forcément considérés au départ.

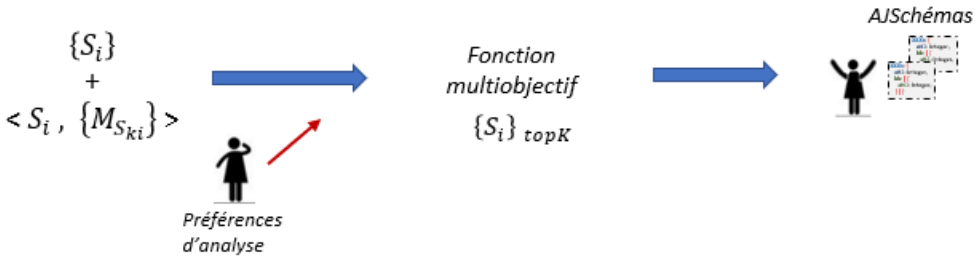


FIGURE 5.6: Phase d'analyse d'alternatives

Pour cela, des critères peuvent être établis selon le contexte applicatif mais peuvent aussi correspondre à de bonnes pratiques préconisées pour le développement ou à une priorité générale. Par exemple, adopter des schémas très "compacts" pour limiter l'empreinte mémoire lorsque des données seront peu utilisées, ou des structures permettant de donner priorité à la qualité logicielle, ou encore en privilégient les schémas les plus "lisibles".

Les métriques les plus adaptées à chaque critère sont mises en relation pour une analyse selon les préférences. Nous définissons un critère comme une préférence, en terme de maximisation ou de minimisation, sur une ou plusieurs métriques pour privilégier un sous-ensemble de structures.

L'analyse de chaque critère permet de comparer partiellement les variantes. Cependant, elle ne permet pas d'examiner l'ensemble des facteurs pouvant avoir une incidence sur une sélection. Nous proposons donc une analyse multiobjectif en considérant tous les critères basés sur un système de points 5.6. A chaque fonction de critère est associé un poids. Les critères peuvent avoir le même poids ou bien privilégier certains critères. La fonction d'évaluation fera la somme pondérée des critères pour chaque structuration. Les poids font référence aux priorités des utilisateurs et/ou à la fréquence des requêtes.

Dans le cadre de cette thèse nous réalisons une analyse non automatisée qui sera introduite dans la validation du chapitre 8. Dans notre recherche, les critères à considérer et le choix des poids sont un sujet à approfondir.

Sachant que les critères peuvent diverger et évoluer, l'utilisation des métriques et des critères pour un choix de schémas peut aider dans un processus continu de "tuning" de la base qui peut conduire à des évolutions de la structuration ou à la création de copies des données avec des structures différentes. Pendant un certain temps, une base pourrait avoir, pour les mêmes données, une copie avec une certaine structure Sx et une autre copie avec une autre structure Sy, comme par exemple, S1 et S3 sous la Figure 5.4.

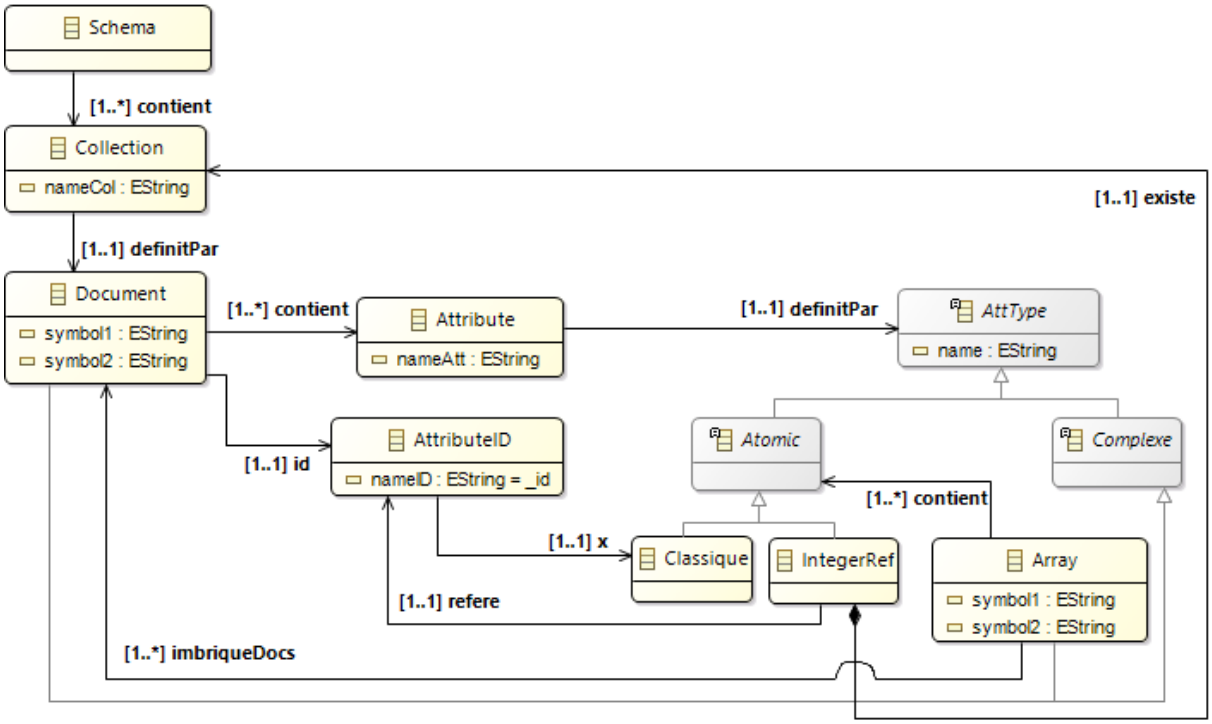


FIGURE 5.7: Méta-modèle correspondant à la grammaire $G_{AJSchema}$

5.3 AJSchéma : Représentation des structures des bases de documents

Pour faciliter le raisonnement sur les choix de structuration des données dans les systèmes orientés documents, nous définissons une représentation, appelée ici *AJSchéma*. Cette représentation est basée sur l'approche de JSON schéma [61] et sur les types de données supportés par MongoDB.

5.3.1 Formalisme AJSchéma

L'objective des AJSchémas est de disposer d'une représentation textuelle, facile à lire, pour présenter les types des structures que seront créées dans la base de documents. Nous proposons pour AJSchéma une grammaire nommée $G_{AJSchema}$. Cette grammaire, correspondant au méta-modèle introduit par la figure 5.7, nous permet de fournir une représentation concise des collections et des types de documents à stocker dans une base orientée documents. Nous définissons la

grammaire $G_{AJSchema}$ ci-dessous :

$G_{AJSchema} = (V_N, V_T, S, P)$

$V_N = schema, collection, pair, type, primitive, complexe, array$

$V_T = boolean, integer, float, string, Integer@idTx$

$Axiome\ S : schema$

$Régles\ P :$

$\langle\ schema\ \rangle$: := $\langle\ collection\ \rangle$ *

$\langle\ collection\ \rangle$: := $\langle\ collectionName\ \rangle$ ‘:’ $\langle\ document\ \rangle$

$\langle\ document\ \rangle$: := ‘{’ $\langle\ attributeID\ \rangle$ ‘,’ $\langle\ attribute\ \rangle$ * ‘}’

$\langle\ attributeID\ \rangle$: := ‘_id :’ $\langle\ atomicRef\ \rangle$

$\langle\ attribute\ \rangle$: := $\langle\ attributeName\rangle$ ‘:’ $\langle\ type\ \rangle$

$\langle\ type\ \rangle$: := $\langle\ atomic\ \rangle$ | $\langle\ complexe\ \rangle$

$\langle\ atomic\ \rangle$: := $\langle\ atomicClassique\ \rangle$ | $\langle\ atomicRef\ \rangle$

$\langle\ atomicClassique\ \rangle$: := $\langle\ Boolean$ | $Integer$ | $Float$ | $String\ \rangle$

$\langle\ atomicRef\ \rangle$: := Integer@id$\langle collectionName\rangle$

$\langle\ complexe\ \rangle$: := $\langle\ array\ \rangle$ | $\langle\ document\ \rangle$

$\langle\ array\ \rangle$: := ‘[’ $\langle\ atomic\ \rangle$ * | $\langle\ document\ \rangle$ * ‘]’

$\langle\ attributeName\ \rangle$: := $\langle\ String\ \rangle$

$\langle\ collectionName\ \rangle$: := $\langle\ String\ \rangle$

$G_{AJSchema}$ décrit pour chaque collection du schéma, le type de ses documents. Un type de document est défini comme un ensemble d'attributs délimités par `{ }`. Les attributs sont définis comme des paires `attributeName:type` qui définissent le nom et le type de chaque attribut. L'attribut nommé *attributeID*, correspond à l'attribut identifiant dont le nom est défini comme *_id*.

Le type d'un *attribut* non identifiant est défini soit comme atomique soit comme complexe. Un type atomique peut correspondre à un type atomique classique (*atomicClasique*) ou à un type atomique de référence *atomicRef*. Le premier est un type Boolean, Integer, Float ou String (ceux de MongoDB). Le type de l'attribut identifiant *attributeID* est défini comme atomique classique.

Nous définissons un type atomique de référence comme *Integer@id<collectionName>*. C'est une valeur de type *Integer* qui permet de référencer par son suffixe @id<collectionName>, des documents d'autre collection. Cette collection doit exister par ailleurs et les documents références doivent apparaître au premier niveau.

Un type complexe est considéré comme un tableau ou un document. Un tableau est symbolisé par [ ] et à l'intérieur, le type de ses éléments atomiques ou documents. La définition d'un type de documents comme un document ou un tableau de documents permet d'introduire la notion d'imbrication dans la définition de la structure d'une collection.

Désormais dans ce manuscrit, nous utilisons indistinctement les termes AJSchéma ou schéma pour faire référence à cette représentation. Pour des systèmes sans schéma comme MongoDB, de telles abstractions peuvent être considérées comme une spécification d'une structure de données pour un processus de construction et / ou de maintenance. L'AJSchéma pourrait aussi être utilisé pour une vérification des types.

5.3.2 Exemple

La Figure 5.8 montre un exemple de structuration de l'information des instances concernant des *tweets* et des *twerson* dans un contexte *Tweeter* dans MongoDB. Il y a deux collections nommées `Tweets` et `GeoTag`. Les instances de documents de la collection `Tweets` contiennent information par rapport à l'identifiant du tweet, le message, la geoLocation, et les personnes qui l'aiment. L'information des personnes qui aiment le tweet est incluse dans un tableau de documents où chaque document représente une personne. Cela ajoute un niveau de profondeur dans la structure de la collection et implique l'imbrication. L'attribut `geoLocation` signale l'emplacement de l'utilisateur lorsqu'il a publié le Tweet. Pour cela, une référence vers un document qui contient l'information pertinente dans la collection `Geotag` est utilisée. Les documents appartenant à la collection `Geotag` contiennent des informations concernant le nom de l'endroit et ses coordonnées.

Dans la Figure 5.9 nous introduisons un AJSchéma qui représente la structure des instances de la Figure 5.8 qu'on pourrait abstraire. Cet AJSchéma considère la structure de deux collections, `Tweets` et `GeoTag`. Le type de documents de la collection `Tweets` est formé par cinq attributs : l'identifiant _id, de type Integer ; le message, de type String ; la geoLocalisation, de type Integer@id(GeoTag) ; et un tableau de type document nommé *twerson* qui représente les personnes qui aiment le tweet. Le type document de ce tableau est formé par trois attributs : l'identifiant _id, de type Integer ; le nom du tweeple twwepleName, de type String ; et un tableau de valeurs String nommé hobbies.

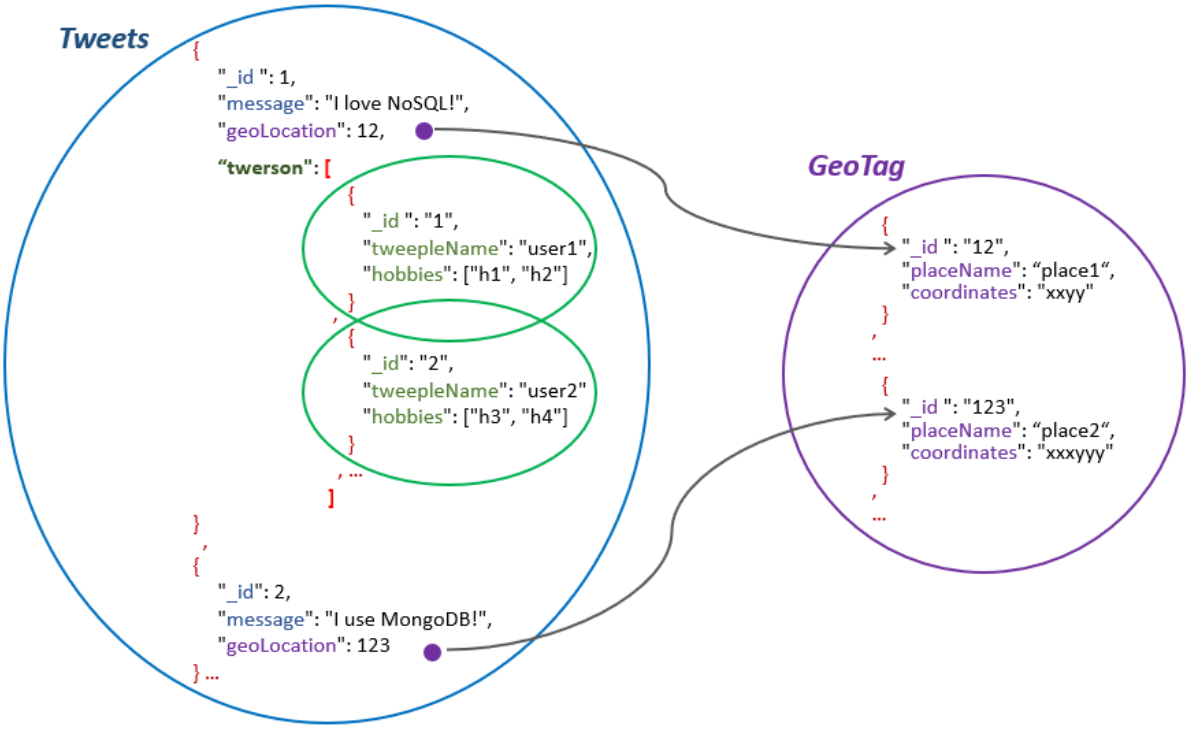


FIGURE 5.8: Exemple d'instances MongoDB en utilisant le référencement et l'imbrication

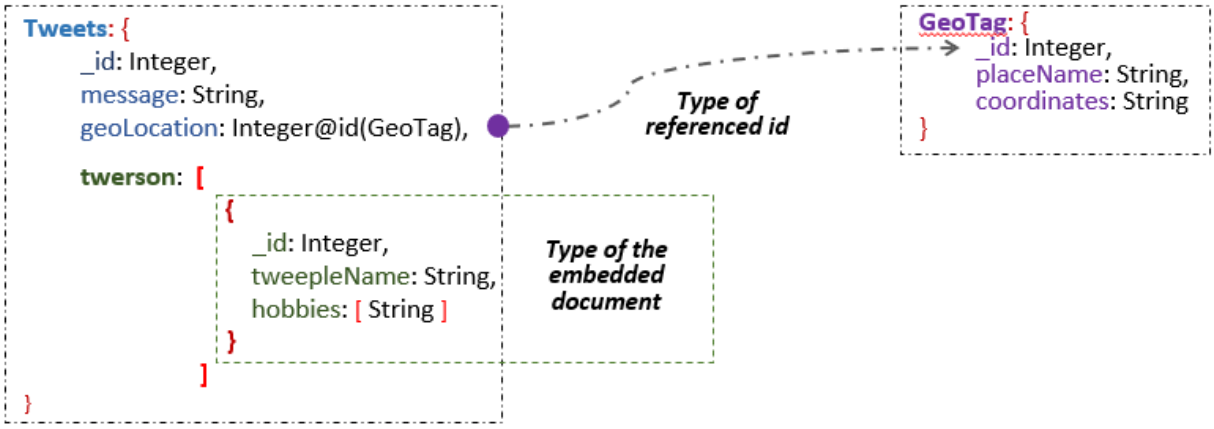


FIGURE 5.9: AJSchéma correspondant aux instances de la Figure 5.8

5.4 Modélisation semi structurée basée sur un modèle UML

Bien que la plupart des bases NoSQL ne prennent pas en compte un schéma, la création des instances est liée au contexte dans lequel elles se trouvent. Nous proposons de revenir sur une phase de conception dans laquelle l'information puisse être définie et donc analysée pour trouver différentes alternatives de représentation de cette information.

Nous nous plaçons dans une démarche de conception où les données sont représentées sous forme de modèle UML. Ces données peuvent être représentées de diverses manières sous-forme de structures de documents en utilisant une représentation AJSchéma. Cette section est consacrée à l'introduction des grandes lignes directrices permettant la modélisation semi structurée à partir d'un modèle UML.

Nous définissons un modèle UML mU comme un ensemble de classes E et d'associations R définies comme suit :

Soit mU un paire (E,R) où :

- $E = \{e_1, ..., e_n\}$ correspond à l'ensemble de classes
- $R = \{r_1, ..., r_n\}$ correspond à l'ensemble d'associations
- $R(e_i) = \{r_1, ..., r_n\}$ c'est l'ensemble des associations de la classe e_i .
- $E(r_n) = \{e_1, e_2\}$ correspond aux classes de chaque extrémité de l'association r_n. $e_1 \neq e_2$
- $A(e_i) = \{a_1, ..., a_n\}$ désigne l'ensemble des attributs de la classe $e_i \in E$.
  Chaque $a_i \in A$ est défini par le paire $a_i.name$:$a_i.type$
- $r_n.name$ c'est le nom de l'association,
- $r_n.ce_i$ correspond à la cardinalité de l'association r par rapport la classe e_i,
- $r_n.role_i$ désigne le rôle de l'association r_i par rapport la classe e_i

Nous introduisons la définition de type correspondant à une classe pour spécifier l'ensemble de ses attributs.

Définition 5.1. Type d'une classe
Le type d'une classe e_i est défini comme l'ensemble de ses attributs plus un identifiant :

$$te_i = \{A(e_i), a_{id}\}$$

où
$a_{id}(e_i)$ c'est un attribut par défaut correspondant à l'identifiant d'une instance de la classe e_i

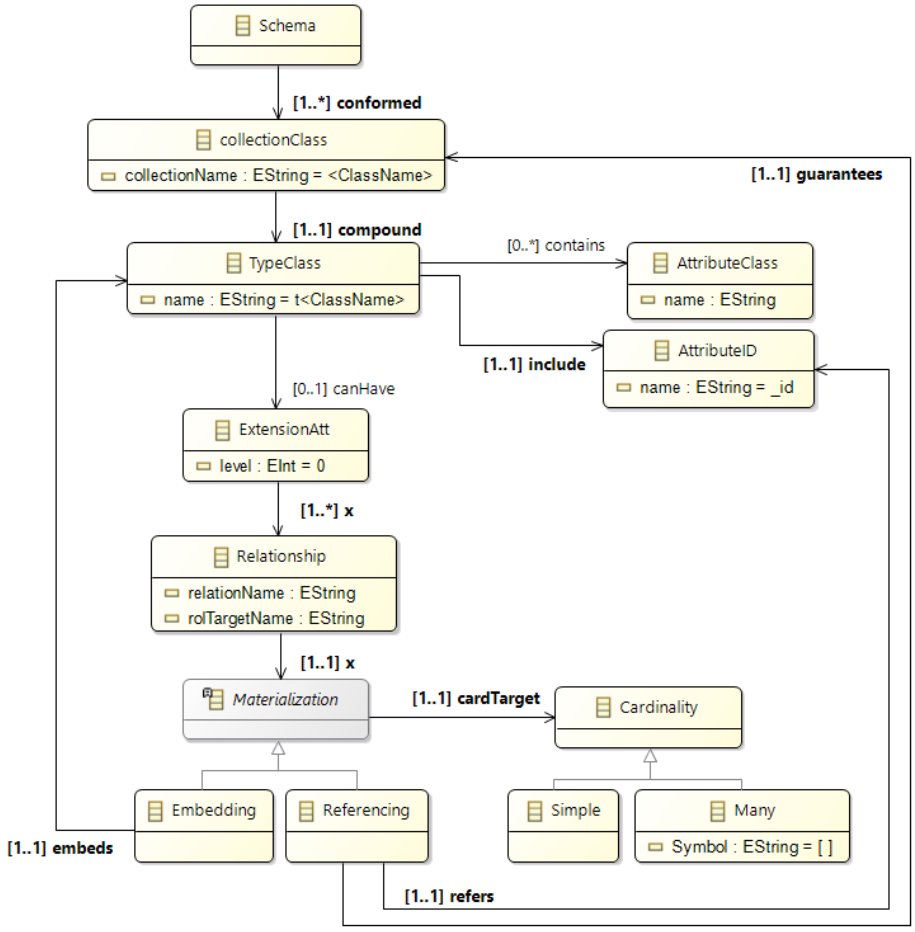


FIGURE 5.10: Méta-modèle lignes directrices de modélisation semi-structurée basés sur un UML

La Figure 5.11 montre un modèle UML avec les classes `Agency`, `BusinessLine`, `Owner`, `Creative` et `Publicity`. Les propriétés de la classe `Agency` sont désignées par le type `tAgency`.

En utilisant le méta-modèle de la Figure 5.10, nommé *SINGULar*, nous établissons des principes d'une modélisation semi structurée basée sur un modèle UML :

— Une classe $e_i \in E$ peut déclencher la création d'une collection.

— Une collection créée à partir de une classe e_i contient de documents dont les attributs correspondent au type de classe te_i au premier niveau.

— Un type de documents te_i peut être étendu en ajoutant un attribut matérialisent une association de la classe e_i. Cet attribut porte le nom du rôle de la classe destination.

— La matérialisation d'une association r_n appartenant à une classe e_i est faite soit par imbrication soit par référencement de la classe destination e_j. En considérant la cardinalité ce_j, le type de l'attribut et défini comme suit :

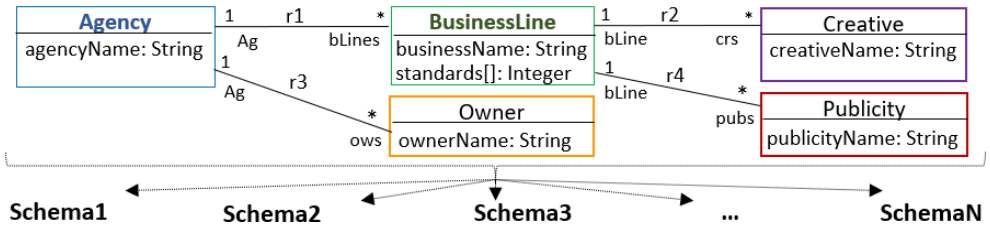


FIGURE 5.11: Plusieurs schémas basés sur un modèle UML

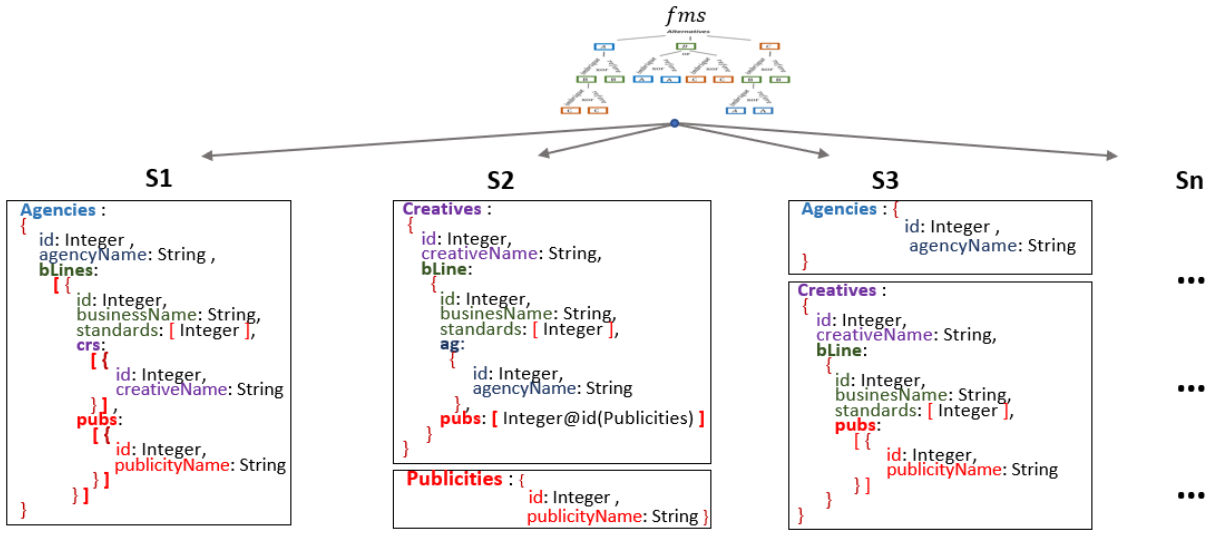


FIGURE 5.12: Exemple d'une alternative AJSchéma basée sur le modèle UML de la Figure 5.11

- $ce_j = 1$ par imbrication : un document dont les attributs correspondent à te_j.
- $ce_j = *$ par imbrication : un tableau de documents que correspondent à te_j.
- $ce_j = 1$ par référence : type Integer@id<Tx> dont $Tx = te_j$.
- $ce_j = *$ par référence : tableau de Integer@id<Tx> dont $Tx = te_j$.

La matérialisation par imbrication induit un niveau de profondeur dans la structure. Le type imbriqué peut être étendu à son tour si la classe à laquelle il correspond a des associations et au moins l'une d'elles est matérialisée. Le référencement peut se produire à tout niveau et force l'existence d'une collection du type te_j.

Les données modélisées sur la Figure 5.11 peuvent être représentées par la structure AJSchéma décrite par la Figure 5.12. Le schéma contient trois collections `Agencies, Owners,` et `Creatives`. Le type de la collection `Agencies` est formé par les attributs `agencyName` et `id`, correspondant au type *tAgency*, et par les attributs `bLines` et `ows`, correspondant respectivement aux matérialisations des associations `r1` et `r3`.

Dans notre exemple, l'attribut `bLines` représente l'association *1-à-plusieurs* `r1` en imbriquant des documents. Son type est donc un tableau de documents de type `tBusinessLines`. L'attribut `ows` correspond au rôle de la relation `r3`. La cardinalité *1-à-plusieurs* et une représentation par référencement conduisent à un attribut de type tableau d'entiers. Le référencement force l'existence d'une collection de type `tOwner`.

5.5 Conclusion

Notre contribution est motivée par des questions de qualité dans des bases de données orientées documents. Nous nous concentrons sur la structuration des données dans les documents JSON, supportée par MongoDB. La flexibilité offerte par ces systèmes est appréciée car il est facile de représenter des données semi-structurées. Cependant, cette flexibilité a un coût dans la performance, le stockage, la lisibilité et la maintenabilité de la base et de ses applications. Les choix de structuration des données relèvent de décisions de conception très importantes et ne doivent pas être négligées.

Dans ce chapitre, nous avons brièvement décrit notre projet SCORUS, qui vise à aider l'utilisateur à clarifier les possibilités de structuration des données et à fournir des métriques permettant de prendre des décisions de manière plus consciente.

Nous avons donné un aperçu de chacune des étapes de SCORUS, à savoir la génération automatique de schémas, l'évaluation des métriques et l'analyse des schémas. Nous avons défini une abstraction de schéma nommée AJSchéma pour raisonner sur la structure arborescente sous-jacente aux données.

Dans la suite nous traitons en profondeur la génération d'alternatives de structuration et la définition et l'évaluation automatique de métriques structurelles. Les définitions et les algorithmes sont présentés dans les chapitres 6 et 7. La validation et la mise en œuvre sont développées dans le chapitre 8.

"Il faut toujours avoir des chaos en soi pour donner naissance à une étoile dansante."

Friedrich Nietzsche

6

Génération d'AJSchémas

.

6.1 Introduction

La flexibilité des modèles semi-structurés supportés par les systèmes NoSQL orientés documents ouvre la porte à de multiples de possibilités de structuration. Cependant, ces alternatives sont nombreuses et très difficiles à envisager, à comprendre et à gérer en raison de l'absence de schéma. De plus, cette absence de schéma induit une dépendance très forte des alternatives au contexte métier, au code de l'application et aux requêtes. Cette dépendance rend complexe l'évolution de la solution, car même un changement subtil, peut masquer les effets à grande échelle difficiles à détecter et à contrôler.

Si nous considérons l'information à traiter et nous essayons de la modéliser, dans un modèle conceptuel UML par exemple, il est possible de générer plusieurs alternatives semi-structurées valides ou invalides selon les besoins. Avec très peu d'information à modéliser, le nombre d'alternatives à considérer est assez intéressant. Lorsque l'information est suffisamment grande, les possibilités sont si nombreuses que des alternatives de structuration potentiellement optimales peuvent être écartées par un utilisateur car, par exemple, il n'a pas les moyens d'analyser toutes les options, il n'a pas assez de temps pour effectuer une analyse approfondie, ou parce que, involontairement, un facteur clé n'a pas été pris en compte.

Considérer une nouvelle information, aussi simple soit-elle, peut donc impliquer de considérer un grand nombre de nouvelles possibilités de structuration dont les impacts sont actuellement difficiles à tracer. Si les différences et les points communs entre ces alternatives pouvaient être identifiés et modélisés, alors un ensemble de possibilités pourrait être obtenu de manière contrôlée permettant de mettre en évidence les alternatives les plus avantageuses en fonction de besoins.

Cette modélisation peut être réalisée avec les modèles de caractéristiques. Ces modèles sont utilisés dans le domaine des lignes de produits logiciels pour exprimer les variantes entre les différents produits à construire et contrôler l'explosion des possibilités qui peuvent survenir [18, 19, 82, 83]. Dans le cas des alternatives semi-structurées, les différences entre ces alternatives peuvent être traitées comme des variantes soumises à des directives de modélisation semi-structurées orientées documents telles que l'imbrication, le référencement, la profondeur et la duplication.

Dans ce chapitre, nous présentons notre proposition concernant la génération d'alternatives semi-structurées orientées documents. Nous proposons d'exprimer les concepts du contexte à travers un modèle conceptuel UML. Nous utilisons les éléments de ce modèle pour modéliser les variations entre des alternatives semi-structurées à travers un modèle de caractéristiques. Ce modèle de caractéristiques est construit grâce à notre algorithme *AMISS*. Par construction, ce modèle fournit des alternatives de semi-structuration valides selon des critères établis. L'utilisateur peut interagir

dans cette modélisation en établissant des restrictions pour forcer ou interdire des alternatives qui sont considérées à la base par *AMISS*.

Ensuite nous dérivons chaque alternative de structuration afin de construire une structure permettant d'analyser et d'évaluer le schéma avec l'ensemble de métriques structurelles du chapitre 7. Nous proposons les algorithmes *ADJusT* et *ADApt*. L'algorithme *ADJusT* interprète chaque alternative de structuration afin de générer une structure arborescente, appelée AJTree, pour l'analyse automatique de la structure. L'algorithme *ADApt* construit l'AJSchéma correspondant, permettant de donner une représentation textuelle à l'utilisateur. Nous créons des schémas avec des collections dont les documents sont homogènes.

Dans la section 6.2, nous présentons un exemple de motivation illustrant la flexibilité et l'évolution des alternatives de modélisation. Un contexte de variabilité basé sur les modèles de caractéristiques est présenté à la section 6.3. Dans les sections 6.4 et 6.5, nous décrivons la modélisation des alternatives semi-structures basées sur un modèle UML. D'abord nous introduisons la modélisation pour une association UML, section 6.4, et ensuite, nous présentons l'algorithme *AMISS* (section 6.5) permettant de modéliser un ensemble d'alternatives de structuration d'un modèle UML. Dans la section 6.6, nous introduisons les algorithmes *ADJusT* et *ADApt* qui permettent d'interpréter chaque alternative de structuration dans un AJTree et un AJSchéma.

6.2 Flexibilité et évolution dans la modélisation semi-structurée

Nous sommes intéressés par la flexibilité d'une modélisation semi-structurée, le grand nombre d'alternatives à gérer et l'explosion d nombre d'alternatives qui peut émerger lorsque de nouvelles informations sont considérées. Nous sommes particulièrement intéressés par une approche permettant de contrôler systématiquement ces aspects. Dans cette section, nous illustrons à travers un exemple, dans un contexte de marketing, des aspects de la flexibilité et l'explosion du nombre d'alternatives semi-structurées en considérant un modèle conceptuel UML.

Nous considérons un contexte initial avec l'information des agences et des secteurs d'activités. Ce contexte est modélisé dans le modèle de classes UML introduit par la Figure 6.1. Le modèle est composé de deux classes et une association 1 .. *. La classe `Agency` représente une agence qui peut gérer plusieurs secteurs d'activités, représentés par la classe `BusinessLine` avec le rôle `bLines`. À son tour, un secteur d'activités est géré par une seule agence (rôle `ag`). Chaque classe fournit l'ensemble de ses attributs.

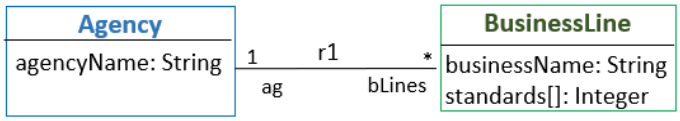


FIGURE 6.1: Modèle de classes pour des agences et des secteurs d'activités

Il y a plusieurs façons de modéliser les classes `Agency` et `BusinessLine`, et l'association r_1 dans une base orientée documents. Concernant les classes, chacune peut inspirer la création d'une collection dont les documents contiennent l'ensemble des attributs de sa classe. Dans notre cas d'exemple, une collection `Agencies` correspondra à une collection d'agences dont les documents contiendront les attributs respectifs de la classe `Agency`. À son tour, une collection `BusinessLines` correspondra à une collection de secteurs d'activité dont les documents contiendront les attributs respectifs de la classe `BusinessLine`.

Concernant l'association, elle peut se matérialiser soit par imbrication soit par référencement des informations de ses classes en considérant les deux sens de l'association. Cela introduit plusieurs façons de structurer les collections. Néanmoins, si nous définissons des *garanties de structuration*, cela peut nous permettre de réduire et contrôler les options qui peuvent émerger. Considérons les collections X et Y comme les collections correspondant aux classes d'une association.

Si nous nous intéressons à la complétude de l'information, des alternatives de structuration devront :

— garantir l'existence d'une collection qui est référencée et

— empêcher l'isolement d'une collection qui n'imbrique pas l'information de sa collection partenaire, soit en la référençant, soit en l'imbriquant dans une autre collection.

De plus, si nous nous intéressons aux références circulaires, des alternatives pourraient :

— empêcher les collections d'être référencées entre elles et

— empêcher une collection X imbriquant l'information de la collection Y, d'être référencée par cette dernière.

Pour empêcher l'isolement potentiel de deux collections, nous pouvons aussi envisager de :

— considérer une *collection-lien* contenant que des références aux deux collections si aucune d'elles imbrique ou référence les informations de sa partenaire.

En considérant ces *garanties de structuration* et de documents homogènes dans les collections, pour l'association r_1 de notre modèle UML de la Figure 6.1, nous obtenons huit alternatives possibles de structuration : cf. Figure 6.2.

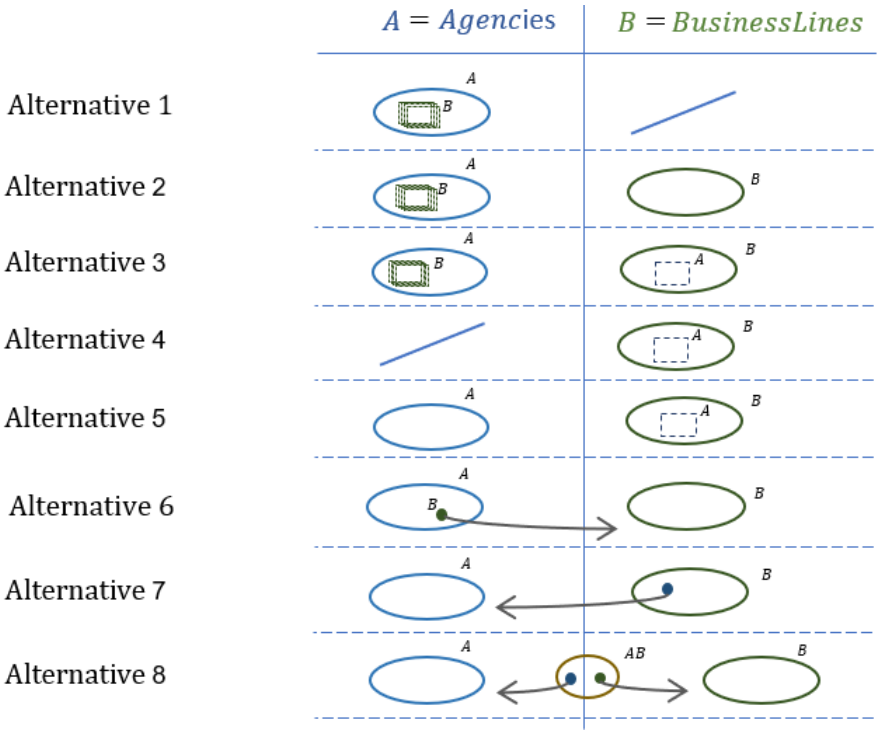


FIGURE 6.2: Alternatives de semi-structuration pour des agences et des secteurs d'activités

Alternative 1 : Collection Agencies, dont ses documents imbriquent un tableau de documents correspondant aux secteurs d'activités.

Alternative 2 : Collection Agencies, comme celle de l'alternative 1.
Collection BusinessLines. L'information des secteurs d'activités est dupliquée.

Alternative 3 : Collection Agencies, comme celle de l'alternative 1.
Collection BusinessLines, dont ses documents imbriquent un document correspondant à son agence.
Information des agences et des secteurs dupliquée.

Alternative 4 : Collection BusinessLines, dont ses documents imbriquent un document correspondant à son agence.

Alternative 5 : Collection Agencies. Collection BusinessLines, comme celle de l'alternative 4.
L'information des agences est dupliquée.

Alternative 6 : Collection BusinessLines. Collection Agencies, dont ses documents référencent leurs secteurs d'activités dans la collection BusinessLines.

Alternative 7 : Collection Agencies. Collection BusinessLines, dont chaque document référence son agence dans la collection Agencies.

Alternative 8 : Collection Agencies et collection BusinessLines.
Collection AB dont les documents contiennent les références à l'agence et à l'un de ses secteurs d'activité.

Les alternatives 6 et 7 respectent la *garantie de structuration* concernant l'existence d'une collection référencée. Les alternatives 2 et 5 respectent la *garantie de structuration* concernant l'isolement en imbriquant l'information de la collection isolée. L'alternative 8 propose une collection supplémentaire afin de référencer les deux collections isolées. Nous appellerons *collection-lien* une telle collection de référencement.

Notons que la duplication est autorisée, ce qui permet d'avoir l'information sous forme de collections et imbriquée en même temps. L'alternative de structuration 2 de notre exemple considère les secteurs d'activités sous la forme de la collection `BusinnessLines` et imbriqués dans leur agence dans la collection `Agencies`.

Il peut y avoir plus des alternatives pour ce modèle qui sont écartées à cause de *garanties de structuration* établies. Par exemple, une alternative de structuration avec les deux collections `Agencies` et `BusinnessLines` sans imbrication et ne se référençant pas l'une avec l'autre doit être écartée. Les deux collections seraient isolées et l'association $r1$ non représentée.

Maintenant, imaginons que les besoins de notre contexte changent. Il faut prendre en compte les designers qui seront en charge de créer des publicités pour les différents secteurs d'activités. Pour cela, une nouvelle classe, appelée *Creative*, est ajoutée à notre modèle UML et liée avec la classe *BusinessLines* par l'association $r2$ avec une cardinalité *...1 respectivement : cf. Figure 6.3.

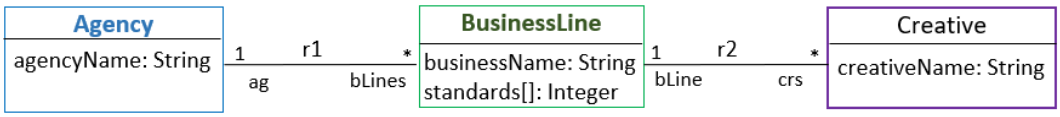


FIGURE 6.3: Évolution du modèle de classes de la Figure 6.1

Cette nouvelle classe *Creative* ajoute de nouvelles alternatives de structuration et implique de mettre à jour celles déjà existantes afin qu'elles considèrent la nouvelle information. Comme résultat, une explosion du nombre d'alternatives de structuration émerge.

Des directives comme nos *garanties de structuration* peuvent aider à réduire et contrôler l'espace des solutions en proposant des alternatives de structuration valides. En les considérant, nous pouvons trouver environ une centaine d'alternatives. Un nombre d'alternatives trop grand à analyser et comparer manuellement.

Des besoins particuliers peuvent être considérés par les utilisateurs afin de réduire et cibler les alternatives. Comme par exemple, éviter de faire des références à partir de niveaux imbriqués, imbriquer une information particulière dans une collection spécifique et obliger l'existence d'une

collection, entre autres.

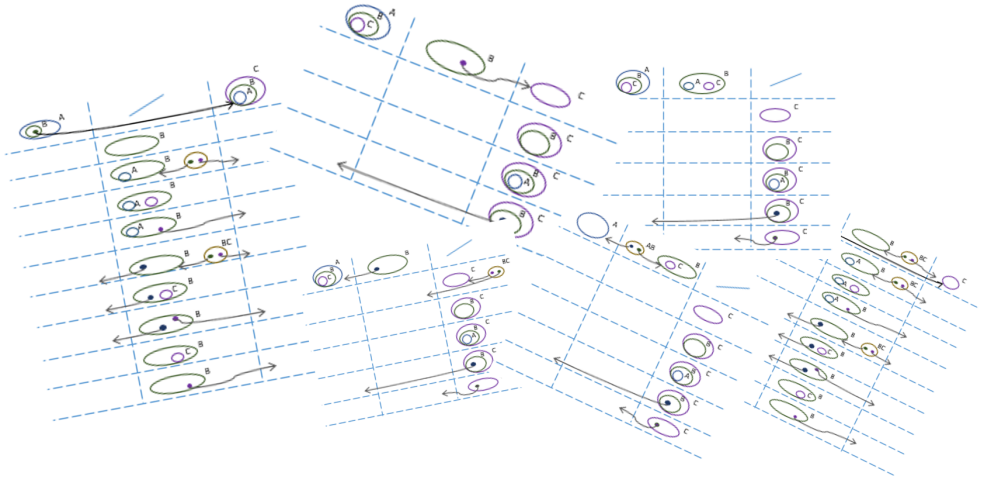


FIGURE 6.4: Explosion d'alternatives

Afin de contrôler autant d'alternatives de structuration, nous proposons d'analyser les différences et les points communs entre celles-ci, les identifier et les modéliser à l'aide de modèles de caractéristiques. Il s'agit d'une stratégie utilisée dans le domaine des lignes de produits logiciel qui permet d'exprimer les variantes entre les produits, alternatives de structuration dans notre cas, en permettant de mettre en évidence celles qui sont valides.

Nous présentons dans la section suivante la façon de modéliser et contrôler cette variabilité avec les modèles de caractéristiques. Nous appliquons ces concepts à la modélisation des alternatives de structuration orientées vers les documents.

6.3 Variabilité et modèles de caractéristiques

Dans le domaine des lignes de produits logiciel, un problème auquel il est confronté est l'expression de la variabilité, la différence entre les produits à construire. Il existe de nombreuses stratégies pour représenter cette variabilité, la plus utilisée étant la variabilité basée sur des modèles de caractéristiques. Pour notre proposition, nous avons décidé d'utiliser cette stratégie parce qu'elle est à la fois simple et puissante. L'objectif de cette section est de présenter les concepts de base de la variabilité qui seront utilisés par la suite au cours de ce chapitre.

La stratégie de variabilité basée sur des modèles de caractéristiques est composée de trois étapes telles que la modélisation, la configuration et la dérivation. La première étape consiste à modéliser la variabilité identifiant les variations des caractéristiques que les produits peuvent présenter. Une fois cette variabilité modélisée, la configuration consistera à identifier chacune des combinaisons possibles de ces caractéristiques. Ensuite, la dérivation prend chacune des configurations et interprète les caractéristiques sélectionnées pour créer les produits finaux correspondants.

6.3.1 Modélisation

La variabilité est donc modélisée par un modèle de caractéristiques [80, 84, 85, 86]. Un modèle de caractéristiques, désigné par fm, est une représentation des options et des contraintes autorisées pour un produit (voiture, logiciel, ...) en permettant d'établir un ensemble de configurations différentes et valides.

Ces options sont définies par un *ensemble de caractéristiques* $\mathscr{F}$ et distribuées dans une structure arborescente. Cette structure est composée d'une caractéristique *racine* et d'un ensemble de caractéristiques descendantes (chemins et feuilles) en permettant de former des groupes de caractéristiques. Des contraintes entre les caractéristiques ($\mathscr{D}$) sont fournies pour forcer ou interdire une sélection de caractéristiques si une autre ou d'autres caractéristiques sont choisies.

Définition 6.1. Modèle de caractéristiques

Un modèle de caractéristiques est défini comme un quadruplet tel que [85, 86] :

$$fm = (\mathscr{F}, \mathscr{L}, rc, \mathscr{D})$$

où,

$\mathscr{F}$ est l'ensemble des caractéristiques
$\mathscr{L}$ représente les liens qui relient les caractéristiques
rc représente la caractéristique racine telle que $rc \in \mathscr{F}$
$\mathscr{D}$ est l'ensemble des contraintes entre les caractéristiques de la forme $X \implies Y$

La table de la Figure 6.5 illustre les différents types de liens qui permettent de désigner les variations entre les caractéristiques :

— *Obligatoire* : Une caractéristique doit être choisie si le parent est choisi. Les caractéristiques obligatoires représentent celles qui sont présentes dans toutes les configurations.
— *Optionnel* : Une Caractéristique peut être choisie si le parent est choisi. Quand une caractéristique n'est pas choisie, toute la branche dont elle est racine est rejetée.
— *Groupe OR* : Groupe des caractéristiques dans lequel une ou plusieurs caractéristiques du groupe doit être choisi si le parent du groupe est choisi.
— *Groupe XOR* : Groupe des caractéristiques dans lequel une caractéristique du groupe doit être choisie si le parent du groupe est choisi.

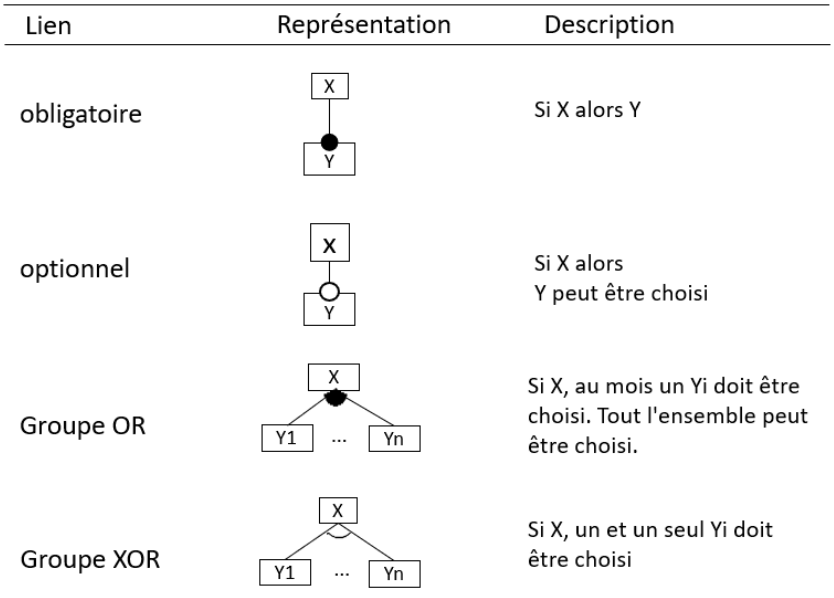


FIGURE 6.5: Liens entre les caractéristiques d'un modèle de caractéristiques (fm)

La Figure 6.6 développe un modèle de caractéristiques pour une *Liseuse*, nommé $fm_{liseuse}$. Ce

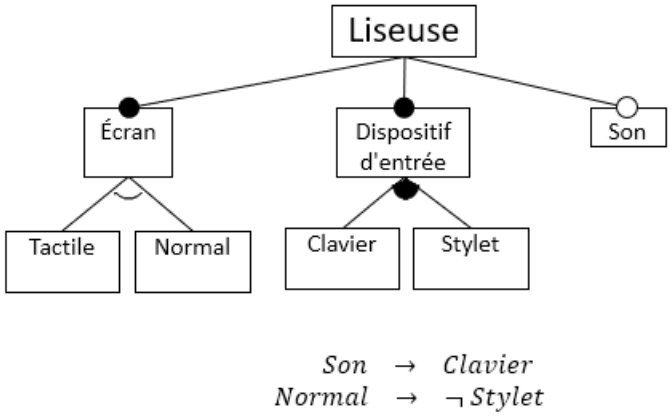


FIGURE 6.6: Modèle de caractéristiques pour une Liseuse

modèle considère l'ensemble de caractéristiques suivant :

$$\mathscr{F}(fm_{liseuse}) = \{\text{Liseuse}, \text{Ecran}, \text{Dispositif d'entrée}, \text{Tactile}, \text{Normal}, \text{Clavier}, \text{Stylet}, \text{Son}\}$$

La structure arborescente considère la caractéristique *Liseuse* comme la racine. Cela implique deux sous-caractéristiques obligatoires, à savoir *Écran* et *Dispositif d'entrée*, et une sous-caractéristique optionnelle nommée *Son*. À son tour, la caractéristique *Écran*, implique qu'il faut choisir entre un écran *Tactile* ou un écran *Normal*. Par rapport au *Dispositif d'entrée*, il est possible de choisir soit le *Clavier*, soit le *Style* ou les deux.

Le modèle $fm_{liseuse}$ considère aussi un ensemble de deux contraintes $\mathscr{D}(fm_{liseuse})$, telles que :

— forcer la sélection d'un clavier dans le cas où le son est choisi et,
— empêcher la sélection d'un dispositif d'entrée Stylet si un écran normal est choisi.

6.3.2 Processus de configuration

Une fois créé le modèle de caractéristiques fm, l'étape de configuration consiste à identifier et à sélectionner les combinaisons entre ses caractéristiques. Les combinaisons ou les configurations valides seront donc celles qui sont conformes aux contraintes du modèle fm et aux contraintes du domaine d'application. Le modèle fournira alors sa sémantique sous forme d'un ensemble de configurations valides $\|fm\|$. Chaque configuration valide sera composée d'un sous-ensemble de noms des caractéristiques choisies du modèle.

Définition 6.2. Configuration
Une configuration C est définie comme un ensemble de caractéristiques sélectionnées parmi l'ensemble des caractéristiques du modèle $\mathscr{F}(fm)$.

$$C(fm) = \{f | f \in \mathscr{F}(fm)\}$$

Une configuration est valide si et seulement si elle est conforme aux contraintes $\mathscr{D}(fm)$ définies dans le modèle.

Définition 6.3. Sémantique
La sémantique d'un modèle de caractéristiques est définie comme l'ensemble de toutes les configurations valides :

$$\|fm\| = \{C | C \text{ est valide } \}$$

Concernant le modèle de caractéristiques pour la liseuse $fm_{liseuse}$ (cf. Figure 6.6) sa sémantique ou ensemble de configurations $\|fm_{liseuse}\|$ est composé de 7 configurations valides, parmi lesquelles nous pouvons trouver :

$$C_1(fm_{liseuse}) = \{\text{Liseuse}, \text{Ecran}, \text{Dispositif d'entrée}, \text{Tactile}, \text{Stylet}\}$$

$$C_2(fm_{liseuse}) = \{\text{Liseuse}, \text{Ecran}, \text{Dispositif d'entrée}, \text{Son}, \text{Tactile}, \text{Clavier}\}$$

6.3.3 Processus de dérivation

Une fois l'ensemble des configurations identifié, chacune est interprétée pour créer une version du produit correspondant aux caractéristiques choisies pour celle-ci.

Définition 6.4. Produit final
Un produit final p est défini comme le dérivation d'une configuration appartenant à l'ensemble des configurations du modèle de caractéristiques. L'ensemble des produits finaux correspond à l'ensemble des configurations :

$$\mathscr{P}(fm) = \{p \mid p_i \leftarrow deriver(C_i) \wedge C_i \in \|fm\|\}$$

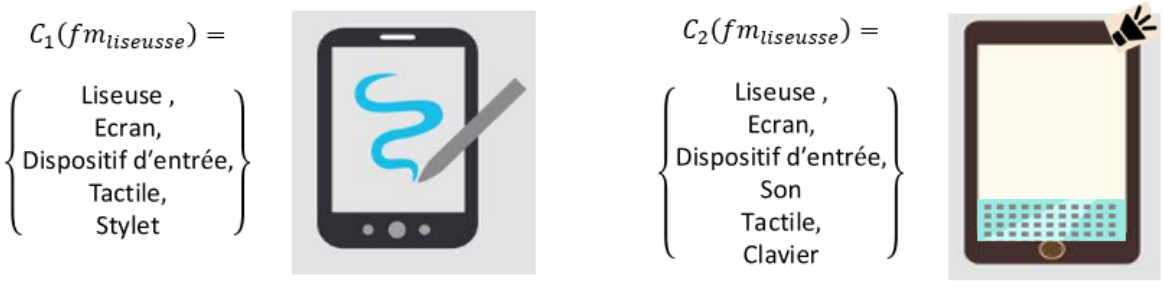


FIGURE 6.7: Dérivation de deux configurations du modèle de caractéristiques $fm_{liseuse}$

La Figure 6.7 introduit deux des 7 produits finaux du modèle de caractéristiques de notre liseuse, $\mathscr{P}(fm_{liseuse})$. Les produits finaux p_1 et p_2 correspondent aux dérivations des configurations C_1 et C_2. La dérivation de la configuration C_1 permet de créer une liseuse tactile permettant d'utiliser un stylet. D'autre part, la configuration C_2 permet de créer une liseuse tactile avec un clavier et une entrée sonore.

6.3.4 Modèles de caractéristiques et structuration orientée document

Comme nous l'avons mentionné précédemment, la flexibilité de structuration offerte par les systèmes NoSQL orientés documents, ouvre la porte à un grand nombre d'alternatives qui sont très difficiles à analyser et contrôler. Néanmoins il existe des facteurs clés qui permettent d'identifier des différences et des points communs parmi les alternatives possibles de structuration.

Un modèle de caractéristiques peut donc aider à modéliser ces différences de structuration et contrôler l'explosion des possibilités qui peuvent survenir. Les différences entre les alternatives de structuration peuvent être traitées comme des variantes d'un produit final, que dans ce cas sera un schéma. Ces variantes peuvent être soumises à des directives de structuration orientées documents telles que l'imbrication, le référencement, la profondeur et la duplication. Les configurations résultantes d'une telle modélisation seront donc un ensemble d'alternatives à dériver pour leur l'analyse.

Dans les sections suivantes, nous présentons notre proposition concernant la génération d'alternatives de structuration orientées documents. Nous proposons d'exprimer le contexte dans un modèle conceptuel UML et d'utiliser ses éléments pour créer un modèle de caractéristiques qui fournit un ensemble de configurations correspondant à des alternatives possibles de structuration. La dérivation de chaque configuration, section 6.6, crée un AJSchéma et une structure arborescente permettant par la suite de l'analyser et de l'évaluer avec les métriques structurelles.

6.4 Modélisation orientée document d'une association UML

Nous abordons la modélisation orientée documents d'un modèle UML, en analysant d'abord dans cette section la modélisation des alternatives de structuration pour une association UML. Cela est notre cas d'analyse de base qui nous permet ultérieurement d'analyser les alternatives de structuration d'un modèle UML. Nous utilisons une approche de variabilité basée sur des modèles de caractéristiques (VBMC) en considérant des aspects clés dans les systèmes NoSQL orientés documents : *imbrication*, *référencement*, *niveaux de profondeur* et *duplication*.

D'abord, nous définissons les éléments d'une association r d'un modèle UML, illustrés dans la Figure 6.8, comme suit :

Soit mU une paire (E,R) où :

- $E = \{e_1,...,e_n\}$ correspond à l'ensemble des classes
- $R = \{r_1,...,r_n\}$ correspond à l'ensemble des associations
- $R(e_i) = \{r_1,...,r_n\}$ c'est l'ensemble des associations de la classe e_i

Soit r une association :

- $E(r) = \{e_1,e_2\}$ correspond aux classes de chaque extrémité de l'association r. $e_1 \neq e_2$
- $A(e_i) = \{a_1,...,a_n\}$ désigne l'ensemble des attributs de la classe $e_i \in E$.
  Chaque $a_i \in A$ est défini par la paire $a_i.name$:$a_i.type$.
- $r.name$ est le nom de l'association r,
- $r.ce_i$ correspond à la cardinalité de l'association r par rapport la classe e_i,
- $r.role_i$ désigne le rôle de l'association r par rapport la classe e_i

Nous nous limitons aux associations binaires sans attributs

Pour mieux spécifier comment les attributs des documents au sein d'une collection sont regroupés, imbriqués et référencés par rapport aux classes, nous introduisons la définition de type correspondant à une classe.

Définition 6.5. Type d'une classe
Le type d'une classe e_i est défini comme l'ensemble de ses attributs et un identifiant par défaut :

$$te_i = \{A(e_i), a_{id}\}$$

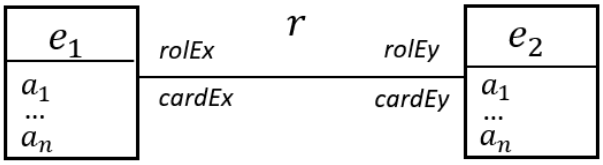


FIGURE 6.8: Éléments d'une association UML

En utilisant tous ces éléments, nous considérons de lignes directrices de modélisation orientées documents basées sur un modèle UML fournies par le méta-modèle SINGULar, cf. Figure 5.10, notamment :

Directrice modélisation 1 : La classe $e_i \in E$ peut déclencher la création d'une collection.

Directrice modélisation 2 : Une collection déclenchée par e_i contient documents du type te_i au premier niveau.

Directrice modélisation 3 : L'association r peut être matérialisée par l'imbrication ou référencèrent de e_i.

A partir de ces directrices, nous proposons de modéliser les huit alternatives de structuration pour la association *r* de la Figure 6.8 telles que celles introduites dans la section 6.2 de ce chapitre :

— *Alternative*1 : Une collection avec l'information de e_1 en imbriquant l'information de e_2

— *Alternative*2 : Une collection avec l'information de e_1 qu'imbrique l'information de e_2 et une collection avec l'information de e_j sans imbrication ni référencement

— *Alternative*3 : Une collection avec l'information de e_1 qu'imbrique l'information de e_2 et une collection avec l'information de e_2 qu'imbrique l'information de e_1

— *Alternative*4 : Une collection avec l'information de e_1 sans imbrication ni référencement et une collection avec l'information de e_2 qu'imbrique l'information de e_1

— *Alternative*5 : Une collection avec l'information de e_2 en imbriquant l'information de e_1

— *Alternative*6 : Une collection avec l'information de e_1 et des références vers l'information de e_2, et une collection avec l'information de e_2 sans références ni imbrication

— *Alternative*7 : Une collection avec l'information de e_1 sans références ni imbrication et une collection avec l'information de e_2 et des références vers l'information de e_1

— *Alternative*8 : Deux collections références ni imbrication avec l'information de e_1 et e_2 et une collection avec des références vers e_1 et e_2

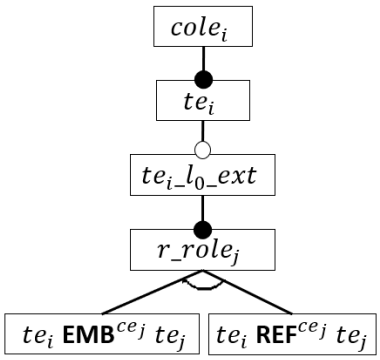


FIGURE 6.9: Modèle fms d'une collection basée sur un classe e_i avec une association r

6.4.1 Modélisation de variations entre alternatives

Nous modélisons les variations de ces alternatives orientées documents avec un modèle de caractéristiques fms permettant d'exprimer les options d'imbrications et référencement. Pour cela, nous nous appuyons sur les lignes directrices de modélisation fournies par le métamodèle SINGULar, qui guide la création d'un schéma (cf. Figure 5.10)

Nous commençons par la modélisation de la variabilité des alternatives de structuration d'une collection déclenchée par une classe e_i. Nous proposons un modèle de caractéristiques fms_{col}, introduit par la Figure 6.9, où ses caractéristiques et variations sont définis comme suit :

- Le nom de la collection définit la caractéristique racine par $cole_i$.
- Le type de documents de la collection au premier niveau est défini comme une caractéristique fille obligatoire de $cole_i$ nommée te_i.
- Une extension optionnelle du type te_i au niveau l_0 est défini comme une caractéristique optionnelle $te_i\_l_0\_ext$ et fille de la caractéristique te_i.
- Le rôle de l'association utilisé pour cette extension est défini par la caracterisque obligatoire r_role_j. Le rôle c'est celui correspondant à la classe opposée e_j.
- Les alternatives de matérialisation de l'association r sont définies par un groupe OR dépendant de la caractéristique r_role_j. Ce groupe a deux caractéristiques filles nommées $te_i\,EMB^{ce_j}\,te_j$ et $te_j\,REF^{ce_j}\,te_j$.
- La caractéristique $te_i\,EMB^{ce_j}\,te_j$ indique l'*imbrication* dans te_i d'un ou de plusieurs documents du type te_j. Une cardinalité ce_j plusieurs indique l'imbrication d'un tableau de documents, une cardinalité 1 indique l'imbrication d'un document.
- La caractéristique $te_i\,REF^{ce_j}\,te_j$ indique le *référencement* depuis un type te_i étendu, d'un ou plusieurs documents du type te_j selon la cardinalité ce_j. Une cardinalité plusieurs indique qu'il aura un tableau de références, une cardinalité 1 indique une seule référence.

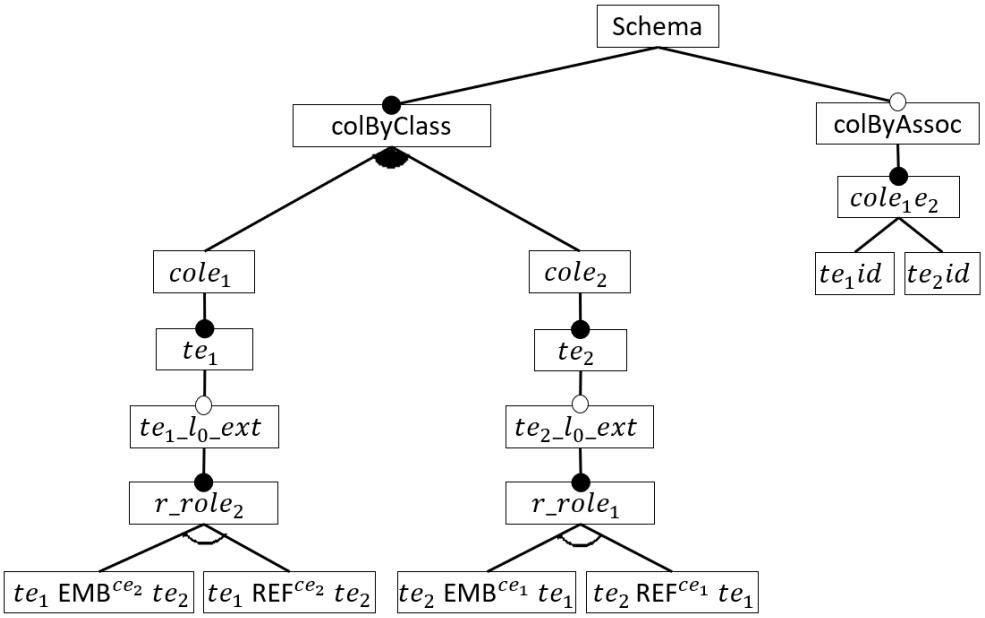


FIGURE 6.10: Modèle de caractéristiques d'une association UML

Basé sur le modèle de caractéristiques d'une collection, nous proposons de modéliser la variabilité des alternatives d'une association dans le modèle de caractéristiques nommé fms_r. La Figure 6.10 introduit la structure arborescente du modèle fms_r , et qui est construite par l'Algorithme 1 en considérant :

— La caractéristique racine est nommée *Schema*.

— La racine a deux caractéristiques filles, une obligatoire nommée *colByClass* et une optionnelle nommée *colByAssoc*.

— La caractéristique *colByClass* détermine la ou les collections qui seront créées en fonction des classes du modèle UML. *colByClass* définit un groupe OR avec une caractéristique fille par classe $e_i \in E$. Leurs noms suivent le patron $cole_i$ (cf. Figure 6.9) et indiquent la collection par classe qui peut être créée.

— Les descendants de $cole_i$ correspondent au modèle fms_{col}.

— La caractéristique *colByAssoc* définit une *collection-lien* nommée $cole_ie_j$ qui référence documents de type te_i et te_j au premier niveau.

Algorithme 1 Génération du modèle de caractéristiques pour une association

Input: association r
Output: Modèle de caractéristiques fm_r

1: **procedure** GENERATEFMASSOCIATION(r) : fms_r
2: $fms_r \leftarrow setRoot(fms_r, "schema")$
▷ addFeature(fm,featureParent,featureChild,type)
3: $fms_r \leftarrow addFeature(fms_r, "schema", "colByClass", Mandatory)$
4: $fms_r \leftarrow addFeature(fms_r, "schema", "colByAssoc", Optional)$
5: **foreach** $e_i \in E$ **do**
6: $e_j \leftarrow$ getTarget(r, e_i)

7: $cole_i \leftarrow$ "col" + $e_i.name$
8: $te_i \leftarrow$ "t" + $e_i.name$
9: $ext \leftarrow$ "t" + $e_i.name$ + "_" + l_0 + "_ext"
10: $relrol \leftarrow r.name$ + "_" + $role_j$
11: $emb \leftarrow$ "t" + $e_i.name$ + " EMB" + ce_j + " " + te_j
12: $ref \leftarrow$ "t" + $e_i.name$ + " REF" + ce_j + " " + te_j

13: $fms_r \leftarrow addFeature(fms_r, colByClass, cole_i, OR)$

14: $fms_r \leftarrow addFeature(fms_r, cole_i, te_i, Mandatory)$
15: $fms_r \leftarrow addFeature(fms_r, te_i, ext, Optional)$
16: $fms_r \leftarrow addFeature(fms_r, ext, relrol, Mandatory)$
17: $fms_r \leftarrow addFeature(fms_r, relrol, emb, XOR)$
18: $fms_r \leftarrow addFeature(fms_r, relrol, ref, XOR)$
19: **end for**
20: $colVue \leftarrow$ "col" + $e_i.name$ + $e_j.name$
21: $fms_r \leftarrow addFeature(fms_r, "colByClass", "2classes", Mandatory)$
22: $fms_r \leftarrow addFeature(fms_r, "2classes", "colVue", Mandatory)$

23: $\mathscr{D}(fms_r) \leftarrow createConstraints(fms_r)$ ▷ contraintes
24: $fms_r \leftarrow addConstraints(fms_r, \mathscr{D})$
25: **return** fms_r
26: **end procedure**

6.4.2 Définition des structurations valides

Le modèle de caractéristiques introduit précédemment génère plusieurs alternatives de structuration. Nous avons établi un ensemble de contraintes pour nous assurer que, par construction, toutes les configurations fournies par le modèle de caractéristiques sont valides. Ces contraintes correspondent aux garanties de structuration présentées dans la section 6.2 de ce chapitre. Celles-ci s'intéresseront donc à l'exhaustivité des informations et au traitement des références circulaires. Rappelons que nos alternatives permettent la duplication de documents et considèrent des documents homogènes.

Nous avons classé les contraintes $\mathscr{D}(fms)$ dans cinq groupes :

- Contraintes d'isolement $\mathscr{D}^I$: empêchent l'isolement d'une collection de type te_i qui n'imbrique pas de documents de type te_j documents.
- Contraintes d'existence $\mathscr{D}^E$: garantissent l'existence d'une collection de type te_i dont les documents sont référencés.
- Contraintes de boucles imbriquées $\mathscr{D}^{BI}$: empêchent les collections d'être référencées entre elles.
- Contraintes de références bouclées $\mathscr{D}^{RB}$: empêcher une collection qui imbrique des documents de type te_i, d'être référencée par une autre collection du même type.
- Contraintes de *liens* d'association $\mathscr{D}^L$: considèrent une collection *lien* ne contenant que des références aux deux collections si elles sont isolées.

Définition 6.6. Contraintes d'isolement

Une contrainte d'isolement $d^I(e_i)$ garantit dans un modèle de caractéristiques tel que fms, que si une collection de type te_i, au premier niveau, n'a pas d'extension pour son association r_n vers te_j, alors cette association doit être considérée soit depuis une collection *lien* soit depuis un niveau imbriqué appartenant à une autre collection (cf. Figure 6.11).

En considérant $\mathscr{D}^I(fms) = \{d^I(te_i) | e_i \in E(r_n), r_n \in R\}$ comme l'ensemble des contraintes d'isolement appartenant au modèle fms_r, une contrainte est définie comme une paire de propositions logiquement équivalentes telles que :

$$d^I(te_i) = (p \implies q)$$

$p = te_i \wedge \neg r_n\_role_j$

$q = cole_i e_j \; (\; (\vee f \doteq r_n \;) \mid f \in fms, f \doteq r_n \wedge f \neq r_n\_role_j, E(r_n) = e_i, ej)$

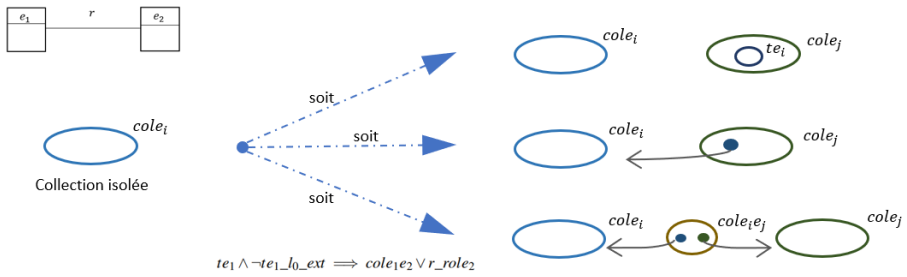


FIGURE 6.11: Exemple de contrainte d'isolement pour la collection $cole_i$ de type te_i

Définition 6.7. Contraintes d'existence

Dans un modèle de caractéristiques tel que fms, les contraintes d'existence $\mathscr{D}^E$ garantissent que si un type te est référencé, alors une collection de ce type doit exister. Une contrainte d'existence $d^E(te_i)$ est définie donc comme une expression propositionnelle telle que :

$$\mathscr{D}^E(fms) = \{d^E(te_i) | f \in fms,\ f \doteq te_j\ REF\ te_i,\ (e_i, e_j) \in E(r_n)\}$$

où

$$d^E(te_i) = (te_j REF^{r_n.ce_i} te_i \implies cole_i)$$

Définition 6.8. Contraintes de références bouclées

Une contrainte de références bouclées $d^{BR}(e_i)$ garantit dans un modèle de caractéristiques tel que fms, que si un te_i réfère le type te_j, alors te_j ne peut pas référencer te_i.

En considérant $\mathscr{D}^{BR}(fms) = \{d^{BR}(r_n) | f \in fms,\ f \doteq te_j\ REF\ te_i,\ (e_i, e_j) \in E(r_n)\}$ comme l'ensemble des contraintes de références bouclées appartenant au modèle fms_r, une contrainte est définie comme une paire de propositions logiquement équivalentes telles que :

$$d^{BR}(r_n) = (te_i ref^{ce_j} te_j \implies \neg te_j ref^{ce_i} te_i)$$

Définition 6.9. Contraintes de boucles imbriquées

Dans un modèle de caractéristiques tel que fms, les contraintes de boucles imbriquées $\mathscr{D}^{BI}$ garantissent qu'un type ne peut pas être imbriqué par le même type qu'il réfère (cf. Figure 6.12).

Une contrainte de boucles imbriquées $d^{BI}(e_i)$ est définie comme une expression où : si un te_i

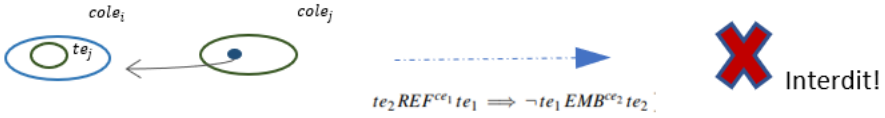


FIGURE 6.12: Exemple de contrainte $\mathscr{D}^{BI}$ pour les collections $cole_i$ de type te_i et $cole_j$ de type te_j

réfère le type te_j, alors te_j ne peut pas imbriquer te_i.

$$\mathscr{D}^{BI}(fms) = \{d^BI(te_i) | f \in fms,\ f \doteq te_j\ REF\ te_i,\ (e_i, e_j) \in E(r_n)\}$$

où

$$d^BI(e_i) = (te_j REF^{r_n.ce_i} te_i \implies \neg te_i EMB^{r_n.ce_i} te_j)$$

Définition 6.10. Contraintes de liens

Dans un modèle de caractéristiques tel que fms, les contraintes de liens ($\mathscr{D}^L$) garantissent qu'une *collection-lien* ne réfère que des collections de type non étendu.

Une contrainte de lien $d^L(r_n)$ est définie comme une expression où : si le lien $cole_ie_j$ existe , alors la collection $cole_i$ non étendue et la collection $cole_j$ non étendue par l'association r_n doivent exister.

$$\mathscr{D}^L(fms) = \{d^L(r_n) | f \in fms,\ f \doteq cole_ie_j,\ (e_i, e_j) \in E(r_n)\}$$

où

$$d^L(r_n) = (cole_ie_j \implies (cole_i \wedge \neg r\_role_j) \wedge (cole_j \wedge \neg r\_role_i))$$

L'ensemble des contraintes du modèle de caractéristiques fms_r sera donc formé par les contraintes d'isolement, d'existence, de boucles imbriquées, de références bouclées et de lien telles que :

$$\mathscr{D}(fms_r) = \{\mathscr{D}^I, \mathscr{D}^E, \mathscr{D}^{BI}, \mathscr{D}^{BR}, \mathscr{D}^L\}$$

où

$$\begin{aligned}
\mathscr{D}^I &= \{\ te_1 \wedge \neg te_1_l_0_ext \implies cole_1e_2 \vee r_role_2 \\
&\quad\ te_2 \wedge \neg te_2_l_0_ext \implies cole_2e_1 \vee r_role_1\ \} \\
\mathscr{D}^E &= \{\ te_1 REF^{ce_2} te_2 \implies cole_2 \\
&\quad\ te_2 REF^{ce_j} te_1 \implies cole_1\ \} \\
\mathscr{D}^{BI} &= \{\ te_1 REF^{ce_2} te_2 \implies \neg te_2 EMB^{ce_1} te_1 \\
&\quad\ te_2 REF^{ce_1} te_1 \implies \neg te_1 EMB^{ce_2} te_2\ \} \\
\mathscr{D}^{BR} &= \{\ te_1 REF^{ce_2} te_2 \implies \neg te_2 REF^{ce_1} te_1\ \} \\
\mathscr{D}^L &= \{\ cole_1e_2 \implies (cole_1 \wedge \neg r_role_2) \wedge (cole_2 \wedge \neg r_role_1)\ \}
\end{aligned}$$

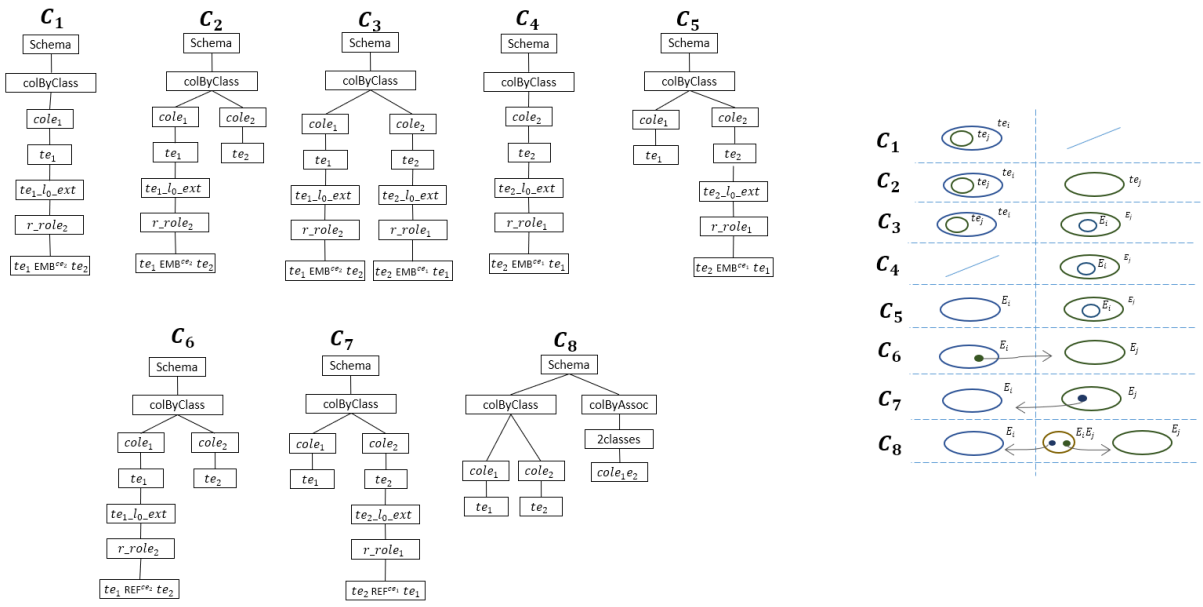


FIGURE 6.13: Vue arborescente des configurations du modèle fms_r correspondant la Figure 6.10

6.4.3 Configuration des alternatives

Compte tenu de ces restrictions, le modèle de caractéristiques fm_r fournit alors huit configurations pour une association UML qui nous intéressent, et qui correspondent aux alternatives de structuration présentées au début de cette chapitre. La Figure 6.13 illustre les caractéristiques sélectionnées pour chaque configuration sous forme d'une structure arborescente, rappelons qu'une configuration ne contient que le noms des caractéristiques choisies. Voici la sémantique $\|fm_r\|$ (configurations) du modèle fm_r :

$$\|fm_r\| = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$$

$$C_1 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, te_1\_l_0\_ext, r\_role_2, te_1 emb^{ce2} te_2\}$$

$$C_2 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, te_1\_l_0\_ext, r\_role_2, te_1 emb^{ce2} te_2, \mathbf{cole_2}, te_2\}$$

$$C_3 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, te_1\_l_0\_ext, r\_role_2, te_1 emb^{ce2} te_2, \mathbf{cole_2}, te_2, te_2\_l_0\_ext, r\_role_1, te_2 emb^{ce1} te_1\}$$

$$C_4 = \{Schema, ColByClass, \mathbf{cole_2}, te_2, te_2\_l_0\_ext, r\_role_1, te_2 emb^{ce1} te_1\}$$

$$C_5 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, \mathbf{cole_2}, te_2, te_2\_l_0\_ext, r\_role_1, te_2 emb^{ce1}, te_1\}$$

$$C_6 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, te_1\_l_0\_ext, r\_role_2, te_1 ref^{ce2} te_2, \mathbf{cole_2}, te_2\}$$

$$C_7 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, \mathbf{cole_2}, te_2, te_2\_l_0\_ext, r\_role_1, te_2 ref^{ce1} te_1\}$$

$$C_8 = \{Schema, ColByClass, \mathbf{cole_1}, te_1, \mathbf{cole_2}, te_2, ColByAssoc, 2classes, \mathbf{cole_1 e_2}\}$$

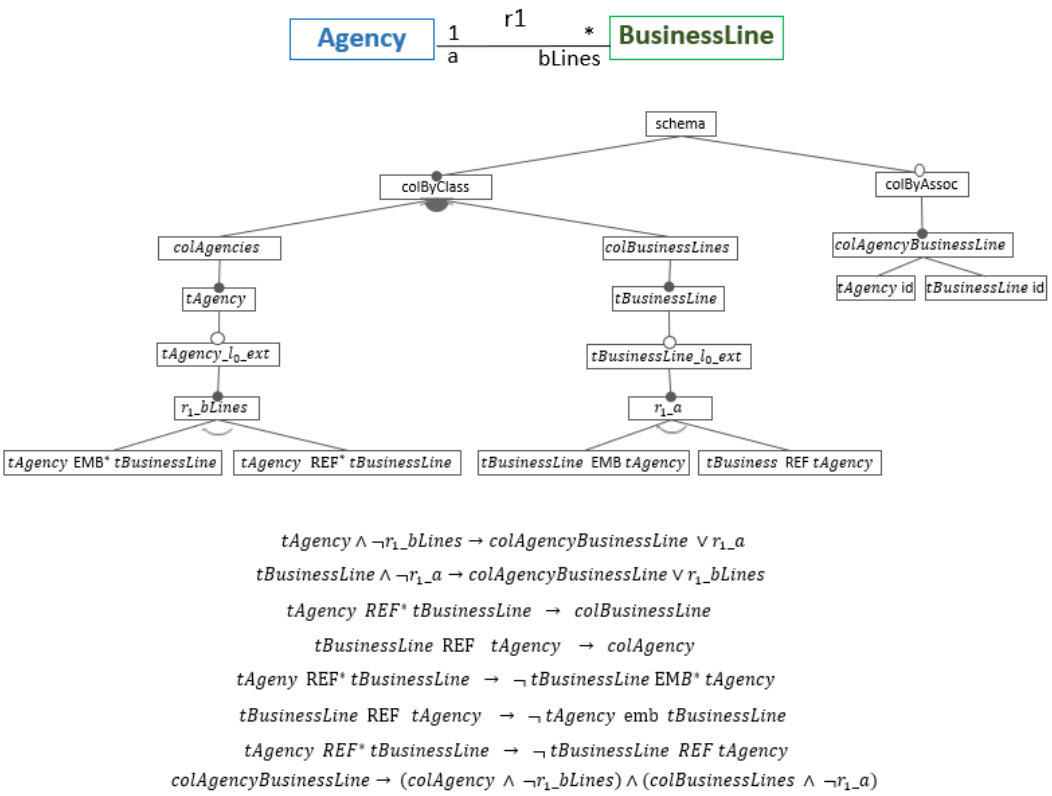


FIGURE 6.14: Modèle fms_r correspondant pour l'association de la Figure 6.1

Les cinq premières configurations (C_1 à C_5) correspondent aux combinaisons avec les choix des imbrications. Les trois dernières configurations (C_6 à C_8) correspondent aux choix des référencement. Les premières configuration C_1 et C_4 suivent le même patron, elles fournissent les caractéristiques pour créer une structure composée d'une seule collection qui permet l'imbrication du type restant. La configuration C_2 fournit la collection $cole_i$ comme celle de C_1 et une collection $cole_j$ qui n'admet pas l'extension de types. La configuration C_5 est construit avec le même principe de c_2. La configuration C_3 fournit deux collections, la collection $cole_i$ comme celle de C_1 et la collection $cole_j$ qui permet l'imbrication de types te_i. Les configurations C_6 et C_7 considèrent une collection sans extension et l'autre avec le référencement aux types correspondant. La dernière configuration considère deux collections sans extensions et une troisième qui référence ces types correspondants.

Revenant à notre exemple de la Figure 6.1, le modèle UML contient les classes Agency et BusinnessLines liées par l'association r_1 avec respectivement une cardinalité 1 ... * et des rôles ag ... bLines. Le modèle de caractéristiques fm_r pour cette association est introduit dans la Figure 6.14. L'ensemble des configurations résultant de ce modèle est généré par notre Algorithme 1 (section 6.4.1) correspond exactement à celui introduit dans la Figure 6.2.

6.5 *AMISS* : Algorithme de ModélIsation de Schémas Semi-structurés

Nous avons déjà vu le cas de la modélisation d'une association, nous allons maintenant la généraliser pour modéliser les alternatives de structuration d'un modèle UML. Pour modéliser un ensemble d'alternatives semi-structurées basées sur un modèle UML, nous proposons de construire un modèle de caractéristiques en utilisant notre algorithme nommé *AMISS* (*A*lgorithme de *M*odél*I*sation de *S*chémas *S*emi-structurés). Cette section est dédiée à la description détaillée de cet algorithme qui sera accompagnée d'un exemple.

AMISS est basé sur la fusion des modèles de caractéristiques correspondant à chaque association du modèle UML, introduit dans la section précédente. La fusion des modèles applique plusieurs opérations et analyse les contraintes pour garantir un modèle de caractéristiques valide. Cet algorithme, repose sur les définitions d'un modèle UML et de notre modèle de caractéristiques correspondant à une association UML.

Soit mU une paire (E,R) où :

- $E = \{e_1,...,e_n\}$ correspond à l'ensemble des classes
- $R = \{r_1,...,r_n\}$ correspond à l'ensemble des associations
- $R(e_i) = \{r_1,...,r_n\}$ est l'ensemble des associations de la classe e_i
- $E(r_n) = \{e_1,e_2\}$ correspond aux classes de chaque extrémité de l'association r_n. $e_1 \neq e_2$
- $A(e_i) = \{a_1,...,a_n\}$ désigne l'ensemble des attributs de la classe $e_i \in E$.
  Chaque $a_i \in A$ est défini par la paire $a_i.name$:$a_i.type$
- $r_n.name$ est le nom de l'association,
- $r_n.ce_i$ correspond à la cardinalité de l'association r par rapport la classe e_i,
- $r_n.role_i$ désigne le rôle de l'association r_i par rapport la classe e_i
- $ER \subseteq E$ désigne le sous-ensemble des classes avec associations

Rappelons que te_i désigne le type d'une classe e_i en considérant ses attributs $A(e_i)$ et un identifiant par défaut a_{id}

Soit fms un modèle de caractéristiques destiné à modéliser la variabilité des alternatives semi-structurées orientées documents basées sur un modèle UML :

- fms_r correspond à un modèle de caractéristiques destiné à modéliser la variabilité des alternatives pour une association UML.
- fms_{full} correspond à un modèle de caractéristiques destiné à modéliser la variabilité des alternatives pour une modèle UML.

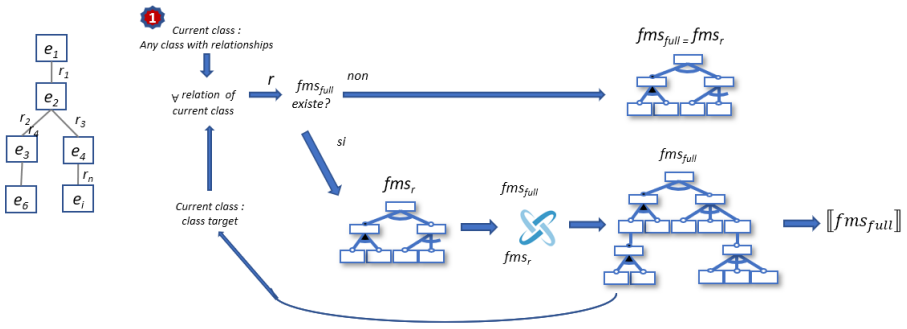


FIGURE 6.15: Structure générale d'*AMISS*

6.5.1 Aperçu général de la modélisation de schémas

L'algorithme *AMISS* est en charge de la création d'un modèle de caractéristiques qui exprime un ensemble d'alternatives de schémas semi-structurés à partir d'un modèle UML. Cette création est décrite par l'Algorithme 2. La Figure 6.15 donne un aperçu général des éléments d'AMISS.

AMISS déclenche la création des collections à partir de chaque classe du modèle UML en considérant les chemins de ses associations et les possibilités d'imbriquer ou référencer les types des classes incluses dans ces chemins. Il s'agit d'une fusion incrémentale des modèles de caractéristiques de chaque association considérée. Cette fusion contrôle l'ensemble des contraintes afin de garantir la complétude et à éviter les boucles entre les références et l'imbrication. Les étapes générales de l'algorithme sont les suivantes :

— Il commence en choisissant n'importe quelle classe de l'ensemble *ER* du modèle UML (cf. Ligne 2). Cette classe est celle qui déclenche le parcours récursif entre les classes et leurs associations dans le modèle UML pour obtenir le modèle de caractéristiques final fms_{full} (cf. Ligne 3). Les classes sans associations sont analysées à la fin (cf. Ligne 4).

— Concernant la fusion des associations, pour chaque association appartenant à la classe courante, un modèle de caractéristiques fms_r est créé. S'il s'agit de la première association analysée, ce modèle correspondra au modèle des caractéristiques final fms_{full}, sinon il fusionnera avec celui-ci. (cf. Lignes 6-11)

— La fusion combine les modèles fms_r et fms_{full} via la classe courante, et considère de nouvelles options d'imbrication. Les contraintes fournies par chaque modèle sont considérées ainsi que des contraintes supplémentaires pour garantir la complétude et contrôler l'impact des nouvelles alternatives sur celles déjà analysées. (cf. Lignes 12-28)

— Une fois la fusion effectuée, *AMISS* passe la main à l'autre classe extrême de l'association courant, afin que les associations appartenant à celle-là soient maintenant analysées (cf. Lignes 30). Le processus est répété jusqu'à que toutes les associations appartenant aux classes qui déclenchent des collections soient analysées (cf. Ligne 6).

Algorithme 2 AMISS : Algorithme de modélisation de schémas semi-structurés

Input: Modèle UML m, classes avec associations binaires ER, les duplications des extensions de types $CSE = cse(e_k)|e_k \in E$, les duplications des extensions des associations r_{rol} $CSR = csr(r_n, e_k)|r_n \in R \bigwedge e_k \in E(r_n)$

Output: Modèle de caractéristiques fms_{full}

$fms_{full} \leftarrow null$
$e_i \leftarrow$ random(ER)
$fms_{full} \leftarrow$ fusionAssociations(fms_{full}, e_i)
$fms_{full} \leftarrow$ treatIsolatedTypes$(fms_{full}, E - ER)$

function FUSIONASSOCIATIONS(fms_{full}, e_i) : fms_{full}
  foreach $r_n \in R(e_i)$ **do**
    $e_j \leftarrow$ getTarget(r_n, e_i)
    $fms_r \leftarrow$ generateFMassociation(e_i, e_j, r_n)
    if fms_{full} *is null* **then**
      $fms_{full} \leftarrow fms_r$
    else
      ▷ Source class - Common classe
      $fms_{branch} \leftarrow$ extractExtension(fms_r, e_i, r_n) ▷ Branche r_role_j et contraintes
      $fms_{full} \leftarrow$ insertBranchCommonClass$(fms_{full}, fms_{branch}, e_i)$
      foreach $f_t \in F_{fms_{full}} \wedge f_m = \bigotimes emb^{ce_i} te_i$ **do**
        if f_t *is leaf* **then**
          $fms_{full} \leftarrow$ insertFeatureExtension$(fms_{full}, f_t, e_i, cse(e_i))$
        end if
        $fms_{branchV} \leftarrow$ createBranchVersion$(fms_{branch}, e_i, e_j, csr(r_n, e_j))$
        $fms_{full} \leftarrow$ embedBranch$(fms_{full}, fms_{branchV}, f_t.child)$ ▷ Dans la extension
      end foreach
      ▷ Target class
      $fms_{branchCol} \leftarrow$ extractSubFM(fms_r, e_j, r_n)
      $fms_{full} \leftarrow$ insertBranchNewClass$(fms_{full}, fms_{branchCol}, e_j)$
      $fms_{full} \leftarrow$ insertFeatureExtension$(fms_{full}, f_t, e_i, cse(e_j))$
      foreach fms_{branch} $filsOf$ $(te_i\_l_0\_ext) \in fms_{full} \bigwedge fms_{branch}.root \neq r\_role_j$ **do**
        $fms_{branchV} \leftarrow$ createBranchVersion$(fms_{branch}, e_i, e_j, csr(r_n, e_i))$
        $fms_{full} \leftarrow$ embedBranch$(fms_{full}, fms_{branchV}, e_j, cse_t)$
      end foreach
      ▷ *collection-lien*
      $fms_{full} \leftarrow$ insertBranchRelation$(fms_{full}, fms_r, r_n, e_i, e_j)$
    end if
    $fms_{full} \leftarrow$ fusionAssociations(fms_{full}, e_j)
  end foreach
  return fms_{full}
end function

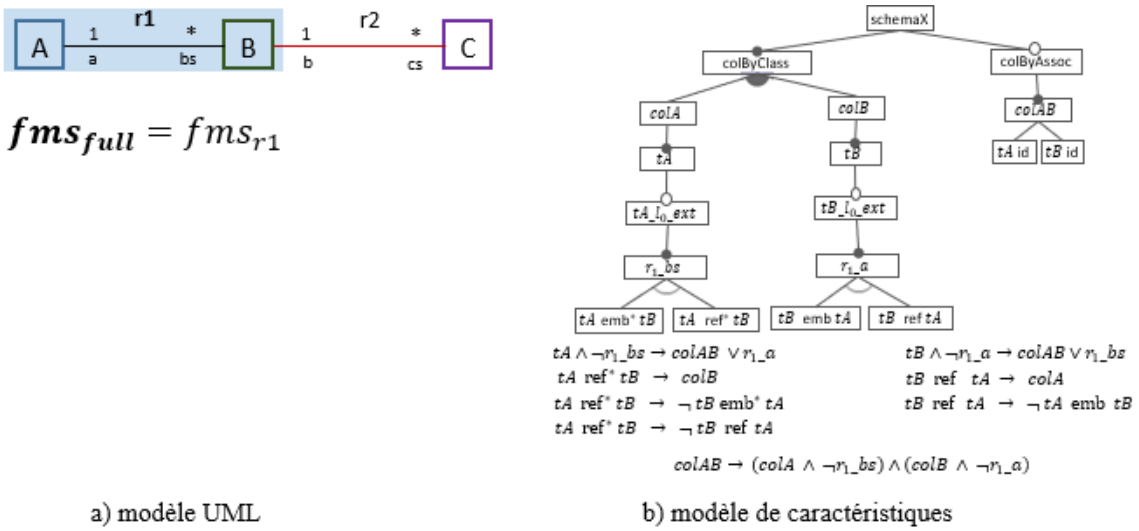


FIGURE 6.16: Exemple *AMISS* : création de fms_r et fms_{full} pour une première association r1

L'ensemble des configurations valides de ce modèle de caractéristiques fms, représente un ensemble d'alternatives de structuration valides pour le modèle UML fourni.

6.5.2 Détail des étapes

Pour créer le modèle des caractéristiques correspondant aux alternatives semi-structurées d'un modèle UML, *AMISS* commence par choisir au hasard une classe source e_i avec des associations. Ensuite, il crée un modèle fms_r pour l'une des associations r_n appartenant à cette classe source e_i. Ce modèle correspondra dans une première itération au modèle final $fmsfull$ (cf. Algorithme 2 ligne 2, 3, 8 et 10).

La Figure 6.16 introduit le processus de modélisation des alternatives semi-structurées applicable au le modèle UML de la Figure 6.3. La classe Agency, nommée A dans la suite, déclenche l'exécution d'*AMISS*. Un modèle de caractéristiques est créé en considérant l'association r_1 avec la classe destination BusinessLine (Classe B dans la suite). Ce modèle, nommé fms_{r1}, est pour l'instant le modèle final fms_{full}.

Une fois créé le modèle fms_r pour l'association r_n appartenant à la classe source e_i, les autres associations de e_i sont mises en attente afin d'analyser la classe extrême de r_n, la classe e_j. Maintenant, cette classe e_j est la nouvelle classe source e_i et l'une de ses associations est modélisée avec le modèle fms_r correspondant. Le nouveau modèle est alors fusionné avec le modèle final $fmsfull$.

Dans notre exemple, la classe B, qui était considérée comme la classe destination pour l'association r_1, Figure 6.16, est ensuite considérée comme la classe source pour l'association r_2. Un modèle fms_{r2} est alors créé et fusionné avec le modèle fms_{full} existant.

La fusion entre un modèle fms_{r_n}, tel que $r_n) \in R(e_i)$ et le modèle final fms_{full} relie d'abord les branches correspondant à la collection du type source te_i et à l'extension de l'association r_n(cf. Algorithme 2 ligne 12 à 20). Ensuite, il relie la branche correspondant à la collection de type te_j (ligne 21 à 27). Finalement, les branches correspondant aux *collections-liens* sont liées (ligne 28). Lorsqu'une branche collection est liée, les possibilités d'imbrication du type concerné par rapport aux alternatives existantes sont prises en compte via une stratégie de gestion des versions (lignes 18 et 25). A ce niveau, nous insérons des duplicates des branches (types et associations) en préservant leur numéro d'ordre de succession. Le processus d'analyse ajoute l'extension pour chaque branche collection.

La fusion des branches correspondant au type source te_i, extrait d'abord du modèle fms_r, la branche fille de l'extension tei_lo_ext, cf. Algorithme 2 ligne 12 et Algorithme 3. La branche r_{nrolej} ainsi que les contraintes correspondant le type e_i et l'association r_n sont aussi extraites, à savoir les contraintes d'isolement $\mathscr{D}^I$, d'existence $\mathscr{D}^E$, de boucles imbriquées $\mathscr{D}^{BI}$ et de boucles des références $\mathscr{D}^{BR}$.

Ensuite, cette branche est ajoutée, cf. Algorithme 4, à la caractéristique d'extension tei_lo_ext du modèle final fms_{full} afin de créer un groupe OR avec les branches existantes. Les contraintes d'existence ($\mathscr{D}^E$) et les boucles ($\mathscr{D}^{BR}$,$\mathscr{D}^{BI}$) sont ajoutées directement à l'ensemble des contraintes $\mathscr{D}(fms_{full})$. Les contraintes d'isolement ($\mathscr{D}^I$) sont fusionnées en faisant une union des conséquences pour les restrictions avec un antécédent commun.

Algorithme 3 extractExtension (fms_r, e, r) : fms_{branch})

1: $fms_{branch} \leftarrow getBranch(fms_r, r\_role)$ ▷ Extraire branche r_role
2: $\mathscr{D}^E(fms_{branch}) \leftarrow d^E(e) \in \mathscr{D}^E(fms_r)$ ▷ Extraire contraintes
3: $\mathscr{D}^{BR}(fms_{branch}) \leftarrow d^{BR}_{(}r) \in \mathscr{D}^{BR}(fms_r)$
4: $\mathscr{D}^{BI}(fms_{branch}) \leftarrow d^{BI}(e) \in \mathscr{D}^{BI}(fms_r)$
5: $\mathscr{D}^I(fms_{branch}) \leftarrow d^I(e) \in \mathscr{D}^I(fms_r)$

Algorithme 4 insertBranchCommonClass($fms_{full}, fms_{branch}, e_i$)

1: $fms_{full} \leftarrow insert(fms_{full}, te_i\_l0\_ext, fms_{branch}, OR)$ ▷ Ajouter sous-arbre
2: $\mathscr{D}(fms_{full}) \leftarrow insertCsts(\mathscr{D}(fms_{full}), \mathscr{D}^E(fms_{branch}))$
3: $\mathscr{D}(fms_{full}) \leftarrow insertCsts(\mathscr{D}(fms_{full}), \mathscr{D}^{BR}(fms_{branch}))$
4: $\mathscr{D}(fms_{full}) \leftarrow insertCsts(\mathscr{D}(fms_{full}), \mathscr{D}^{BI}(fms_{branch}))$
5: $\mathscr{D}(fms_{full}) \leftarrow insertCsts(\mathscr{D}(fms_{full}), \mathscr{D}^I(fms_{branch}), UNION)$

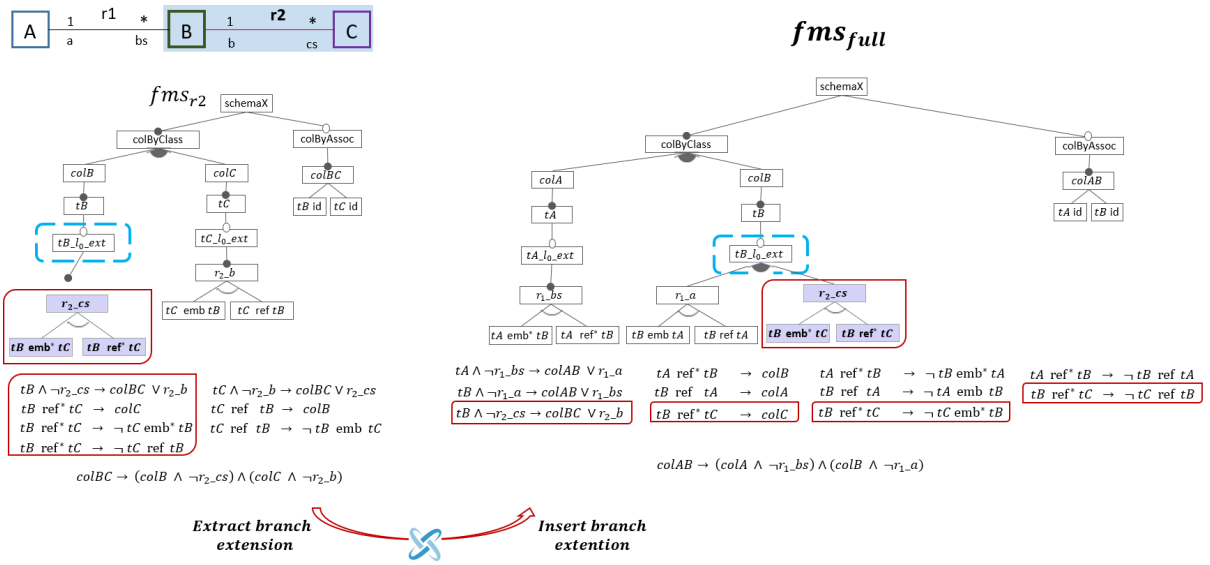


FIGURE 6.17: Exemple *AMISS* : création de fms_{r2} et fusion de la branche commun avec fms_{full}

Dans notre exemple, la Figure 6.17 introduit à gauche le modèle fms_r pour l'association r_2 en considérant la classe B comme la classe source. La branche r_{2cs}, fille de la caractéristique tB_lo_ext, ainsi que leurs contraintes respectives sont extraites. Cette branche est ajoutée à l'extension tB_lo_ext du modèle final fms_{full}. Les contraintes sont aussi ajoutées.

Définition 6.11. Contrainte d'isolement imbriquée

Une contrainte d'isolement imbriquée $d^I I(e_i)$ garantit dans un modèle de caractéristiques tel que fms, que si un type te_i, imbriqué par le type te_k, n'a pas d'extension pour son association r_n vers te_j, alors des caractéristiques r_n garantissent sa présence dans d'autres collections.

En considérant $\mathscr{D}^{II}(fms) = \{d^I I(te_i) | \, f \in fms, \, f \doteq r_n role_j, \, (e_i, e_j) \in E(r_n)\}$

comme l'ensemble des contraintes d'isolement imbriquées appartenant au modèle fms, une contrainte est définie comme une paire de propositions logiquement équivalentes telles que :

$$d^I I(te_i) = (te_i \, emb^{r_n.ce_i} \, te_j \wedge \neg r_n\_ce_j\_version \implies r_n\_role_j \vee r_n\_role_i)$$

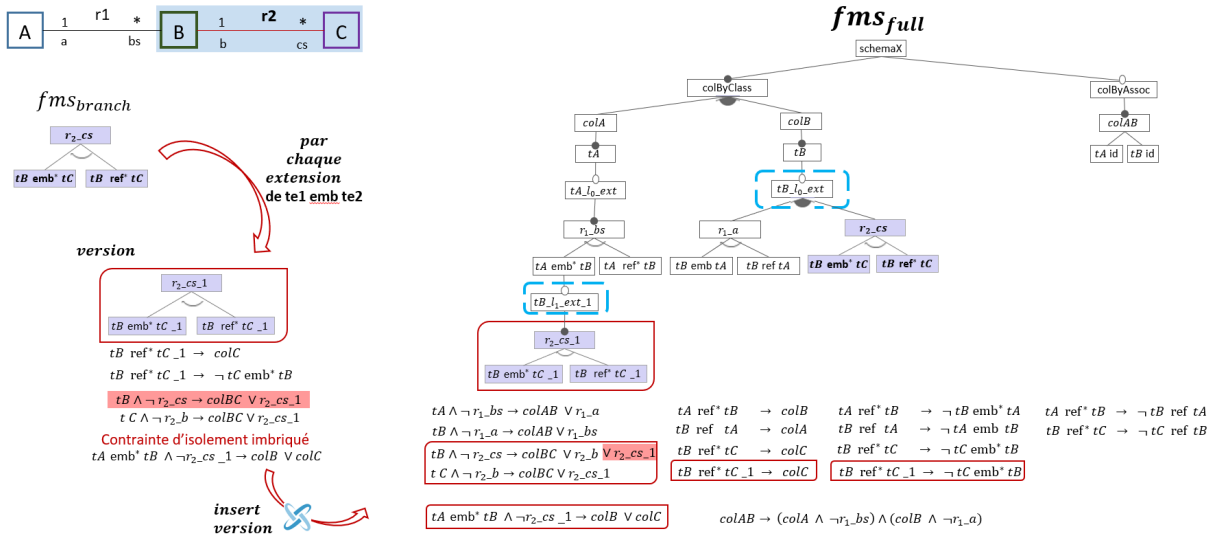


FIGURE 6.18: Exemple *AMISS* : fusion de la branche commune, versions et imbrications

Lorsqu'une nouvelle branche est ajoutée à l'extension de te_i dans le modèle fms_{full}, alors une version de cette branche est ajoutée à l'extension de chaque te_i imbriqué, cf. Algorithme 6. Si l'extension imbriquée a des branches, la branche version est ajoutée en mode OR. Si l'extension imbriquée n'existe pas dans le modèle final fms_{full}, cette caractéristique est donc ajoutée avec sa version correspondante en mode obligatoire, cf. Algorithme 7.

La version de la branche à insérer, cf. Algorithme 5, est créée en ajoutant le consécutif de duplication correspondant à toutes les caractéristiques de la branche. Ce consécutif correspond à la succession de copies faites de la branche. Des contraintes correspondant à la version de la branche sont également créées. Celles-ci concernent les contraintes d'existence $\mathscr{D}^E$, de boucles imbriquées $\mathscr{D}^{BI}$, d'isolement $\mathscr{D}^I$ et d'isolement imbriquées $\mathscr{D}^{II}$. Toutes les branches ainsi que les contraintes sont ajoutées au modèle.

La Figure 6.18 introduit l'insertion des versions de la branche r_{2cs} correspondant à l'extension du type tB dans le modèle fms_{full}. Une seule caractéristique imbrique le type tB, $tA\ emb^*\ tB$, alors une seule version de r_{2cs} est créée. Notons que cette version ajoute à toutes les caractéristiques le consécutif 1 parce que seulement une version de cette branche association a été conçue. Leurs contraintes respectives sont également créées. Une fois la version créée, elle est ajoutée à l'extension de $tA\ emb^*\ tB$, car celle-ci n'a pas été étendue jusqu'ici, sa caractéristique d'extension est ajoutée, et ensuite la version.

Algorithme 5 createBranchVersion ($fms_{full}, fms_{branch}, e_i, csr$)

1: $fms_{branchV} \leftarrow fms_{branch}$ ▷ Copier la branche et versionner
2: **foreach** $f \in fms_{branchV}$ **do**
3: $f.name \leftarrow addSufix(f.name, csr)$
4: **end foreach**
5: $\mathscr{D}(fms_{branchV}) \leftarrow createCsts(fms_{branchV}, \mathscr{D}^E)$

Algorithme 6 embedBranch($fms_{full}, fms_{branchV}, fExtension$)

1: **if** $fExtention\ is\ leaf$ **then**
2: $fms_{full} \leftarrow insertFMfeatures(fms_{full}, fms_{branchV}, fExtension, Mandatory)$
3: **else**
4: $fms_{full} \leftarrow insertFMfeatures(fms_{full}, fms_{branchV}, fExtension, OR)$
5: **end if**

Algorithme 7 insertFeatureExtension($fms_{full}, f, niveau, e_i, cse$)

1: $nameFeature \leftarrow t[ei]\_l[niveau]\_ext\_[cse]$
2: $fms_{full} \leftarrow addFeature(fms_{full}, f, nameFeature, Optional)$ ▷ addFeature(fm,parent,f,type)

La fusion des branches correspondant la destination e_j de l'association courante, (cf. Algorithme 2 ligne 21 à 27), consiste d'abord à extraire du modèle fms_r la branche correspondant à la collection e_j, y compris ses contraintes, et ensuite à l'insérer dans le groupe OR de la caractéristique colByClass du modèle fms_{full}, cf. Algorithme 2 lignes 21 et 22.

Ensuite, toutes les branches de l'extension *tei_lo_ext* dans fms_{full}, à l'exclusion de celle de l'association actuelle, sont versionnées et ajoutées à une nouvelle extension imbriquée de te_j.

Dans notre exemple, la Figure 6.19 introduit la fusion de la branche correspondante la collection de type *tC*. Notons que la branche complète est extraite avec ses contraintes afin de joindre le groupe OR de la caractéristique *colByClass* et l'ensemble des contraintes du modèle fms_{full}. Notons dans la figure 6.20, que l'imbrication de branches appartenant à l'extension du type *tB* imbrique dans son extension la collection du type *tC*

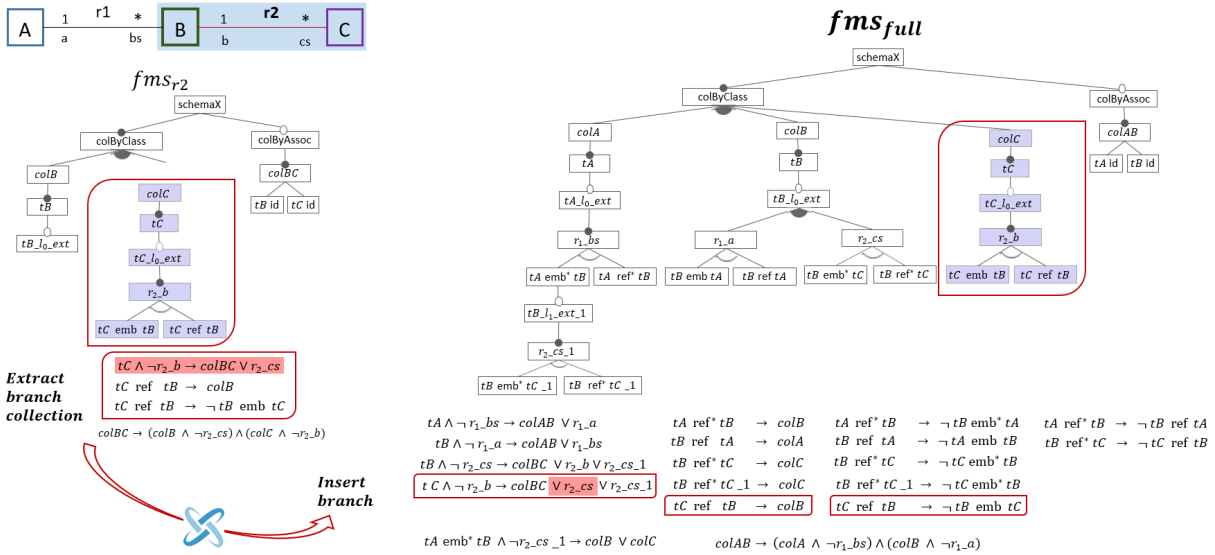


FIGURE 6.19: Exemple *AMISS* : fusion d'une nouvelle branche

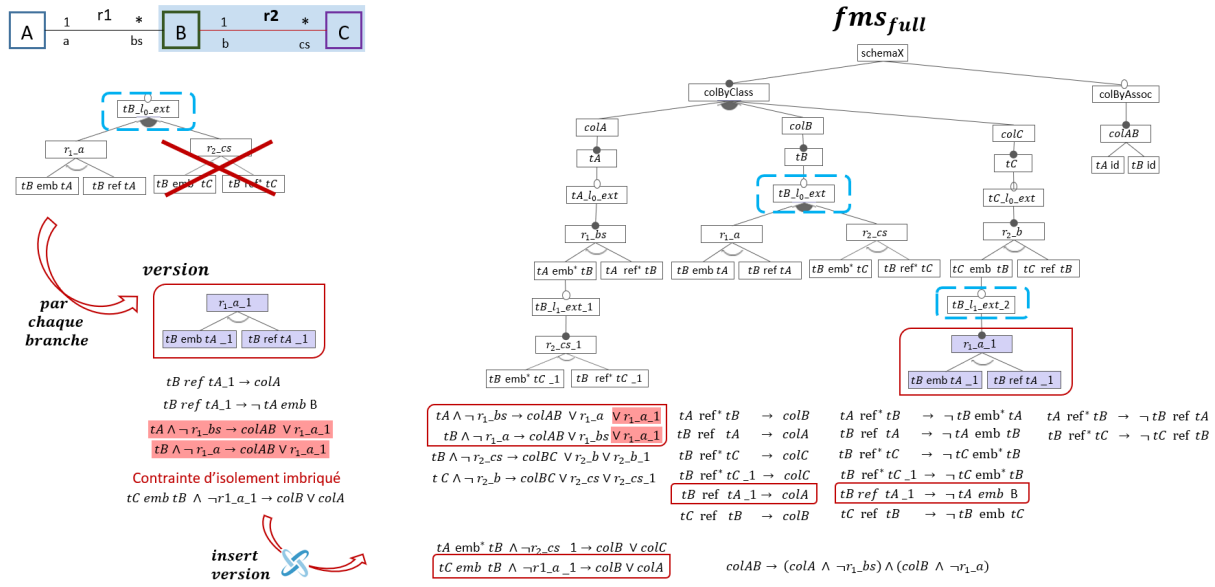


FIGURE 6.20: Exemple *AMISS* : fusion d'une nouvelle branche et imbrication

Concernant la fusion des branches correspondant les *collections-liens*, la branche $cole_i e_j$ est extraite du modèle fms_r, y compris sa contrainte, et elle est ajoutée à la caractéristique *colByAssoc* du modèle fms_{full} en mode OR, cf. Algorithme 8.

Une fois la fusion terminée, les associations de la classe destination e_j sont analysés à nouveau

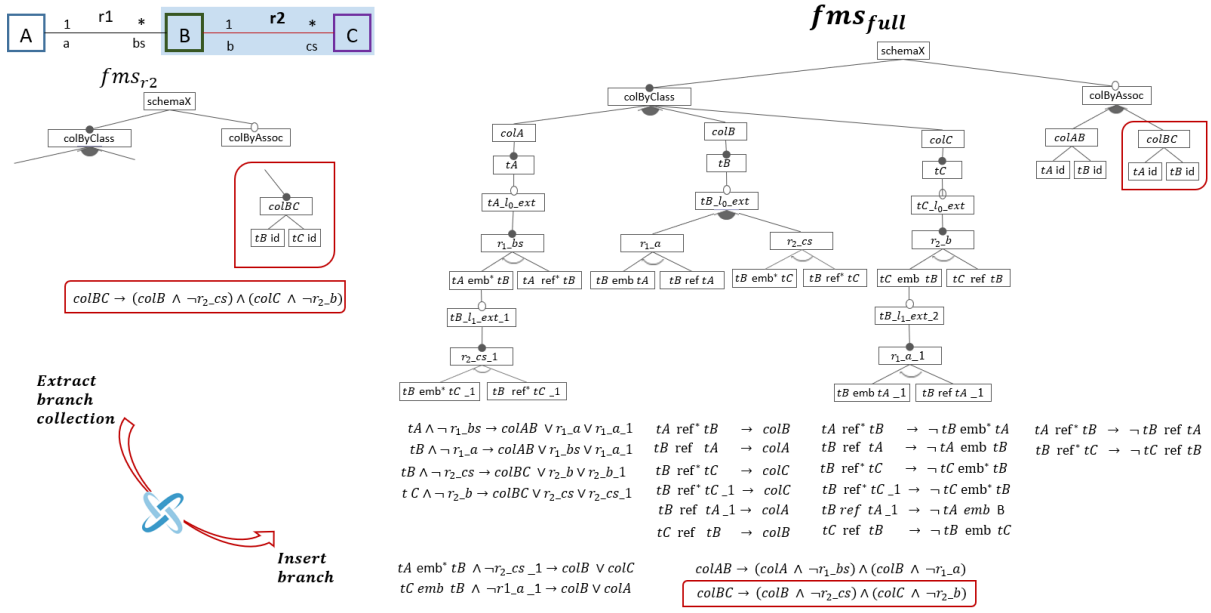


FIGURE 6.21: Exemple *Amiss* : fusion de la branche vue

avant de celles appartenant à la classe source e_j, cela garantit une fusion incrémentale des associations présentes dans un chemin.

Dans notre exemple, la Figure 6.21 introduit la fusion de la branche correspondant à la *collection-lien colBC* avec la collection *lien colAB* en mode OR. Notons qu'avec cette fusion la modélisation du modèle fms_{full} prend fin parce que nous n'avons plus d'associations ou de classes à analyser, ni des classes isolées.

Dans le cas de classes isolées dans le modèle UML, cf. Algorithme 2 ligne 4, une caractéristique collection avec son type et sans extension est ajoutée au groupe *colByClass*. Un contrainte est aussi considérée pour garantir toujours sa sélection.

Algorithme 8 insertBranchRelation ($fms_{full}, fms_r, r, e_i, e_j$)

1: $fms_{branch} \leftarrow getBranch(fms_r, cole_ie_j)$
2: $\mathscr{D}^L(fms_{branch}) \leftarrow d^L(r) \in \mathscr{D}^L(fms_r)$
3: $fms_{full} \leftarrow insert(fms_{full}, colByAsso, fms_{branch}, OR)$
4: $\mathscr{D}(fms_{full}) \leftarrow insertCsts(\mathscr{D}(fms_{full}), \mathscr{D}^L(fms_{branch}))$

6.6 Dérivation d'une alternative de structuration

Afin de pouvoir analyser et évaluer les alternatives de structuration correspondant aux configurations $\|fms\|$ fournies par le modèle de caractéristiques fms, nous proposons deux dérivations de chaque configuration afin d'obtenir : une représentation arborescente nommée AJTree, avec laquelle il sera possible d'évaluer le schéma automatiquement avec nos métriques structurelles, et une représentation AJSchéma avec laquelle le schéma peut être présenté de manière textuelle à l'utilisateur. Dans cette section, nous présentons les algorithmes *ADJusT* et *ADApt*. L'algorithme *ADJusT* interprète l'ensemble de caractéristiques d'une configuration et construit l'AJTree respectif. L'algorithme *ADApt* construit l'AJSchéma correspondant. Nous créons des schémas avec des collections dont les documents sont homogènes.

6.6.1 ADJusT : Algorithme de dérivation d'une représentation AJTree

Afin de faciliter l'évaluation automatique d'une alternative de structuration fournie par une configuration $C_i \in \|fms\|$, nous proposons un processus de dérivation axé sur la traduction de C_i sous forme d'une représentation arborescente nommée *AJTree*. La dérivation de toutes les configurations appartenant fms donne lieu à un ensemble des représentations AJTree correspondant à l'ensemble d'alternatives de structuration basées sur un modèle UML .

$$\mathscr{P}_{AJTree}(fms) = \{AJTree \mid AJTree_i \leftarrow deriverIntoTree \wedge C_i \in \|fms\|\}$$

La représentation arborescente AJTree contient des informations sur les types de données, les niveaux de profondeur et les types imbriqués et référencés. Une telle représentation considère un choix de modélisation semi-structurée basée sur un modèle UML tel que :

- Le nœud `Racine`, ShemaX, a un fils par `Collection` présente dans le schéma. Le type de la collection est représenté par le sous-arbre fils.
- Les nœuds `TypeData`, sous forme *typename*, représentent le premier type inclus dans le type d'une collection.
- Les nœuds de la forme *typename@li* indiquent l'`Extension` du type *typename* au niveau *li* (en commençant par le niveau 0).
  L'information concernant les associations $R(e_i)$ choisies apparaissent comme suit :
- Un nœud `Association` avec le nom (et le rôle utilisé) de chaque association est établi avec un nœud fils, soit *TypeDataREF*, soit *TypeDataEMB* selon le choix.
- Les nœuds *TypeDataEMB* suivent la forme $type_1$ *EMB* $type_2$ en indiquant l'imbrication du $type_2$ dans $type_1$. Ce nœud peut avoir un nœud fils `Extension` lié au type $type_2$.
- Les nœuds *TypeDataREF* suivent la forme $type_1$ *REF* $type_2$ en indiquant le référencement du $type_2$ existant dans une autre collection. Ces nœuds n'ont pas de fils.

— Les tableaux, notés [], peuvent être utilisés pour les associations 1-à-plusieurs.

Algorithme 9 ADJusT : Algorithme de Dérivation d'une configuration $C_i \in \|fm\|$ vers un $AJtree_i$

```
1: function DERIVERINTOAJTREE(C_i ∈ ||fms||) : AJTree
2:     AJTree ← getInstanceTree(fms, C_i)                ▷ Prendre fms selon C_i
3:     AJTree ← removeNode(AJTree, colByClass)
4:     AJTree.racine ← adjustRootName(fm_C.racineName, sequence i of C_i)
5:     foreach noued n ∈ AJTree do
6:         typeNode ← getTypeFeature(f)
7:         n.name ← adjustName(typeNode, n.name)
8:     end foreach
9: end function
```

La dérivation d'une configuration $C_i \in fms$ vers cette représentation AJTree est effectuée par l'algorithme *ADJusT* (cf. algorithme 9). Celui-ci réalise une traduction comme suit :

— Chaque caractéristique de la configuration C_i correspond à un nœud dans l'AJTree en considérant les mêmes liens.

— La caractéristique *colByClass* est supprimée de Ci et ses liens seront assignés à la racine de l'AJTree

— Un réglage de noms est donc mis en place en fonction du type de caractéristique et de nœud, comme suit :

- La `Racine` ajuste son nom en changeant x par un numéro
- Les caractéristiques correspondant aux nœuds `Collection` ajustent leurs noms aux pluriels.
- Les caractéristiques correspondant aux nœuds `TypeData` conservent leur noms.
- Les caractéristiques correspondant aux nœuds `Extension` conservent le nom du type à étendre suivi du symbole @ et le niveau.
- Les caractéristiques correspondant aux nœuds`Association` enlèvent de leur nom le suffixe du versionnage.
- Les caractéristiques `TypeDataEMB` et `TypeDataREF` enlèvent de leur nom le suffixe du versionnage. Les symboles correspondant à la cardinalité plusieurs sont ajustés.

La Figure 6.22 introduit un exemple de dérivation d'une configuration $C_i \in \|fms\|$ vers un AJTree correspondant à une alternative structurante créée a partir du modèle *uml* de la Figure

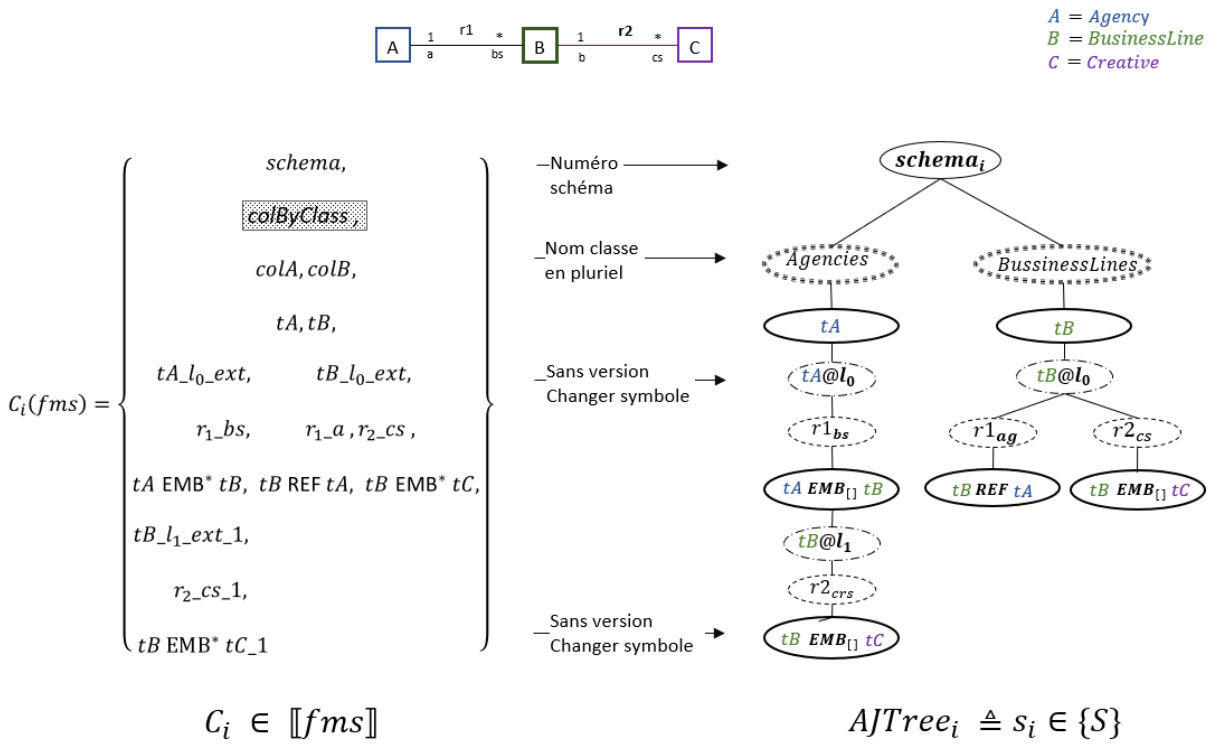


FIGURE 6.22: Exemple de dérivation d'une configuration de fms vers AJTree

6.3. À gauche, les caractéristiques de la configuration C_i correspondent à une alternative de structuration composée par deux collections *colA* et *colB*, déclenchées par les classes `Agencie` (A) et `BusinessLine` (B). L'information de la classe `Creatives` (C) n'est pas considérée comme collection. Le type de la collection *colA* est composé du type *tA* étendu par l'association r_1 en imbriquant le type *tB* qui à son tour est étendu par l'association r_2 en imbriquant le type *tC*. Le type de la collection *colA* est composé du type *tB* étendu deux fois au même niveau par l'association r_1 en référençant le type *tB* et l'association r_2 en imbriquant le type *tC*.

À droite, la représentation AJTree préserve la même distribution arborescente de la configuration C_i en supprimant la caractéristique *colByClass*. Les noms de la racine et de la collection sont ajustés, tous les suffixes de version et d'extension sont supprimés et les symboles ajustés. Le nœud *tBusiness* $EMB_{[]}$ *tCreative* ne contient pas le versionnage de la caractéristique d'origine.

6.6.2 ADApt : Algorithme de dérivation d'une représentation AJSchéma

Afin d'expliciter les types des structures de données d'une alternative de structuration fournie par une configuration $C_i \in \|fms\|$, nous proposons une dérivation axée sur la traduction de l'AJTree correspondant à la configuration C_i sous forme d'une représentation AJSchéma. La dérivation de toutes les configurations appartenant à fms sous forme d'AJTree donne lieu à un ensemble des représentations *AJSchéma*.

$$\mathscr{P}_{AJSchema}(fms) = \{AJSchema \mid AJSchema_i \leftarrow deriverIntoAJSchema \wedge AJTree_i \leftarrow C_i \in \|fms\|\}$$

Cette dérivation est réalisée par l'algorithme *ADApt* (cf. Algorithme 10). Il parcourt la structure arborescente AJTree et établit l'équivalence avec la grammaire $G_{AJSchema}$ introduite dans le chapitre précèdent, section 5.3. Quand à l'Algorithme 11, il crée la représentation textuelle correspondante à un AJSchéma.

Algorithme 10 ADApt : Dérivation d'une configuration $C_i \in \|fms\|$ vers un $AJSchema_i$

```
1:  function DERIVERINTOAJSCHEMA(fm_C) : AJSchema
2:      AJSchema ← addSchemaName(AJSchema, fmc.rootName)

3:      foreach node nodeCol fils of colByClass do
4:          colName ← nodeCol.name
5:          ajsCol ← colName + ":" + addEnter() + "{" + addEnter()
6:          nodeDataType ← nodeCol.child()
7:          dataType ← nodeDataType.name
8:          atts ← getAttributesUML(dataType)

9:          foreach att of atts do
10:             ajsCol ← ajsCol + addTab() + att.name + ":" + att.type + "," + addEnter()
11:         end foreach

12:         if nodeDataType is not leaf then
13:             ajsCol ← ajsCol + createAJSchemaSub(nodeDataType, 1)
14:         else
15:             ajsCol ← adjustEndLine(ajsCol)
16:         end if

17:         ajsCol ← ajsCol + addEnter() + "}"
18:         AJSchema ← AJSchema + ajsCol + addEnter(2)
19:     end foreach
20: end function
```

Algorithme 11 ADApt - createAJSchemaSub

```
1: function CREATEAJSCHEMASUB(TreenodeDataType, Integertabs) : String
2:     StringassocName, rol, matType, dataType, ajsExt ← null
3:     NouednodeMat ← null
4:     BooleancardMany ← false
5:     List < Attribute > atts ← null
6:     foreach nodeAssoc grandChild of nodeDataType do
7:         assocName ← nodeAssoc.name
8:         rol ← nodeAssoc.rol
9:         nodeMat ← getFils(nodeAssoc)
10:        matType ← getMatType(nodeMat)
11:        cardMany ← isCardMany(nodeMat)
12:        dataType ← getDataTypeMat(nodeMat)
13:        atts ← getAttributesUML(dataType)
14:        if matType is EMB then
15:            ext ← insertTab(ext, tabs)
16:            ext ← rol + insertEnter() + insertTab(ext, tabs)
17:            if matMany is true then ext ← ext + "[" end if
18:            ext ← "{" + insertEnter()
19:            foreach att of atts do
20:                ext ← applyTab(ext, tabs + 1)
21:                ext ← att + tab() + att.name + " : " + att.type + ", enter()"
22:            end foreach
23:        else
24:            ext ← insertTab(ext, tabs)
25:            ext ← rol + " : "
26:            if matMany is true then ext ← ext + "[" end if
27:            ext ← "Integer@" + dataType
28:        end if
29:        if nodeMat is not Leaf then
30:            ext+ = createAJSchemaSub(nodeDataType.fils, tabs + 1)
31:        else
32:            ajsCol ← adjustEndLine(ajsCol, matType)
33:        end if
34:        ext ← insertTab(ext, tabs)
35:        if matType is EMB then ext+ = "}" end if
36:        if matMany is true then ext+ = "]" end if
37:    end foreach
38:    return ext
39: end function
```

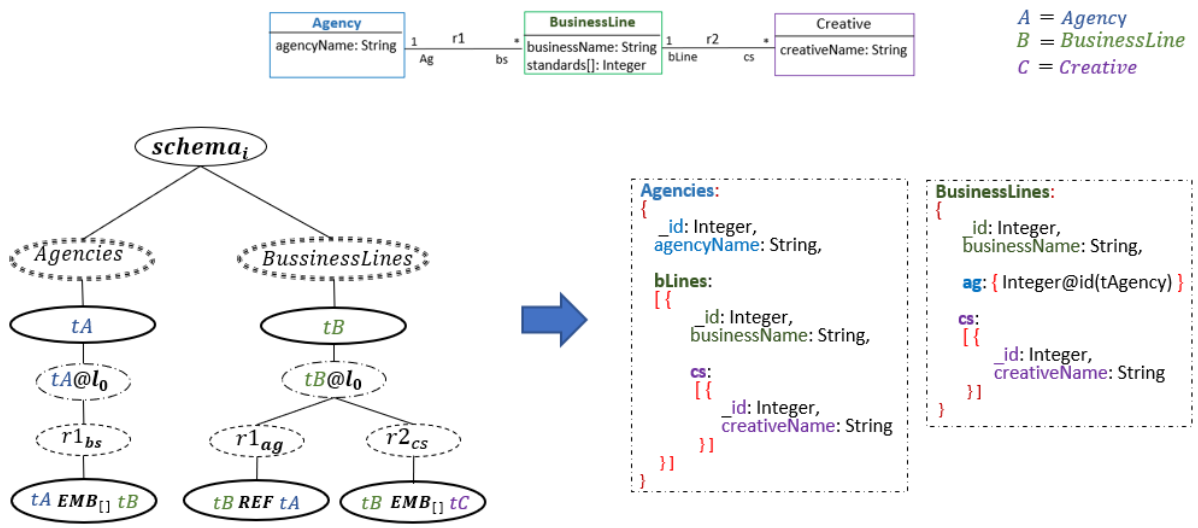


FIGURE 6.23: Exemple de dérivation d'un AJTree vers un AJSchéma

La Figure 6.23 introduit un exemple de dérivation de l'AJTree correspondant à la configuration $C_i \in \|fms\|$ de la Figure 6.22. Nous notons dans la représentation AJSchéma, les deux collections `Agencies` et `BusinessLines` avec ses types détaillés.

6.7 Conclusion

La flexibilité de la structuration des données offerte par les systèmes NoSQL orientés documents, comme MongoDB, permet des centaines d'options de modélisation, chacune avec ses avantages et ses inconvénients en fonction de plusieurs facteurs. Cependant, il est difficile de choisir les structures quand il de nombreuses options, et que les structures ne sont pas explicites à cause de l'absence de schéma dans le système. De plus, il n'y a pas de critères objectifs qui permettent d'évaluer et de comparer de telles structures.

Nous avons exploré dans ce chapitre la modélisation de plusieurs alternatives semi-structurées basées sur un modèle UML en contrôlant les variantes et l'explosion de combinaisons qui peuvent émerger. Pour cela, nous utilisons les modèles de caractéristiques, une stratégie de génie logiciel pour modéliser les variations entre produits.

Nous avons proposé la modélisation des alternatives semi-structurées pour une association UML

et nous avons également proposé l'algorithme *AMISS* permettant de générer automatiquement un ensemble d'alternatives de structuration d'un modèle UML. L'algorithme *ADApt* donne la possibilité de visualiser rapidement les alternatives et facilite l'analyse.

L'algorithme *AMISS* ne génère pas toutes les possibilités car nous ne considérons pas les tuples partiels ou les combinaisons de tuples partiels. Nous avons restreint la génération en définissant des contraintes qui nous permettent d'obtenir des alternatives pertinentes prenant en compte différentes options d'imbrication et de duplication de données.

L'algorithme *AMISS* peut facilement être étendu en ajoutant d'autres restrictions fournies par l'utilisateur qui permettent de restreindre les alternatives, telles que la réduction du nombre de niveaux d'imbrication ou la restriction de la duplication des données. Cette possibilité rend la proposition plus pratique, la génération et l'analyse plus efficaces.

Nous avons développé les algorithmes *ADJust* et *ADApt* qu'interprètent chaque alternative de structuration afin de générer une structure arborescente et le AJSchéma correspondant qui permet de donner une représentation adoptée à la lecture pour l'utilisateur. La structure arborescente permet d'analyser et d'évaluer automatiquement des métriques sur la structure. Dans le chapitre suivant nous proposons un ensemble de métriques structurelles pour refléter des caractéristiques importantes dans le choix de structuration des bases de documents.

"Nous ne pouvons pas choisir les circonstances extérieures, mais nous pouvons toujours choisir la façon dont nous répondons à celles-ci."

Epictète

7

Métriques structurelles

.

7.1 Introduction

La manière dont sont structurées les données a un fort impact sur la taille de la base, les performances des requêtes et la lisibilité du code des requêtes, ce qui influence la maintenabilité et l'utilisabilité de la base ainsi que de ses applications. Cela a été constaté de manière expérimentale dans le chapitre 4 où plusieurs patrons de comportement ont été mis en évidence. Notamment, les collections avec des documents imbriqués sont favorables aux requêtes qui suivent l'ordre de l'imbrication. L'accès aux données dans un autre ordre est désavantagé. Les performances des requêtes nécessitant l'accès à des données intégrées à différents niveaux dans la même collection seront pénalisées. La raison est que la complexité des manipulations requises dans ce cas, est proche de celle des jointures de plusieurs collections. En outre, les collections avec des documents imbriqués ont une empreinte mémoire plus importante que la représentation équivalente avec des références.

Dans le choix de la structuration des données, les priorités peuvent s'avérer divergentes, comme le souhait de dupliquer des documents dans plusieurs collections parce qu'ils sont consultés dans des contextes différents mais aussi le souhait de réduire le coût du stockage. Il devient alors intéressant de pouvoir considérer plusieurs structurations candidates pour retenir un seul choix ou plusieurs alternatives parallèles pour un certain temps selon les cas. Pour faciliter l'étude d'alternatives, nous avons proposé le générateur de structures présenté dans le chapitre précédant. Il reste la question de l'absence de critères objectifs d'analyse.

Nous avons analysé un certain nombre de caractéristiques critiques sur les structures afin d'établir des critères qui aident dans le choix d'un schéma. Dans la suite nous proposons des métriques structurelles mesurables sur les schémas de manière à disposer d'éléments objectifs d'appréciation et de comparaison au regard des priorités de l'utilisateur. Ces métriques ne nécessitent pas d'une base peuplée de données et peuvent être utilisées dans une phase de conception et d'analyse. Elles peuvent bien sur être utilisées en association avec des informations statistiques sur les données si elles existent.

Dans les sections 7.3 à 7.7 nous présentons la définition formelle des métriques proposées. Elles seront illustrées sur le cas d'étude de la section 7.2. La section 7.8 présente les conclusions de ce chapitre.

7.2 Cas d'étude

Afin d'illustrer nos métriques structurelles, nous présentons dans cette section un cas d'étude dans un contexte de marketing. Dans un premier temps, nous décrivons le diagramme de classes UML de notre exemple. Ensuite, nous introduisons un AJSchéma correspond à une alternative structurante parmi N possibles a partir de ce modèle UML. Enfin, nous présentons la structure arborescente AJStree qu'illustre la complexité hiérarchique de la structure et sur laquelle est basée le calcul automatique des métriques.

La Figure 7.1 développe un modèle de classes UML dans un contexte de Marketing. Le modèle considère cinq classes et les associations sont 1 .. * avec la sémantique habituelle. La classe `Agency` représente une agence qui a plusieurs propriétaires, représentes par la classe `Owner`. A son tour, une agence peut gérer plusieurs secteurs d'activité, représentés par la classe `BusinessLine`. Chaque secteur d'activité est en charge de plusieurs designers (classe `Creative`) et lié à plusieurs publicités (class `Publicity`). Chaque designer et publicité appartient à un seul secteur d'activité. Un secteur d'activité est géré par une seule agence et un propriétaire ne peut avoir qu'une seule agence. Chaque classe fournit son type pour désigner ses attributs : *tAgency*, *tBusinessLine*, *tOwner*, *tCreative*, *tPublicity*.

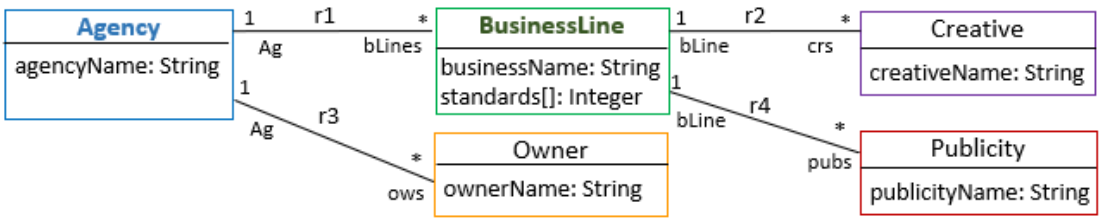


FIGURE 7.1: Diagramme UML dans un contexte de Marketing

Ce modèle est susceptible d'être traduit de plusieurs manières afin d'obtenir le schéma d'une base de données semi-structurée. La Figure 7.2 présente un possible AJSchéma nommé *Schema1* qui propose une combinaison avec trois collections contenant toutes les informations requises.

Schema1 contient trois collections *Agencies*, *Creatives* et *Owners*. Les documents de la collection *Agencies* sont définis comme du type *tAgencie*. Ce type est modifié en ajoutant la matérialisation des associations : $r1$ représentée l'attribut *bLines* de type tableau *bLines* qui contiendra des documents de type *tBusinessLine*, et $r3$ représentée par l'attribut *ows* de type tableau qui contiendra des références aux documents de type *tOwner*. D'autre part, le type *tBusiness* ajoute à son tour un nouveau niveau d'imbrication avec la matérialisation de $r2$ avec le tableau *crs* de type *tCreative*.

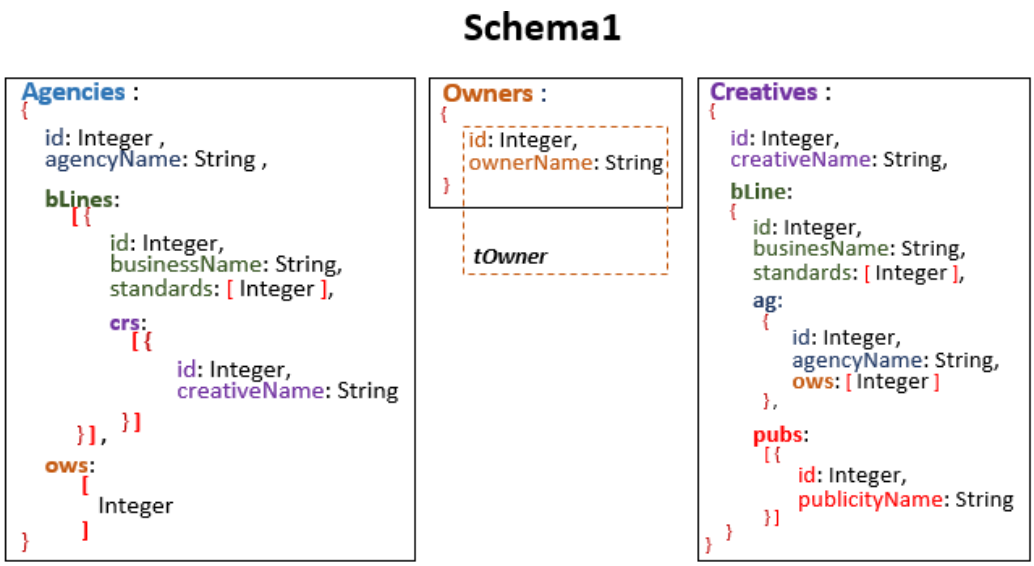


FIGURE 7.2: Exemple d'un AJSchéma basé sur le modèle UML de la Figure 7.1

La collection *Creatives* définit ses documents comme des documents de type *tCreative* avec un attribut supplémentaire *bLines* qui considère un document de type *tBusiness*. À ce type s'ajoute deux attributs, le document *ag* de type *tAgency* et le tableau de documents *pubs* de type *tPublicity*. À son tour, le document *ag* fait référence aux documents *tOwner* via un tableau.

L'arbre de la Figure 7.3 correspond à la structure du *Schema1* de la Figure 7.2. Avec cet arbre, nous pouvons rapidement apprécier différents aspects clés de la complexité de ce schéma, tels que l'existence de collections et types de documents, les niveaux d'imbrication et les différents types de documents qui peuvent être trouvés au même niveau, les références et les duplications des types. Ces aspects sont calculés automatiquement en utilisant cette représentation. Un ensemble de métriques regroupées en 6 catégories est présenté dans la suite. Nous décrivons leurs motivations, leurs équations et leur illustration à partir de cet exemple.

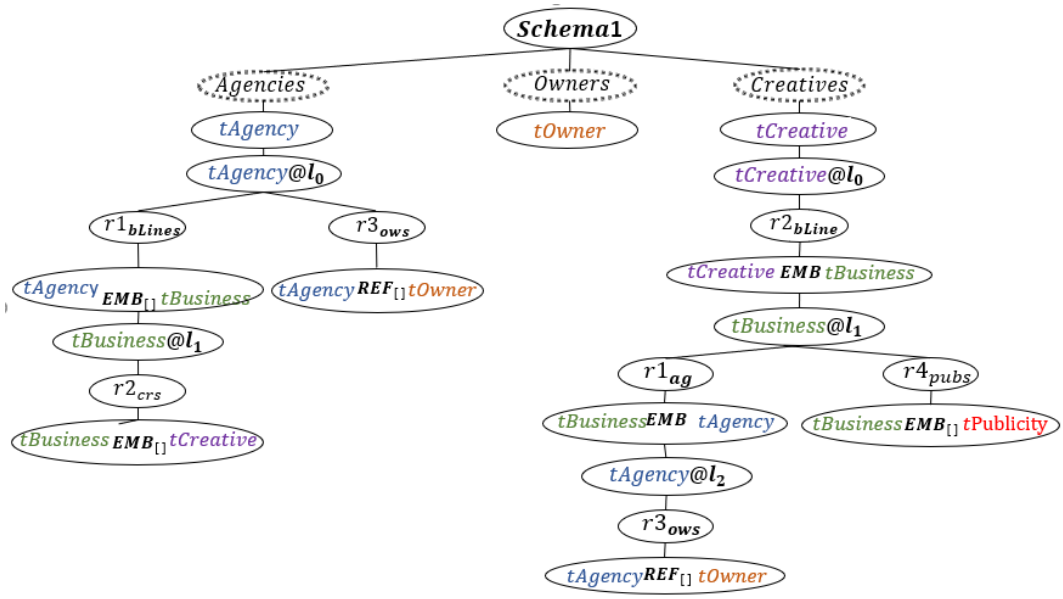


FIGURE 7.3: Représentation d'arbre correspondent au AJSchéma de la Figure 7.2

7.3 Métriques d'existence

Le choix de créer une collection pour un type du document est motivé principalement par le besoin d'accès rapide ou fréquent à l'extension du type ou à un document du type en question. Au contraire l'imbrication d'un type du document dans un autre peut être motivé par le fait que l'information est fréquemment consultée conjointement. S'assurer qu'un type du document n'est pas imbriqué à certains endroits peut aussi être intéressant, notamment si le document est peu accédé dans ce contexte ou si l'on cherche à réduire la complexité d'une collection.

Dans cette catégorie nous définissons des métriques qui reflètent l'existence d'un type du document t dans un schéma. Nous considérons deux cas : (1) l'existence d'une collection de documents de type t, et (2) la présence de documents de type t imbriqués dans d'autres documents. Ces cas sont couverts respectivement, par la métrique *colExistence* et la métrique *docExistence* définies ci-après.

Existence de collection : cette métrique (équation 7.1) permet d'identifier si les collections ont des nœuds fils au niveau 0 correspondant au type t.

$$colExistence(t) = \begin{cases} 1 & \text{le nœud } t \text{ apparaît dans le schéma} \\ 0 & \end{cases} \tag{7.1}$$

Existence de type imbriqué : cette métrique (équation 7.2) vérifie si l'imbrication de documents de type t est matérialisé dans le graphe par un nœud $*EMBt$.

$$docExistence(t,\varphi) = \begin{cases} 1 & t \in \varphi \quad \text{un nœud } *EMBt \text{ apparaît dans un chemin partant de } \varphi \\ 0 & t \notin \varphi \end{cases} \tag{7.2}$$

Notons sur la Figure 7.3 que pour les types *tAgency, tOwners* et *tCreative* il y a des collections (nœuds $@l_0$) alors que ce n'est pas le cas pour *tPublicity*. Ceux-ci existent exclusivement imbriqués dans des documents *tCreative* ($docExistence(Creatives, tPublicity) = 1$). Notons aussi que des documents de type *tBusiness* sont imbriqués dans deux collections, *Agencies et Creatives*. Nous verrons par la suite que la prise en compte de ce fait peut s'avérer pertinent dans l'analyse des schémas.

Nombre de collections : cette métrique (équation 7.3) indique le nombre de collections présentes dans le schéma. Son évaluation est le nombre de nœuds enfants de la racine du graphe.

$$nbrCol() = n \quad \text{nombre de nœuds enfants de la racine} \tag{7.3}$$

Sur la Figure 7.3, le schéma contient trois collections représentées par les nœuds enfants de la racine (*Agencies, Owners* et *Creatives*).

7.4 Métriques d'imbrication

Cette catégorie est consacrée aux métriques qui indiquent le niveau d'imbrication des documents. En général, plus une information sera imbriquée profondément, plus il sera coûteux d'y accéder sauf si l'information intermédiaire est aussi recherchée par la requête. Savoir à quel niveau d'imbrication apparaît un type du document permet d'évaluer les coûts de navigation et d'aller-retour entre les niveaux ("intra-joint") pour y accéder, ou des opérations de restructuration nécessaires pour extraire le format le plus approprié.

Profondeur d'une collection : la métrique *colDepth* (équation 7.4) indique le niveau de profondeur où se trouve le document le plus imbriqué. L'imbrication des documents est représentée par les nœuds EMB dans le graphe.

$$colDepth(\varphi) = max(depth(p_i)) \quad p_i \text{ est un chemin partant du nœud } \varphi \tag{7.4}$$

$$depth(p) = n \qquad \text{nombre de nœuds } EMB \text{ dans le chemin } p \tag{7.5}$$

Profondeur d'un schéma : la métrique *globalDepth* (équation 7.6) indique le niveau d'imbrication le plus profond des collections d'un schéma.

$$globalDepth(x) = max(colDepth(\varphi_i)) \qquad \forall \text{ collection } \varphi_i \in x \tag{7.6}$$

Connaître la profondeur d'imbrications des collections aide à mieux cerner leur cas d'utilisation et à estimer la pertinence de la structure. Les imbrications successives contribuent à une certaine forme de complexité mais n'implique pas forcément des requêtes moins performantes. Une collection très imbriquée peut être avantageuse si des requêtes prioritaires nécessitent une majorité des informations imbriquées. Si ce n'est pas le cas, l'impact des opérations de projection sera à prendre en compte (voir les métriques suivantes) ainsi que la restructuration des données pour la réponse si le chemin d'accès de la requête et le sens d'imbrication des données ne coïncident pas.

Sur l'exemple, la profondeur des collections *Owners, Agencies et Creatives* est respectivement de 0, 2, et 2. La profondeur maximale du schéma est par conséquence de 2. Notons que dans la collection *Creatives*, le type *tAgency* n'ajoute pas de niveau d'imbrication, il ajoute uniquement un tableau avec des références sur *Owners*.

Profondeur d'un type du document :

Il peut y avoir des cas dans lesquels nous serons intéressés à savoir si un type est à un certain niveau, soit pour assurer la correspondance avec le chemin de navigation qui suit une requête, soit pour prévoir l'impact de trouver l'information et reconstruire la réponse.

La métrique *docDepthInCol* (équation 7.7) indique le niveau où apparaît un type du document t dans une collection φ. Si les éléments de la collection sont que de type t (sans enfants du nœud t) la profondeur est zéro, sinon on cherche le niveau le plus profond où est imbriqué un document de ce type (nœuds $EMBt$) en suivant les chemins racine-feuilles.

$$docDepthInCol(\varphi,t) = \begin{cases} 0 & \text{le nœud fils de } \varphi \text{ n'est pas des enfants} \\ max(docDepth(p_i,t) & p_i \text{ est un chemin de la racine } \varphi \text{ à une feuille} \end{cases} \tag{7.7}$$

$$docDepth(p,t) = n \quad \text{nombre de nœuds } EMB \text{ entre la racine et } *EMBt \tag{7.8}$$

Par exemple, dans la collection *Creatives*, le niveau d'imbrication de *tPublicity* est 2, celui de *tCreative* est 0. *tCreative* est aussi imbriqué au niveau 2 de la collection *Agencies*.

Connaître les niveaux minimum et maximum où un type du document se trouve permet d'estimer combien de niveaux intermédiaires il faut traiter pour l'accès le plus ou le moins direct. Nous introduisons également les métriques *maxDocDepth* (7.9) et *minDocDepth* (7.10) qui indiquent respectivement le niveau le plus profond et le niveau le moins profond où un type du document apparaît dans le schéma.

$$maxDocDepth(t) = max(DocDepthInCol(\varphi_i,t)) \quad \varphi_i \in x \wedge t \in \varphi_i \tag{7.9}$$

$$minDocDepth(t) = min(DocDepthInCol(\varphi_i),t) \quad \varphi_i \in x \wedge t \in \varphi_i \tag{7.10}$$

Sur l'exemple, nous nous intéressons au type *tBusiness*. Notons qu'il n'y a pas une collection qui correspond à ce type. Dans le schéma, le niveau le mois et plus profond où nous trouvons ce type c'est le niveau 1, dans les collection `Agencies` et `Creatives`. Bien que dans ce cas le niveau minimum et maximum soit le même, cela ne signifie pas que le type est présent dans une seule collection.

7.5 Largeur des documents

Ici nous nous intéressons à la complexité d'un type de document en termes du nombre d'attributs et de leur type, atomique ou complexe (documents ou tableaux de documents imbriqués). Ces métriques sont motivées par le fait que des documents avec plusieurs attributs complexes peuvent induire des opérations d'accès et de projection plus conséquentes. En effet, pour extraire les attributs nécessaires à l'évaluation d'une requête, il est nécessaire d'enlever les autres attributs, ce qui s'avère plus coûteux pour des documents "larges". Lors de l'évaluation d'un schéma, on pourra notamment analyser le choix d'une structure à la fois "large" et très imbriquée.

Plusieurs facteurs peuvent contribuer à augmenter la complexité d'un type du document comme le nombre d'attributs et leur complexité. Les attributs de type document ou tableaux peut augmenter la complexité, cependant une quantité considérable d'attributs atomiques peut avoir également un

impact important. En ce qui concerne les tableaux, le traitement des éléments de type atomique peut être différent de ceux du type document. Un document avec plusieurs attributs de type tableau de documents est considéré complexe.

Afin de couvrir cet aspect, nous proposons la métrique *docWidth* (7.11). Cette métrique reflète la "largeur" d'un type du document en associant un coefficient aux types d'attributs qu'il contient tels que le type atomique ($cfAtom$), le type document ($cfDoc$), le type tableau de valeurs atomiques ($cfTblAtom$) et tableau de documents ($cfTblDoc$). La "largeur" dépendra considérablement de ces coefficients et le nombre d'attributs correspondant à son type. Les quantités d'attributs par type sont fournies par les métriques *nbrAtomicAttributes*, *nbrDocAttributes*, *nbrArrayAtomicAttributes* et *nbrArrayDocAttributes* qui déterminent la quantité d'attributs atomiques, documents, tableaux de type atomique et tableaux de type document présents dans un type de document (7.13, 7.13, 7.14, 7.15).

$$\begin{aligned} docWidth(t,\varphi) = {} & cfAtom * nbrAtomicAttributes(t,\varphi) + \\ & cfDoc * nbrDocAttributes(t,\varphi) + \\ & cfTblAtom * nbrArrayAtomicAttributes(t,\varphi) + \\ & cfTblDoc * nbrArrayDocAttributes(t,\varphi) \end{aligned} \tag{7.11}$$

$$nbrAtomicAttributes(t,\varphi) = n \quad \text{nombre d'attributs de type atomique présents dans le type t} \tag{7.12}$$

$$nbrDocAttributes(t,\varphi) = n \quad \text{nombre d'attributs de type document présents dans le type t} \tag{7.13}$$

$$nbrArrayAtomicAttributes(t,\varphi) = n \quad \text{nombre d'attributs de type tableau de valeurs atomiques} \tag{7.14}$$

$$nbrArrayDocAttributes(t,\varphi) = n \quad \text{nombre d'attributs de type tableau de documents} \tag{7.15}$$

Pour docWidth(), nous proposons d'attribuer les valeurs des coefficients en augmentant la valeur en fonction de la complexité du type des attributs afin de refléter leur complexité. $cfAtom$=1, $cfDoc$=2, $cfTblAtom$=1 et $cfTblDoc$=3. $cfTblDoc$ a le plus grand poids.

Les métriques indiquant le nombre d'attributs peuvent être utilisées séparément selon les analyses souhaitées. La taille des tableaux n'est pas prise en compte ici car elle n'est pas forcément disponible. Si c'est le cas, il semble intéressant de différencier les ordres de grandeur des tableaux. Des tableaux de taille 4 ou 5 sont du même ordre de grandeur, contrairement à des tableaux prévus pour de milliers d'éléments.

Les collections *Agencies* et *Creatives* de l'exemple, utilisent des document de type *tBusiness* mais ils n'ont néanmoins pas les mêmes attributs. Dans textitCreatives ce type inclut des tableaux d'agences et publicités (*docWidth(tBusiness,Creatives)* = 8) , contrairement à *Agencies* où le type *tBusiness* inclut seulement un tableau de creatives (*docWidth(tBusiness,Agencies)* = 4)

7.6 Taux de référencement

Le maintien de l'intégrité référentielle devient important pour les collections dont les documents sont souvent référencés par des documents d'autres collections. Pour une collection avec des documents d'un certain type t, la métrique *refLoad* (7.16) indique le nombre d'attributs (d'autres types) qui sont des références potentielles sur les documents de type t.

$$refLoad(\varphi) = n \qquad \text{soit } t@l_0 \text{ le nœud fils de } \varphi\text{, n est le nombre de nœuds } *REF\,t \text{ du schéma} \tag{7.16}$$

Pour la collection *Owners* de notre exemple, ses documents sont référencés par 2 collections : *Agencies* référence au niveau 0 en utilisant un tableaux de références alors que *Creatives* référence dans un document imbriqué au niveau 2.

7.7 Métriques de redondance

Nous nous intéressons ici à la redondance de données qui peut exister dans la base. La redondance des documents peut accélérer les accès et limiter certaines opérations coûteuses (par exemple des jointures). Cependant, elle impacte l'empreinte mémoire de la base et sa maintenabilité en matière de cohérence (complexité d'écriture des programmes et coût). Pour la métrique de redondance *docCopiesInCol* (7.17), nous utilisons l'information de cardinalité des associations conjointement avec les choix faits sur le schéma. La redondance apparaît pour certains cas de représentation de l'association par imbrication de documents.

$$docCopiesInCol(t,\varphi) = \begin{cases} 0 & : t \notin \varphi \; docExistence(\varphi,t)=0 \\ 1 & : \text{ le nœud fils de } \varphi \text{ n'est pas des enfants} \\ \prod card(r_{rol},t) & : \begin{array}{l} r_{rol} \text{ parent de un nœud } EMB \\ \text{dans le chemin entre } \varphi \text{ et } *EMBt \end{array} \end{cases} \tag{7.17}$$

$$card(r,\varepsilon) = n \qquad \text{cardinalité de } r \text{ du côté } \varepsilon \text{ dans le diagramme UML} \tag{7.18}$$

Dans la collection *Creatives* du schéma de la Figure 7.2, l'attribut pour business, nommé *bLine*, introduit de la redondance pour les agences. Par l'association $r1$ du modèle UML de la Figure 7.1, une agence A peut être associée à n1 business. Il y aura donc autant de copies du document A. Si de plus un business est référencé par n2 designers (association r2), il y aura n1 x n2 copies du document A.

Par ailleurs, nous proposons la métrique *docTypeCopies(t)* (7.19) qui indique le nombre de fois qu'un type du document est utilisé dans le schéma. Ceci reflète le nombre de structures qui peuvent potentiellement stocker des documents de type t. Cette métrique utilise la métrique d'existence.

$$docTypeCopies(t) = \sum_{i=1}^{|x|} docExistence(\varphi_i,t) \tag{7.19}$$

Dans la section 7.3, nous avons repèré l'existence du type *tBusiness* en le cherchant collection par collection du schéma. A l'aide de cette dernière métrique, nous pouvons indiquer que le type *tBusiness* se trouve présent dans 2 collections.

7.8 Synthèse et Conclusion

Dans ce chapitre, nous nous sommes intéressés à des questions de qualité des structures de données pour des bases de documents JSON, telles que MongoDB. La flexibilité de structuration de ces bases est appréciée par la souplesse permise pour représenter des données semi-structurées. Cependant, cette flexibilité a un coût dans les performances, le stockage, la lisibilité et la mantenabilité des bases et des applications. Ainsi, le choix de la structuration des données est très importante et ne doit pas être négligé.

Catégorie	Nom de métrique	Description	Par		
			sch	Col	type
Existence	*colExistence*	Existence d'une collection		x	
	docExistence	Existence d'un type du document dans une collection		x	x
	nbrCol	Nombre de collections	x		
Imbrication	*colDepth*	Profondeur maximale d'une collection		x	
	globalDepth	Profondeur maximale d'un schéma	x		
	DocDepthInCol	Niveau où un type du document se trouve dans une collection		x	x
	maxDocDepth	Niveau le plus profond où apparaît un type du document	x		
	minDocDepth	Niveau le mois profond où apparaît un type du document	x		
Largeur	*docWidth*	"Largeur" d'un type du document		x	x
	nbrAtomicAttributes	nombre d'attributs de type atomique			x
	nbrDocAttributes	nombre d'attributs de type document			x
	nbrArrayAtomicAttributes	nombre d'attributs de type tableau de valeurs atomiques			x
	nbrArrayDocAttributes	nombre d'attributs de type tableau de documents			x
Référencement	*refLoad*	Nombre de références à une collection		x	
Redondance	*docCopiesInCol*	Copies d'un type du document t dans une collection		x	x
	docTypeCopies	Nombre d'utilisations d'un type du document	x		

TABLE 7.1: Métriques structurelles proposées

Nous avons défini des métriques structurelles pour des structures JSON afin de révéler des aspects de leur complexité. Elles sont résumées dans le tableau 7.1. Elles couvrent des aspects tels que la existence, l'imbrication, la largeur, le referencement et la redondance. Ces aspects sont évalués dans le cadre des collections, des types ou du schéma entier. Afin de faciliter la manipulation et l'évaluation automatique, nous nous appuyons sur l'abstraction AJSchéma et sa représentation arborescente précédemment introduite.

Notre proposition s'est basé sur divers travaux en génie logiciel, dans la théorie de graphes, dans le contexte des bases orientées objet et XML. Les métriques d'existence d'un élément et leur

nombre sont utilisées dans ces travaux selon leur modèle de données. Les métriques d'imbrication sont proches des métriques de longueur de chemin définies dans des structures arborescentes et de graphes. La métrique de referencement d'une collection peut être rapprochée du dégrée entrant d'un nœud dans une modélisation avec un nœud d'un graphe. Si un document et ses propriétés sont modélisés avec des nœuds d'un graphe, la largeur d'un document peut être vue comme le dégrée sortant du nœud. A propos des métriques en génie logiciel, nous pouvons faire un parallèle avec des métriques statiques : l'imbrication de structures avec les appels imbriqués, le referencement avec le couplage de composants et la largeur de documents avec le nombre d'attributs et de méthodes d'une classe.

Les métriques structurelles que nous proposons reflètent des éléments de complexité du schéma qui jouent sur des aspects de qualité de la base. Elles peuvent ainsi être utilisées pour analyser et comparer différentes manières de structurer les données. Les métriques proposées n'ont pas l'ambition de représenter un ensemble complet mais sont une base qui peut évoluer. L'évaluation automatique des métriques a été implémentée dans le prototype *ScorusTool* développé dans cette thèse. Ce prototype ainsi qu'un scenarii d'utilisation des métriques sont présentés dans le chapitre 8.

"Il faut viser la lune, parce qu'au moins, si vous échouez, vous finirez dans les étoiles."

Oscar Wilde

8

Validation et mise en œuvre

.

8.1 Introduction

Ce chapitre réuni plusieurs aspects des propositions de cette thèse : la validation, l'expérimentation et la mise en oeuvre. Comme vu précédemment, l'approche SCORUS et nos contributions visent à faciliter l'analyse des structurations possibles des bases orientées documents. Tenant compte du coût de création et de manipulation de ces bases, nos propositions se placent au niveau structurel. Elles permettent d'assister le développeur, notamment en amont de la création des bases de documents, et nécessitent peu d'effort de sa part. L'idée est de permettre d'apprécier facilement les alternatives de structuration, et leurs caractéristiques, et ainsi contribuer à un choix éclairé.
Dans l'approche SCORUS nous proposons

— de travailler sur un modèle UML des données et de générer automatiquement un ensemble de variantes d'AJSchémas,
— d'évaluer les AJSchémas avec les métriques structurelles proposées et
— d'analyser et comparer ces AJSchémas à l'aide de ces métriques avec un guidage par les critères prioritaires pour les utilisateurs.

Ces trois phases peuvent composer un processus de conception ou être utilisées indépendamment à des moments différents du cycle de vie de la base de documents.

Dans la section 8.2, nous présentons un scénario d'utilisation de l'approche SCORUS. Ce scénario est axé sur l'utilisation des métriques structurelles et l'analyse et la comparaison d'AJSchémas en fonction d'un contexte applicatif. Nous nous sommes intéressés à la structuration des données utilisées dans l'étude expérimentale initiale (présentée dans le chapitre 4) en considérant des structurations non analysées précédemment.

Suite aux résultats intéressants obtenus par cette analyse structurelle, nous avons mené une étude de performances sur des bases MongoDB opérationnelles. Cette étude est présentée dans la section 8.3. Nous avons créé de multiples bases que nous avons peuplées avec un jeu de données conséquent avec toutes les structures considérées par l'analyse structurelle. Comme dans notre étude expérimentale initiale, nous avons étudié les besoins de stockage et avons programmé les requêtes pertinentes pour analyser leurs performances d'exécution. A nouveau, notre objectif premier est d'analyser l'impact de la structuration des données et voir les performances des requêtes sur des bases organisées selon les AJSchémas "recommandés" par l'analyse structurelle réalisée.

La troisième partie de ce chapitre, la section 8.4, est consacrée au prototype *ScorusTool*. Nous présentons l'architecture globale du prototype et ainsi que les principes de mise en oeuvre des modules développés. Nous approfondissons notamment les modules liés au générateur automatique des structurations pour un modèle UML et l'évaluateur automatique de métriques structurelles.

La mise en oeuvre du générateur d'alternatives de structuration suit une approche originale dans la communauté des bases de données. Elle s'inspire des lignes de produit logiciel et utilise une stratégie de variabilité avec des modèles de caractéristiques. Nous avons utilisé le langage FAMILIAR [87] qui permet de manipuler un modèle de caractéristiques et générer les alternatives possibles.

La section 8.5 de ce chapitre est consacrée à nos conclusions.

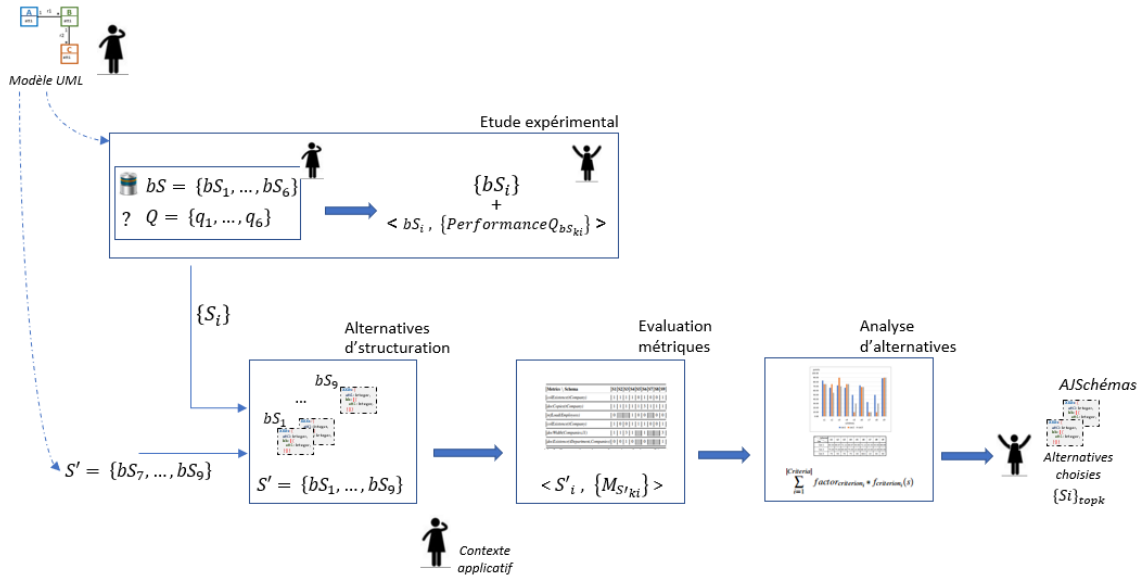


FIGURE 8.1: Scénario de validation de l'approche SCORUS

8.2 Scénario de validation de l'approche SCORUS

Les travaux présentés ici cherchent à illustrer et donner une forme partielle de validation de l'approche préconisée dans cette thèse. Le scénario présenté (cf. Figure 8.1) suit les trois phases déjà évoquées, en considérant, plusieurs variantes de structuration de données, en évaluant les métriques structurelles et en réalisant une analyse guidée pas les critère prioritaires d'un cadre applicatif. Il s'agit en priorité de faire émerger la ou les structurations (nous utiliserons la désignation AJSchémas dans la suite) les plus favorables selon certains critères mais aussi d'écarter des choix très défavorables ou encore, d'envisager des alternatives pas forcement évidentes naturellement.

L'analyse structurelle présentée ici portera sur les données d'entreprise (cf. Figure 8.2) utilisées dans le chapitre 4. Nous étudierons des AJSchémas qui suivent les directives de modélisation semi-

structurée introduites dans le chapitre 5. Le contexte applicatif se base sur les requêtes fréquentes fournies par notre étude expérimentale initiale. Ce contexte permet d'établir des critères de choix et d'identifier des métriques structurelles connexes qui seront utilisées pour évaluer et comparer les AJSchémas.

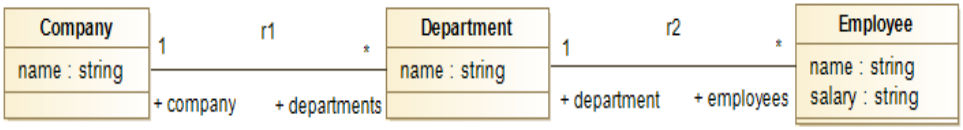


FIGURE 8.2: Diagramme UML des données d'entreprise

8.2.1 Alternatives de structuration

Nous nous plaçons dans le cas d'un développeur qui va gérer les données modèlisées dans la Figure 8.2 dans une base orientée documents type MongoDB. Pour le choix des structurations, nous proposons la génération automatique d'AJSchémas susceptibles d'être utilisés. La génération des AJSchémas suit l'algorithme *AMISS* (cf. section 6.5).

Nous utilisons ici un sous-ensemble de neuf possibilités, $S = \{S1, \ldots, S9\}$, illustrées dans la Figure 8.3. Les AJSchémas $S1$ à $S6$ correspondent à une abstraction de la structuration des données des bases utilisées dans l'étude expérimentale initiale. Les structurations proposées par les AJSchémas $S7$ à $S9$ n'avaient pas été envisagées précédemment.

Les AJSchémas de l'ensemble S présentent divers choix concernant l'existence de collections, l'imbrication de documents, le référencement et la duplication de documents. Dans certains cas il y a différents niveaux d'accès et plusieurs copies d'un même document. Les caractéristiques dominantes de ces AJSchémas sont :

— *Structuration avec référencement de documents et sans d'imbrication :* S1 et S4

— *Collection unique et imbrication totale des données :* S3, S5, S7

— *Structuration avec imbrication et référencement de documents :* S2, S8, S9

— *Structuration avec imbrication de documents et duplication potentielle des documents imbriqués :* S5, S6, S7, S8

— *Structuration avec imbrication de documents et duplication de documents sur deux collections ou plus :* S6, S9

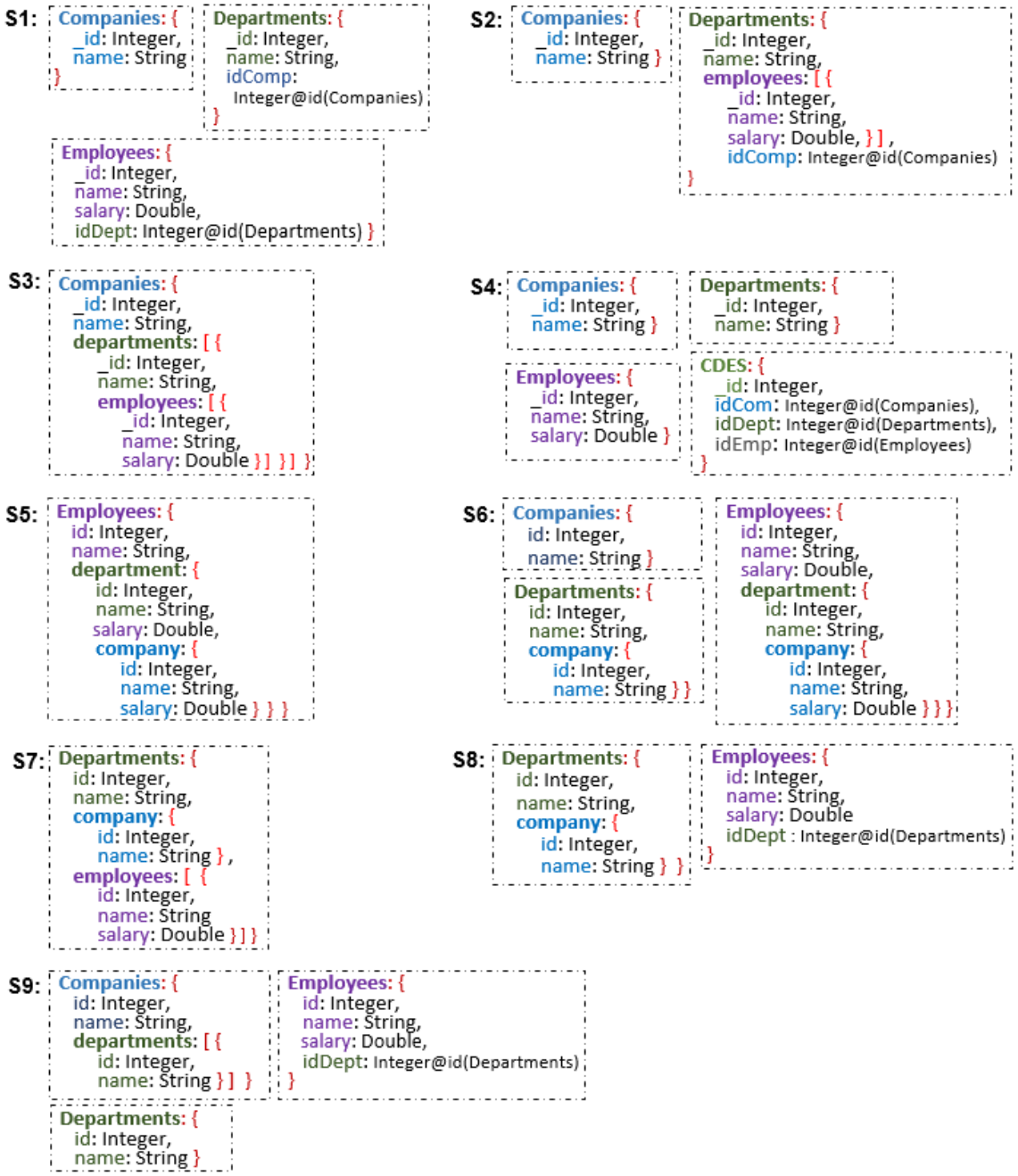


FIGURE 8.3: Ensemble d'AJSchémas étudiés

8.2.2 Métriques structurelles et critères applicatifs

Nous reprenons le contexte applicatif de l'étude expérimentale initiale afin d'établir les critères prioritaires dans le choix de la structuration des données. Nous disposons des informations suivantes. Les requêtes les plus prioritaires portent sur les entreprises et le nom de leurs départements (priorité forte) mais aussi pour connaître l'employé qui a le salaire le plus élevé dans l'entreprise en donnant l'identifiant de l'entreprise ou le nom de l'entreprise. Ces informations sur les accès prioritaires ainsi que d'autres informations permettent d'établir des critères d'analyse des AJSchémas (cf. Table 8.1). Tenant compte des accès prioritaires, les entreprises jouent un rôle important (critère1) ainsi que la facilité pour manipuler leurs instances (critère 5). Les départements sont accédés via les entreprises (critère 6). De plus, on sait que la cohérence des données sur les entreprises est importante. Il est donc préférable de limiter les copies des données des entreprises (critère 2). Par ailleurs, l'accès à l'ensemble des employés exclusivement n'est pas prioritaire (critère 4).

	Critères à considérer	**Métriques utiles**
1	Favoriser l'existence de la collection **Companies**	*colExistence*
2	Éviter les copies de documents **tCompany**	*docCopies*
3	Réduire les références à la collection **Employees**	*refLoad*
4	L'existence d'une collection **Employees** n'est pas prioritaire	*colExistence*
5	Réduire la complexité de la collection **Companies**	*colExistence, docWidth*
6	Privilégier l'imbrication de **tDepartment** dans **Companies**	*colExistence, docExistence, docDepthInCol*

TABLE 8.1: Critères applicatifs retenus et métriques utilisées

Une fois les critères à prendre en compte définis, nous avons identifié une ou plusieurs métriques pertinentes à évaluer en fonction de chaque critère et en considérant des aspects structurels impliqués. Les métriques identifiées par rapport aux critères correspondants sont introduites dans le tableau 8.1. Les critères 1, 4, 5 et 6 dépendent des informations des entreprises et des employés en tant que collections, la métrique *colExistence* est donc associée à ces critères. Comme nous nous intéressons à la cohérence des informations des entreprises à travers leurs copies potentielles, nous suggérons d'utiliser la métrique *docCopies* (critère 2). Afin d'identifier si les départements sont dans la collection Companies, nous proposons d'utiliser la métrique *docDepthInCol* qui permet de savoir si un type de document est imbriqué dans une collection (critère 6).

8.2.3 Évaluation des métriques structurelles

Nous avons évalué les métriques identifiées pour les neuf AJSchémas étudiés. Les résultats sont présentés dans la table 8.2. Cette table montre, par exemple, l'évaluation de la métrique *docCopies* sur le type *tCompany* pour l'ensemble d'AJSchémas. Cette métrique indique le nombre de collections où le type de document *tCompany* est présent. Notons que le type *tCompany* est présent dans une seule collection dans chaque schéma, à l'exception du schéma *S*6, où il existe dans trois collections. Les cases grises indiquent que la métrique ne s'applique pas à ce AJSchéma.

Métriques \ Schémas	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
colExistence(tCompany)	1	1	1	1	0	1	0	0	1
docCopies(tCompany)	1	1	1	1	1	3	1	1	1
refLoad(Employees)	0			1	0	0		0	0
colExistence(tEmployee)	1	0	0	1	1	1	0	1	1
docWidth(Companies,l1)	1	1	3	1		1			3
docExistence(tDepartment,Companies)	0	0	1	0		0			1

TABLE 8.2: Évaluation des métriques sur les AJSchémas étudiés

8.2.4 Analyse des AJSchémas

Dans cette phase nous allons analyser et comparer les neuf AJSchémas sur la base des métriques et des critères applicatifs. La table 8.3 illustre la formalisation des critères et leur évaluation pour chaque AJSchéma. Chaque critère est formalisé comme une fonction dont la valeur est à maximiser ou à minimiser. Les valeurs des critères ont été normalisées entre 0 et 1.
Par exemple, le critère 2 indique qu'il est important de limiter le nombre de copies des données des entreprises. Ceci est traité en minimisant la valeur de $docCopiestCompany(s)$. L'AJSchéma qui aura la plus petite valeur pour cette métrique est considéré comme le plus intéressant et gagnera un maximum de points pour le critère 2. Ici, pour le critère 2, tous les AJSchémas ont la valeur maximale 1 excepté S6. En effet, S6 stocke des entreprises dans trois collections ce qui est interprété comme trois copies potentielles.

L'évaluation de chaque critère introduit un ordre relatif entre les AJSchémas et permet de les classer en fonction de chacun d'eux. Par exemple, au regard du critère 4, les AJSchémas S1, S4, S5, S6, S8 et S9 sont à privilégier par rapport aux autres.

	Critère \ AJSchéma	S1	S2	S3	S4	S5	S6	S7	S8	S9
1	$f_{c_1}(s) = colExistenceCompanies^{max}(s)$	1.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	1.00
2	$f_{c_2}(s) = docCopiestCompany^{min}(s)$	1.00	1.00	1.00	1.00	1.00	0.33	1.00	1.00	1.00
3	$f_{c_3}(s) = refLoadEmployees^{max}(s)$	1.00	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00
4	$f_{c_4}(s) = colExistenceEmployees^{max}(s)$	1.00	0.00	0.00	1.00	1.00	1.00	0.00	1.00	1.00
5	$f_{c_5}(s) = levelWidthCompaniesL_1{}^{min}(s)$	1.00	1.00	0.33	1.00	0.00	1.00	0.00	0.00	0.33
6	$f_{c_6}(s) = docDeptInCompanies^{min}(s)$	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00

TABLE 8.3: Évaluation des critères d'analyse des AJSchémas

Cependant, le fait de privilégier un AJSchéma dépend de l'ensemble des critères. Par conséquent, pour prendre des décisions en les impliquant tous, nous adoptons une analyse multi-critères basée sur une somme pondérée des critères par AJSchéma.

L'importance de chaque critère est reflétée par son poids dans la somme pondérée. La fonction d'évaluation d'un AJSchéma, noté *schemaEvaluation* fait la somme pondérée des critères. Une évaluation élevée reflète une bonne adaptation d'un AJSchéma aux besoins considérés.

$$schemaEvaluation(s) = \sum_{i=1}^{|Criteria|} factor_{criterion_i} * f_{criterion_i}(s) \tag{8.1}$$

Nous avons considéré trois cas qui ont donné lieu à trois pondérations différentes introduites dans la table 8.4 :

— Cas 1, tous les critères ont la même importance.

— Cas 2, priorité aux critères concernant la facilité d'utilisation de l'extension des entreprises.

— Cas 3, ajout de priorité de la collection Employees en supposant qu'il est motivé par un nouveau patron d'accès important.

Le cas 2 représente le fait que les requêtes les plus prioritaires portent sur les entreprises et les noms de leurs départements. Ceci est exprimé par le critère C1 mais aussi, en partie, avec les critères C5 et C6. Ainsi, les pondérations proposées dans ce cas donnent beaucoup de poids à C1 et renforcent C5 et C6 par rapport aux autres critères.

Dans le cas où l'on ne disposerait pas d'information prioritaire sur un critère, on utiliserait une pondération uniforme pour tous les critères comme dans la cas 1. Ceci peut aussi être motivé par le fait que les critères considérés représentent des priorités qui alternent dans le temps. Les critères ne sont pas tous prioritaires au même moment. Dans ce cas on peut préférer une structuration qui représente un compromis.

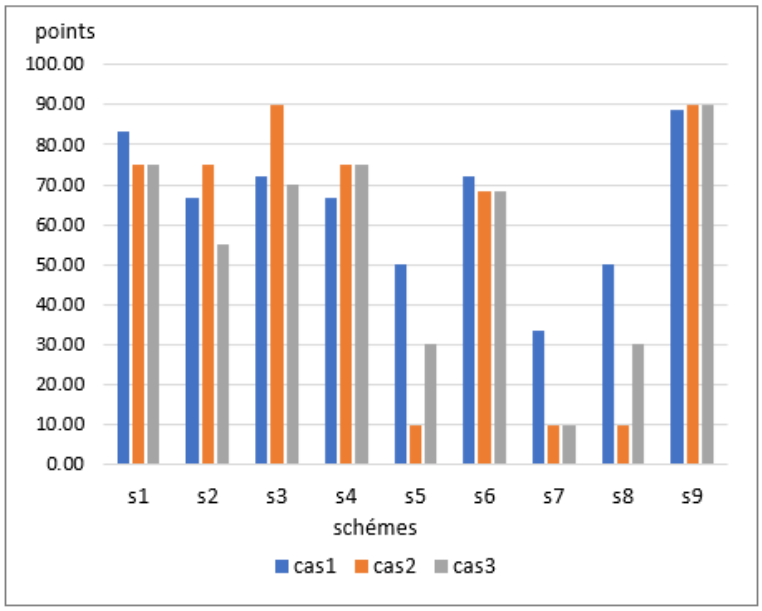
Les cas 3 est introduit dans cet étude pour considérer une situation où, en plus des priorités déjà connues, on doit également faciliter l'accès aux données des Employées. Dans ce cas, le critère 4 devient important contrairement à la situation du cas 2 qui est "dominé" par le critère 1. Ce critère 1 est indispensable pour l'accès directe à l'information des entreprises
.

Critère	**Facteur**		
	Cas 1	Cas 2	Cas 3
Critère 1	16.667	50	30
Critère 2	16.667	10	10
Critère 3	16.667	0	0
Critère 4	16.667	0	20
Critère 5	16.667	15	15
Critère 6	16.667	25	25

TABLE 8.4: Facteur d'évaluation des 6 critères pour les cas retenus

La Figure 8.4 montre le résultat de l'évaluation des neuf AJSchémas pour les trois cas. Le choix des pondérations reste un sujet à approfondir.

Les évaluations placent les AJSchémas S5, S7, S8 comme les moins bons dans les trois cas considérés. La structuration dans S5 et S7 utilise une seule collection qui n'est pas prioritaire dans les critères pris en compte. S3 se démarque dans le cas 2, pour la forte priorité de la collection *Companies*. Companies est la seule collection de S3 et de plus elle imbrique des documents en accord avec les critères recherchés.



Schema / Cas	s1	s2	s3	s4	s5	s6	s7	s8	s9
Cas 1	83.33	66.67	72.22	66.67	50.00	72.22	33.33	50.00	88.89
Cas 2	75.00	75.00	90.00	75.00	10.00	68.33	10.00	10.00	90.00
Cas 3	75	55	70	75	30	68.3	10	30	90

FIGURE 8.4: Évaluation de 9 AJSchémas selon les trois cas retenus

S9 et S6 sont parmi les AJSchémas dont les scores sont stables pour les trois cas. S6 paie le coût de ne pas imbriquer les départements dans la collection *Companies* et d'introduire de la redondance de documents alors qu'on préfère l'éviter (critère 2). S9 est le meilleur car il correspond à tous les critères. Les bons résultats aux trois cas dénote une forme de "polyvalence" de cet AJSchéma qui permet de résister aux évolutions des priorités. S9 a été une surprise de cette étude car sa structuration n'avait pas été jugée "naturelle" par le développeur. Pour consolider notre analyse, nous avons réalisé une campagne d'analyse de performances avec des bases MongoDB. Celle-ci est présenté dans la section suivante.

8.3 Expérimentation à grande échelle

Dans le but d'étudier le comportement des bases créées selon les AJSchémas retenus dans le scénario de validation, nous avons mené une nouvelle expérimentation. Nous souhaitons notamment voir et valider les performances des requêtes sur les bases structurées selon $S7, S8$ et $S9$ qui n'avaient pas été considérées dans l'étude expérimentale initiale présentée dans le chapitre 4. Pour une validation plus conséquente, nous avons travaillé ici avec un jeu de données plus grand que celui de l'expérience du chapitre 4.

8.3.1 Configuration

Pour cette expérimentation, nous avons travaillé avec 18 bases MongoDB. Nous avons créé neuf bases bSi', sans index, et neufs copies avec index, nommées $bSi' - Ix$. Chaque base contient les données de 100 entreprises, 20 départements par entreprise et 2000 employés par département. Au total, une base stocke l'information sur 2000 départements et 4000000 d'employés. Ces données sont représentées dans les bases de manières différentes. La structure d'une base bSi' (respectivement $bSi - Ix$) correspond à celle donnée par l'AJSchémas Si. Les 9 structurations considérées sont illustrées sur la Figure 8.3. L'information détaillée des index est présentée dans la section 8.3.5.

Dans cette expérimentation nous avons utilisé le même ensemble de requêtes Q que dans l'expérimentation initiale. Nous avons notamment programmé les requêtes pour les bases correspondant à $S7, S8$ et $S9$ et nous avons re-utilisé les programmes pour $S1$ à $S6$ développés pour l'étude initiale.

L'expérience a été exécutée sur un serveur dédié avec 2 CPU socket, 4 core hyperset dupliqués logiquement [1]. Nous avons utilisé la version 3.2.7 du système MongoDB. Chaque requête a été exécutée 31 fois sur chaque base. Nous avons analysé les performances sur la base des 30 exécutions à chaud. La première exécution a été écartée.

1. L'étude du chapitre 4 a utilisé un ordinateur portable processeur Intel Core i7 avec 1,8 GHz et 8 Go de RAM et avec MongoDb 3.2

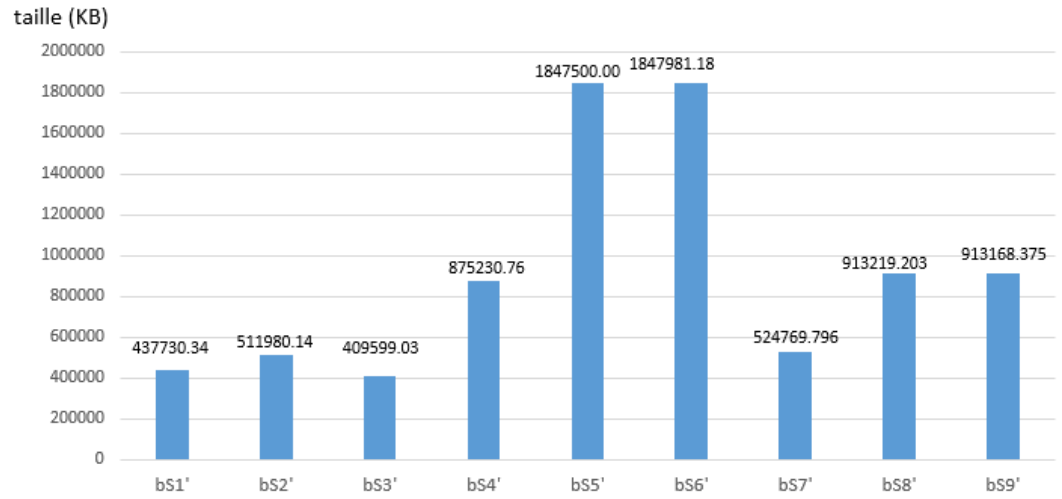


FIGURE 8.5: Taille des bases de documents *bSi'*

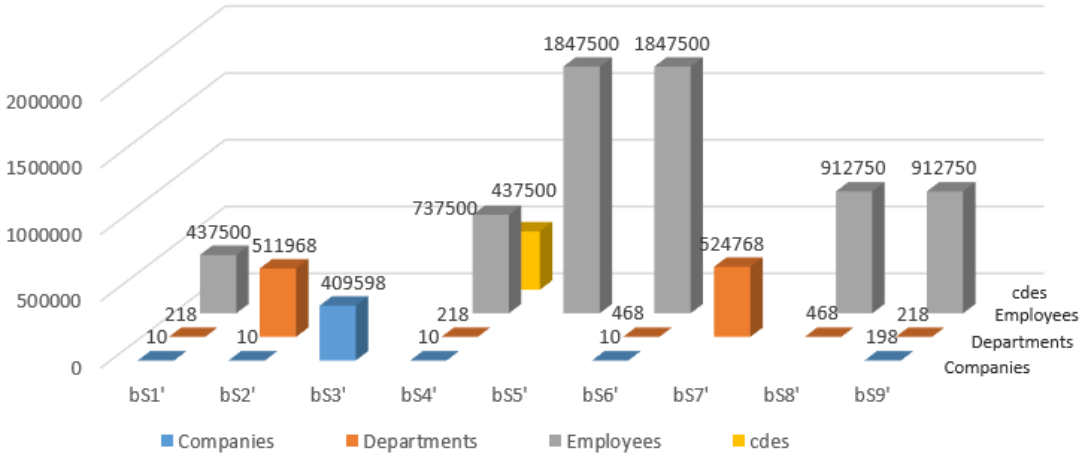


FIGURE 8.6: Taille des collections dans chaque base

8.3.2 Bases de documents sans index

La Figure 8.5 montre la taille des bases de documents *bSi'* implémentées pour cette expérience. Ici nous considérons exclusivement les neuf bases sans index. La Figure 8.6 montre la taille de chaque collection dans chacune de ces bases.

Notons la grande taille des bases $bS5'$ et $bS6'$ par rapport aux autres bases. Ceci est dû, principalement, à la collection `Employees` qui a une structure entièrement imbriquée et à la duplication des données. En effet, les documents concernant `Company` et `Department` sont dupliqués dans la collection `Employees` et dans les deux collections supplémentaires `Companies` et `Departments`. Les bases $bS3'$ et $bS7'$ sont relativement petites. Elles contiennent également des collections avec des structures entièrement imbriquées mais les données ne sont pas dupliquées dans d'autres collections. Ces bases gèrent une collection unique. Dans la collection `Departments` de $bS7'$, les données des entreprises peuvent être dupliquées mais pas celles des employés. Dans la base $bS3'$ aucune donnée n'est dupliquée.

Les collections de $bS4'$ et de $bS1'$ représentent les associations entre entités par des références entre documents. Il n'y a pas d'imbrication de documents. Notons sur la Figure 8.5 que la taille de $bS4'$ double celle de $bS1'$. Ceci est dû à la collection `CDES` qui matérialise les associations entre les entreprises, ses départements et ses employés.

En ce qui concerne les bases $bS8'$ et $bS9'$, leurs structures combinent imbrication, référencement et duplication de documents. Dans $bS8'$ les données des entreprises peuvent être dupliquées dans la collection `Departments` car une entreprise a N départements. La base $bS9'$, quand à elle, duplique explicitement les données des départements en créant une collection `Departments` et en les imbriquant au sein de leur entreprise respective dans la collection `Companies`. Dans `Companies` chaque département n'apparaîtra qu'une fois.

La taille de $bS8'$ et de $bS9'$ reste moyenne comparée aux autres bases. Elles sont plus volumineuses que $bS1'$ et $bS3'$ qui ne dupliquent pas de données, mais bien plus petites que $bS5'$ et $bS6'$. $bS5'$ et $bS6'$ ont des données dupliquées et leur collection `Employees` est très imbriquée.

Pour l'ensemble des bases, l'un des principaux facteurs influant sur leur taille est la structure des employés. D'une part, les employés constituent la plus grande quantité de données à stocker et, d'autre part, ils ont des associations dont la matérialisation peut entraîner la duplication de données.

8.3.3 Requêtes mises en œuvre

Pour cette expérimentation nous reprenons le même ensemble de requêtes Q utilisé pour l'étude expérimentale présentée dans le chapitre 4, section 4.3.3. Le jeu de données ici est bien plus important et nous considérons les trois nouvelles alternatives de structuration de données.

Rappelons les requêtes utilisées :
[Q1] Les employés ayant un salaire égal à $1003.
[Q2] Les employés ayant un salaire plus élevé que $1003.
[Q3] Les employés ayant le salaire le plus élevé.
[Q4] Les employés ayant le salaire le plus élevé par entreprise et l'identifiant de l'entreprise.
[Q5] Les employés ayant le salaire le plus élevé par entreprise et le nom de la société.
[Q6] Le salaire le plus élevé.
[Q7] Les informations sur les entreprises, y compris le nom de leurs départements.

La mise en œuvre de ces requêtes est spécifique à chaque base tenant compte qu'elles utilisent chacune une structuration de données différente (cf. Figure 8.3). Cela signifie que pour chaque requête, 9 programmes différents ont été développés, pour un total de 81 implémentations. Pour cette expérimentation, nous avons utilisé les mêmes implémentations de l'expérience initiale pour les bases $bS1$ à $bS6$ (structurations S1 à S6) et nous avons programmé les requêtes pour les nouvelles bases structurées selon S7,S8 et S9. Nous étudions les requêtes par rapport à leurs facilité de programmation et leur performances. Les mêmes programmes sont testés sur les bases sans index et avec index.

Rappelons que la programmation des requêtes est dite impérative dans le système MongoDB. Ce système ne propose pas de langage déclaratif. Cela introduit une très forte dépendance vis-à-vis de la manière dont sont structurées les données. Par conséquent, la mise en œuvre d'une requête peut différer beaucoup d'un schéma à un autre. Le nombre de collections lues et la complexité des types des documents stockés sont déterminants pour l'écriture des requêtes. A titre d'exemple, la Figure 8.7 montre la requête Q3 pour la base $bS4'$ et la Figure 8.8 présente cette même requête pour la base $bS3'$.

```
db.employees.aggregate(
   [
       {$group:{_id:null, "maxValue": {$max:"$salary"}}}
   ],
  {cursor:{}}
).forEach(function(g)
{

    db.employees.find({ "salary":g.maxValue}).forEach(function(e)
    {
        results.push( e);
    })

});
```

FIGURE 8.7: Requête Q3 pour la base $bS4'$

```
db.companies.aggregate(
    [
        {$project:{"departments.employees":1, _id:0}},
        {$unwind:"$departments"},
        {$unwind:"$departments.employees"},
        {$group: { _id: null, maxSalary: {$max: '$departments.employees.salary'}}}
    ],
    { cursor:{} }
).forEach(

    function(g){
        db.companies.aggregate(
            [
                {$match:{ "departments.employees.salary": g.maxSalary } },
                {$project:{"departments.employees":1, _id:0}},
                {$unwind:"$departments"},
                {$unwind:"$departments.employees"},
                {$match:{ "departments.employees.salary": g.maxSalary } }
            ] ,
            { cursor:{} }
        ).forEach(
            function(e){
                results.push( e );
            }
        )
    }

);
```

FIGURE 8.8: Requête Q3 pour la base $bS3'$

Le développement de chaque requête peut être coûteux en termes de temps et d'efforts humains à cause de l'absence de langage déclaratif. Pour les développements avec JavaScript, l'implémentation est particulièrement dépendante des décisions du développeur.

Notons que pour certaines requêtes de cette expérience, l'utilisation de l'opérateur *lookup* a été envisagée car elle semblait appropriée. Néanmoins, l'implémentation de jointures naturelles avec cet opérateur s'est avéré très peu performante. Nous avons alors préféré la mise en œuvre d'un autre algorithme de jointure.

8.3.4 Analyse de performances des requêtes

Cette section présente les résultats de l'expérimentation. Nous présentons les performances des requêtes pour chaque base et discutons des caractéristiques de la structuration de données qui sont déterminantes. Au regard des performances sur les bases, nous discutons le résultat de l'analyse

	Critères du scénario	Requêtes connexes	Commentaire
C1	**Favoriser** l'existence de la collection **Companies**	Q7, Q4, Q5	
C2	Éviter les copies de documents **tCompany**		Critère favorable pour minimiser la taille de la base et faciliter le maintien de la cohérence
C3	Réduire les références à la collection **Employees**	Q4, Q5	
C4	Existence **pas prioritaire** d'une collection **Employees**	Q4, Q5 (Cas3 :Q1,Q2,Q3)	
C5	Réduire la complexité de la collection **Companies**	Q7	
C6	**Privilégier** l'imbrication de **tDepartment** dans **Companies**	Q7	

TABLE 8.5: Critères d'analyse structurelle et requêtes connexes

structurelle présentée précédemment dans le scénario de validation de SCORUS. Nous nous intéressons particulièrement aux requêtes en relation avec les critères de sélection (cf. table 8.5) utilisés dans le scénario de validation.

Dans la suite, nous introduisons l'analyse des expériences avec les bases sans index et dans la section 8.3.5 nous abordons les bases avec index.

Nous analysons l'exécution des requêtes $Q1$ à $Q7$ sur les bases de documents $bS1'$ à $bS9'$. Aucun index n'a été créé sur ces bases. La Figure 8.9 indique le temps d'exécution médian pour chaque requête sur les neuf bases.

Afin de faciliter la discussion sur le comportement des différentes bases, la Figure 8.10 montre, pour chaque requête, l'ordre relatif des bases au regard des performances d'exécution de la requête. Les bases sont ordonnées de gauche à droite, en commençant par celles où les performances sont les meilleures. Dans la suite nous organisons notre analyse en nous guidant par les caractéristiques principales de la structuration des données dans les bases.

Collection unique et imbrication complète des données
Nous considérons ici des bases telles que $bS3'$, $bS5'$ et $bS7$' où les données sur lesquelles nous travaillons sont regroupées dans une collection. Leur structuration est rappelée par les AJSchémas montrés sur les figures 8.11, 8.12, 8.13. Dans ces structures, la manière dont sont imbriquées les données va être très importante. Les requêtes pour lesquelles le sens de l'imbrication est favorable sont exécutées de manière très performante.

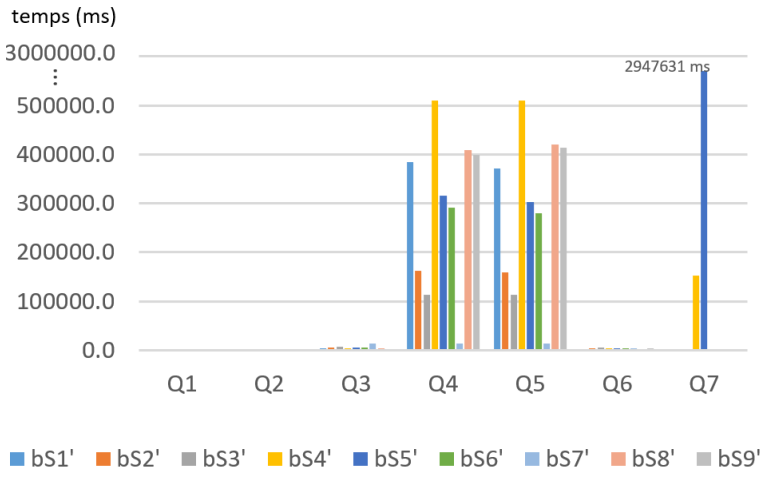


FIGURE 8.9: Médiane du temps d'exécution des requêtes par base avec setup2

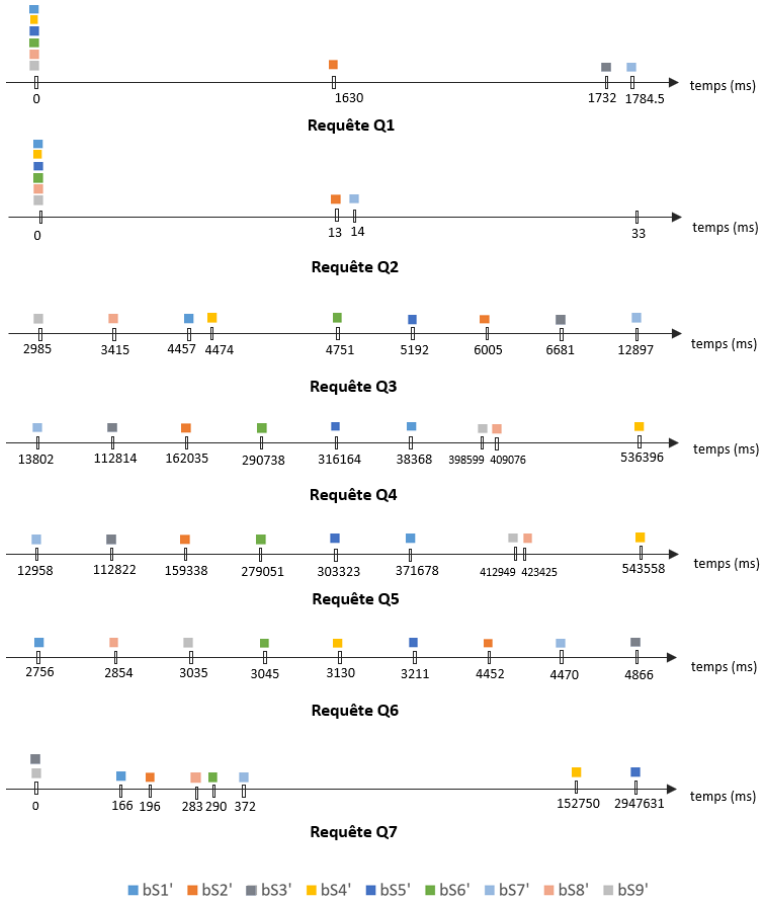


FIGURE 8.10: Synthèse des exécutions : positions relatives des bases pour chaque requête

Par exemple, considérons les requêtes Q4 et Q5 qui accèdent aux employés par rapport à leur entreprise. Dans $bS3'$, les `entreprises`, au niveau 0, imbriquent leurs `départements` au niveau 1 qui imbriquent leurs `employés` au niveau 2. L'ordre d'imbrication des données est favorable à Q4 et Q5 mais il y aura un surcoût lié au traitement des `départements` qui sont stockés "entre" les `entreprises` et les `employées`. Ce surcoût correspond à la navigation à travers les départements pour atteindre les employés mais aussi aux opérations nécessaires à enlever les données de département car ils ne doivent pas apparaître dans la réponse finale.

Q4 et Q5 présentent des bonnes performances sur $bS3$ mais les meilleures performances sont obtenues sur $bS7'$. En effet la structuration de $bS7'$ (cf. Figure 8.13) est proche de celle de $bS3'$. L'imbrication des données est favorable à la requête et ici aussi il y a un surcoût lié au traitement des `départements`. Néanmoins, ici les `départements` sont au niveau 0 ce qui rend leur traitement plus simple que dans $bS3'$.

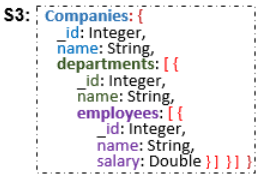


FIGURE 8.11: AJSchéma S3 correspondant la base $bS3'$

Considérons maintenant le cas de $bS5'$, où les performances de Q4 et Q5 sont mauvaises. La raison est que l'ordre d'imbrication des données est contraire au sens de regroupement nécessaire aux requêtes. L'évaluation des requêtes Q4 et Q5 nécessite des *intrajoins* qui traversent les structures imbriquées vers "le bas" et "remontent" à plusieurs reprises afin d'accéder aux données utiles et de supprimer les informations inutiles.

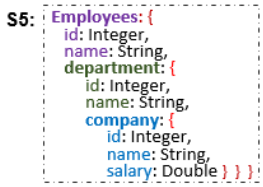


FIGURE 8.12: AJSchéma S5 correspondant la base $bS5'$

Des comportements similaires se produisent pour $Q7$ sur $bS3'$, $bS5'$, $bS7'$. $Q7$ nécessite l'information des `entreprises` et les noms de leurs `départements`. L'exécution sur $bS5'$ présente les pires performances. Elle paie le surcoût lié à la présence des données des employés au niveau 0, alors qu'ils sont inutiles pour Q7, mais aussi la difficulté de traiter les données qui sont imbriquées dans un sens contraire à ce que la requête nécessite. Sur $bS3'$, l'ordre d'imbrication est favorable à la requête alors que cela n'est pas le cas pour $bS7'$. Les performances de $Q7$ sur $bS7'$ sont néanmoins parmi les meilleures car l'accès aux données utiles n'est pas surchargé par des

traversées de données inutiles à la requête. Contrairement à $bS5'$, la traversée d'employés n'est pas ici obligatoire.

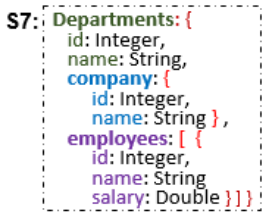


FIGURE 8.13: AJSchéma S7 correspondant la base $bS7'$

Collections séparées et imbrication de documents

Nous considérons maintenant les bases où les données sont gérées dans plusieurs collections dont certaines avec des données imbriquées et dupliquées. C'est le cas des bases $bS6'$ à $bS9'$ dont les AJSchéma sont rappelés sur les Figures 8.14, 8.13, 8.16 et 8.15. Notons que $bS6'$ étend $bS5'$ en additionnant la collection `Companies` et la collection `Departments` qui imbrique l'information des entreprises. Dans $bS6'$, les départements apparaissent dans 2 collections et les entreprises dans trois collections.

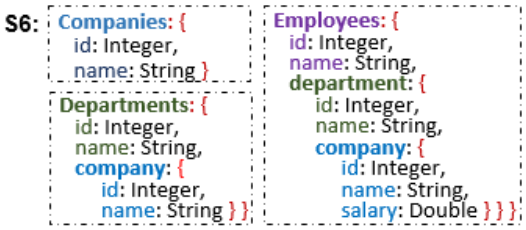


FIGURE 8.14: AJSchéma S6 correspondant la base $bS6'$

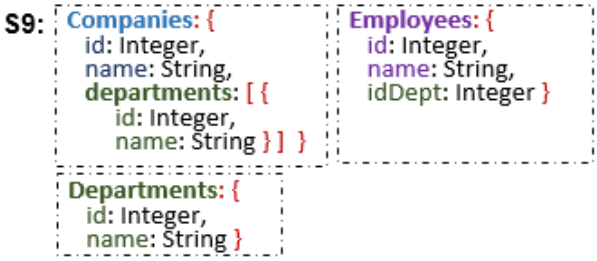


FIGURE 8.15: AJSchéma S9 correspondant la base $bS9'$

Considérons la requête Q7 qui demande l'information sur les entreprises en incluant le nom des départements. L'organisation idéale d'une collection d'entreprises avec les départements imbriqués, sans autres données, est proposée par $bS9'$. Sans surprise les performances de Q7 sur $bS9'$ sont les meilleures. Néanmoins l'évaluation sur une collection de départements avec les entreprises imbriquées, comme $bS8'$, reste dans les bonnes performances. Ce résultat est bon

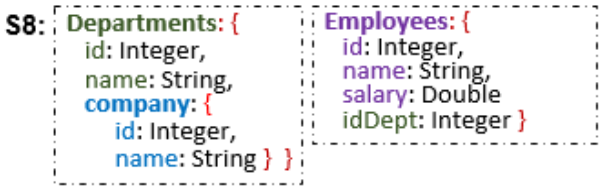


FIGURE 8.16: AJSchéma S8 correspondant la base $bS8'$

malgré l'imbrication dans le sens inverse et les aller-retour dans l'imbrication car le volume de données n'est pas très important. Notons qu'il n'est pas nécessaire de traiter des données inutiles ni pour accéder aux données recherchées, ni dans les niveaux intermédiaires. Cela explique aussi les meilleures performances sur $bS6'$ par rapport à $bS5'$.

Considérons maintenant les requêtes $Q1$, $Q2$, $Q3$, $Q6$ qui portent sur les employés exclusivement. La collection `Employees` qui donne un accès direct aux données est idéale. Elle est notamment disponible sur $bS9'$ qui se place parmi les meilleures mais aussi sur $bS8'$ par exemple.

Collections séparées sans imbrication de documents
Nous considérons des structurations avec plusieurs collections sans imbrication de documents comme S1 et S4 (cf. figures 8.17 et 8.18). Les bases $bS1'$ et $bS4'$ utilisant ces structurations montrent de très bons temps d'exécution pour les requêtes Q1, Q2 et Q3. Ceci est expliqué par le fait que l'ensemble de données à lire par ces requêtes correspond parfaitement à une collection de la base. Les requêtes qui nécessitent des données de 2 ou 3 collections telles que $Q4$ et $Q5$ sont au contraire pénalisées par la jointure.

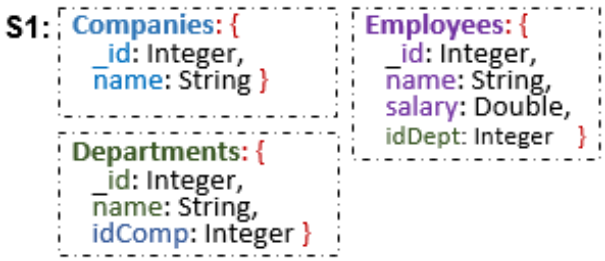


FIGURE 8.17: AJSchéma S1 correspondant la base $bS1'$

Discussion par rapport à l'analyse structurelle
Dans le cadre de cette thèse nous cherchons à aider les développeurs dans le choix de la structuration des données. Nous avons proposé une approche d'analyse structurelle qui peut être réalisée sans payer le coût de la création des bases de documents. Lors du déroulement du scénario d'analyse structurelle présenté en section 8.2, nous avons eu un résultat inattendu sur le schéma $S9$. $S9$ est ressorti comme le plus polyvalent tenant compte des trois cas étudiés. Comme les

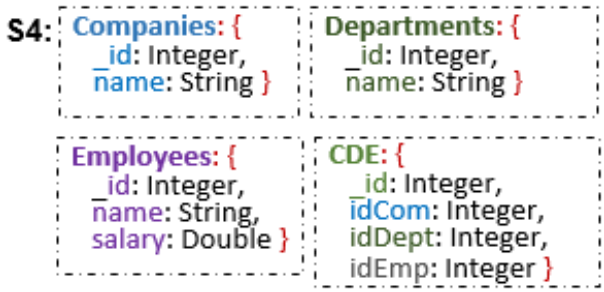


FIGURE 8.18: AJSchéma S4 correspondant la base $bS4'$

performances des requêtes lors de l'exécution sur les bases reste bien sûr un critère important, nous avons mené l'expérimentation présentée précédemment pour étudier le comportement.

Nous avons pu constater que la recommandation de $S9$ était pertinente au regard des performances des requêtes sur $bS9'$. Les bonnes performances de $Q4$, $Q5$ et $Q7$ correspondent à la bonne note attribuée à $S9$ lorsque les critères d'existence de la collection `Companies` et l'imbrication des départements dans leurs entreprises sont privilégiés. Les bonnes performances de $Q1$, $Q2$, $Q3$ et $Q6$ correspondent au bon score attribué à $S9$ lorsque les besoins évoluent et nécessitent l'accès aux employés. Lorsque tous les critères sont pris en compte avec le même poids, $S9$ est également favorable.

Notons par ailleurs les performances sur $bS1'$ (voir AJSchéma sur la figure 8.17). La structuration des données correspond à une approche qui pourrait être dite "normalisée" dans le monde relationnel. Les performances des requêtes sur $bS1'$ sont correctes, sauf pour les requêtes qui nécessitent deux jointures comme $Q4$ et $Q5$.

8.3.5 Bases avec index et performance des requêtes

Cette section est consacrée à l'expérience d'exécution des requêtes sur les bases avec index. Pour cela, nous avons créé des copies des neuf bases de documents bSi' et nous avons créé pour chacune des index. Le choix des index est spécifique à chaque base car leur structuration est différente. Pour le choix des index nous avons également pris en compte l'ensemble des requêtes étudiées. Les bases indexées sont nommées $bSi' - Ix$ par la suite.

Le Table 8.6 présente un tableau récapitulatif des index créés pour chacune des bases. Pour chaque base le tableau présente :

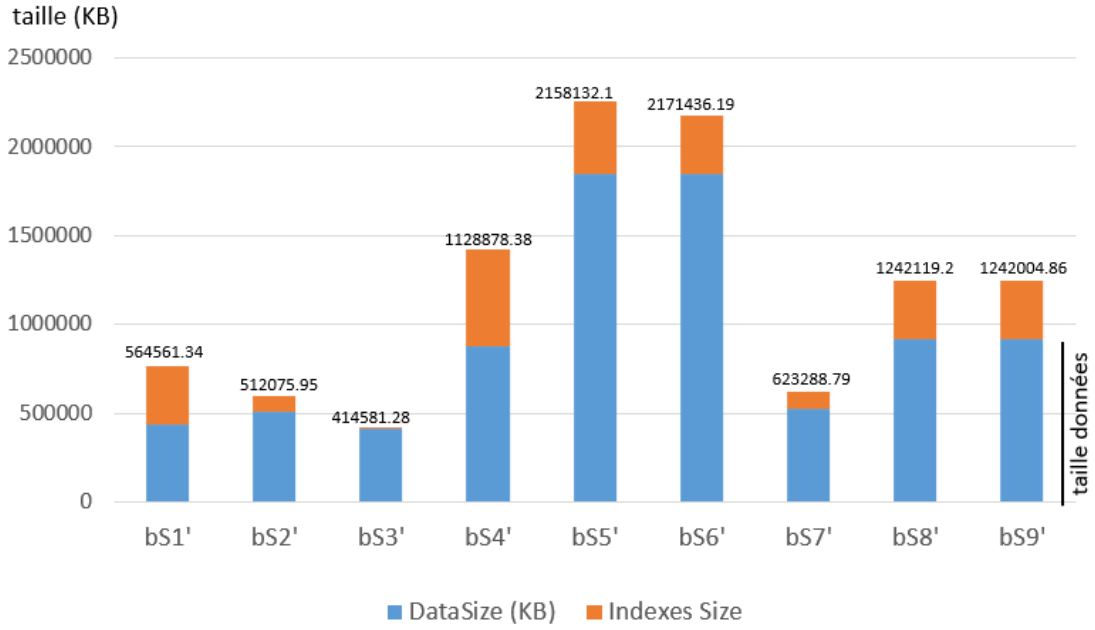


FIGURE 8.19: Taille des bases de documents avec des index - BSi'-Ix

— le nom de la collection indexée,
— l'attribut indexé avec le chemin d'imbrication si nécessaire,
— le type d'index qui est "single field" ou "multikey" (pour les array),
— le niveau d'imbrication de l'attribut indexé,
— la taille mémoire occupée par l'index.

Pour le choix des index créés pour chaque base nous avons pris en compte l'implémentation spécifique des requêtes pour cette base. Notons que nous avons créé des indexes sur tous les identifiants (par exemple, l'id de l'entreprise) et sur le salaire. Ces informations se retrouvent à des niveaux d'imbrication différents selon les bases.

La Figure 8.19 illustre la taille des bases indexées. La taille totale est illustrée avec deux bandes. La bande inférieure bleue correspond à la taille des données alors que la bande supérieure marron représente la taille des indexes.

La taille des index sur la base $bS3' - Ix$ est particulièrement petite comparée aux autres bases. Cette base à une imbrication successive de tableaux et l'index du salaire est de type multikey à cause des tableaux. MongoDB a des optimisations particulières pour les index sur des attributs imbriqués dans des tableaux. Le gain est très clair ici. Par rapport à $bS3' - Ix$, les index sur salaire dans les bases $bS2' - Ix$ et $bS7' - Ix$, par exemple, occupent bien plus de place. Cela est dû au fait

que les employés sont regroupés au niveau 1 et à l'intérieur d'un document et non d'un tableau.

Base	Collection	Index	Type	Level	Size KB
bS1'- Ix	employees	salary	Single field	1er	98279
	employees	id_dept	Single field	1er	98279
	departments	id_comp	Single field	1er	63
	companies	id_company	Single field	1er	7
bS2'- Ix	departments	employees.salary	Multikey	2e	98279
	departments	id_comp	Single field	1er	63
	companies	id_company	Single field	1er	7
bS3'- Ix	companies	departments.employees.salary	Multikey	3e	4974
	companies	id_company	Single field	1er	7
	companies	departments.name	Single field	1er	63
bS4'- Ix	cdes	id_comp	Single field	1er	98279
	employees	id_employee	Single field	1er	98279
	employees	salary	Single field	1er	98279
	departments	id_department	Single field	1er	63
	companies	id_company	Single field	1er	7
bS5'- Ix	employees	salary	Single field	1er	98279
	employees	department.company.id_company	Single field	3e	98279
	employees	department.id_department	Single field	2e	98279
bS6'- Ix	employees	salary	Single field	1er	98279
	employees	department.company.id_company	Single field	3e	98279
	departments	company.id_company	Single field	2e	63
bS7'- Ix	departments	employees.salary	Multikey	2e	98279
	departments	company.id_company // id_comp	Single field	1er	167
bS8'- Ix	employees	salary	Single field	1er	98279
	employees	id_dept	Single field	1er	103717
	departments	company.id_company	Single field	2e	63
bS9'- Ix	employees	salary	Single field	1er	98279
	employees	id_dept	Single field	1er	103717
	companies	departments.name	Single field	1er	7

TABLE 8.6: Liste des index des 9 bases $bSi' - Ix$

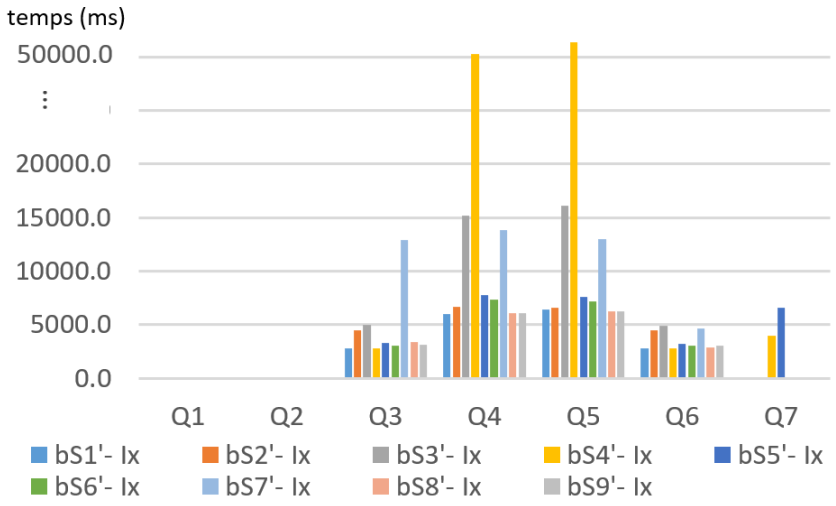


FIGURE 8.20: Médiane du temps d'exécution des requêtes sur les bases avec index

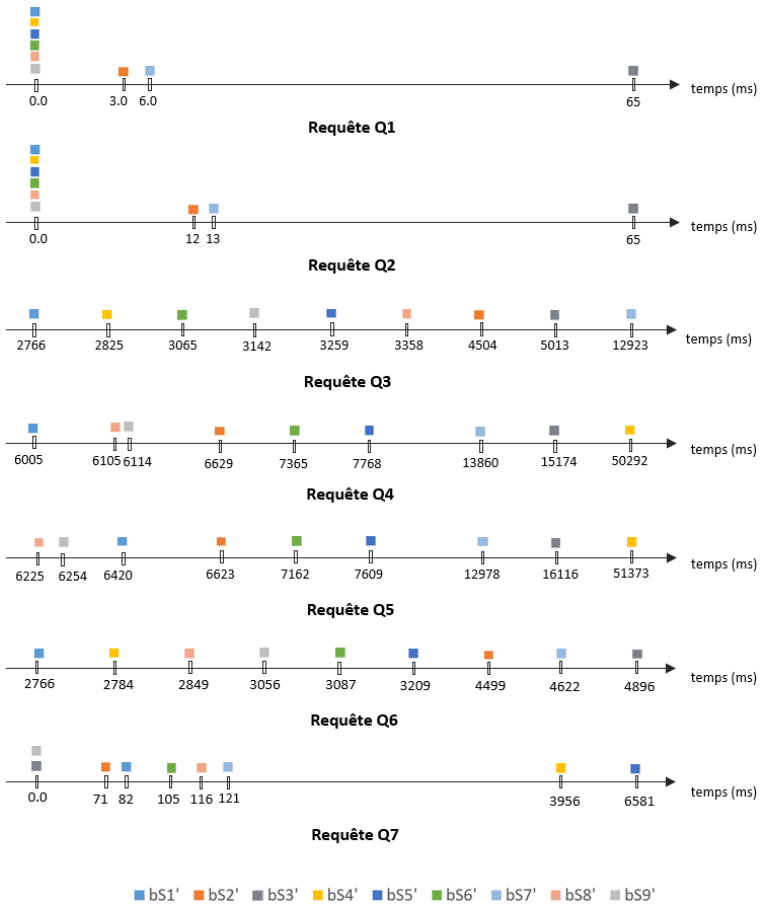


FIGURE 8.21: Positions relatives des bases avec index par rapport à l'exécution de chaque requête

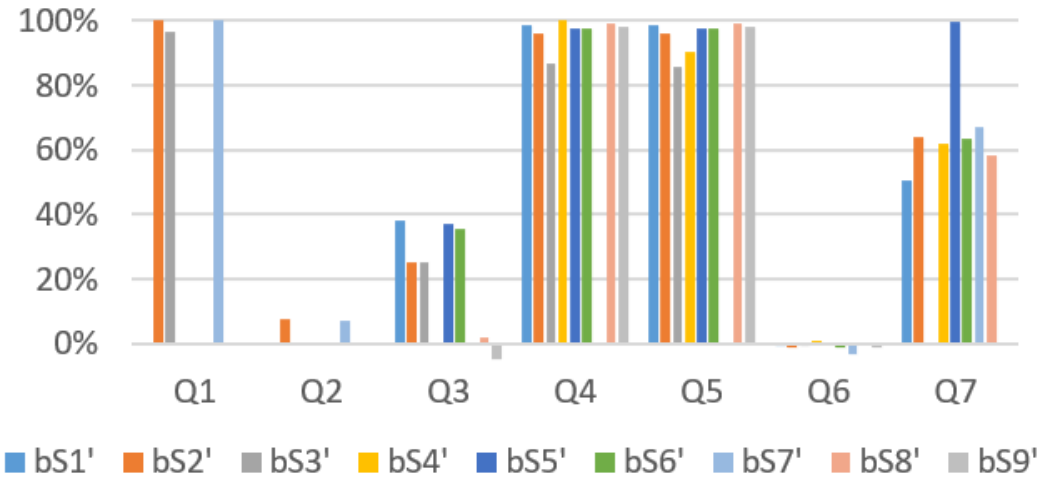


FIGURE 8.22: Amélioration de performance obtenue à l'aide des indexes

La taille des index utilisés dans la base $bS9' - Ix$ représente une taille moyenne par rapport aux tailles des index dans les autres bases. Ceci constitue un autre aspect qui peut jouer en faveur de cette base.

Les Figures 8.20 et 8.21 fournissent une synthèse des résultats des exécutions des sept mêmes requêtes de notre étude sur les neuf bases avec index $bSi' - Ix$. La Figure 8.22 montre l'amélioration obtenue à l'aide des index par rapport aux bases sans index. Une amélioration de X% signifie que le temps de réponse d'une requête est réduit de X%.

Comme attendu, les index améliorent les performances. En particulier, l'amélioration de $Q4$ et $Q5$ est élevée sur toutes les bases. Le bénéfice des index sur la base $bS3' - Ix$ est moins important que pour les autres bases. $bS3' - Ix$ se retrouve ici avec les performances les moins bonnes.

Notons que les index améliorent les performances de $Q7$ sur toutes les bases. Néanmoins, l'amélioration obtenue sur toutes les bases ne suffit pas pour dépasser les bons résultats sur $bS3' - Ix$ et sur $bS9' - Ix$. Cela montre, comme dans notre expérience initiale, que même si les index sont efficaces, il y a des cas où leur usage est limité si la structuration des données est inadaptée.

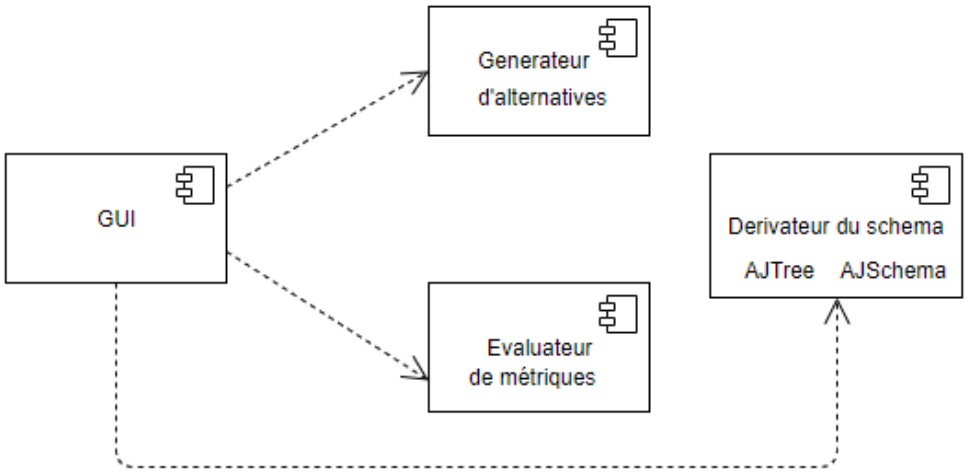


FIGURE 8.23: Architecture du prototype *ScorusTool*

8.4 Mise en œuvre de *ScorusTool*

Dans le cadre de cette thèse, nous avons développé le prototype nommé *ScorusTool* afin d'implémenter nos contributions concernant la génération de schémas, l'évaluation de métriques et la manipulation de représentations AJSchéma et AJTree.

Cette section est consacrée aux aspects techniques de l'implémentation de *ScorusTool*. Nous présentons l'architecture et ses principaux composants. Ensuite, nous présentons deux cas de validation du prototype.

8.4.1 Architecture et principaux choix

Nous avons développé des composants pour chaque fonctionnalité de notre contribution afin de faciliter leur utilisation indépendante ou enchaînés dans un processus. Nous fournissons également une interface graphique pour la présentation des résultats et l'interaction avec l'utilisateur.

L'architecture de *ScorusTool* est introduite dans la Figure 8.23 et le diagramme de classes correspondant dans la Figure 8.24. Cette architecture est composée notamment des quatre composants suivants :

— *Générateur d'alternatives*, qui, d'un modèle UML, génère différentes alternatives de structurations orientées document

— *Dérivateur de schémas*, qui traduit une alternative de structuration dans un AJTree et dans un AJSchéma

— *Evaluateur de métriques*, qui évalue les métriques structurelles sur un schéma

— *GUI*, qui fournit une interface graphique pour l'interaction avec l'utilisateur et la présentation des résultats.

Nous avons développé *ScorusTool* avec l'environnement de programmation Eclipse. Nous avons utilisé notamment le langage de programmation JAVA, le Framework EMF [88, 89] et le langage exécutable FAMILIAR [87].

Dans la suite nous présentons une vue globale de chaque composant.

8.4.2 Composant « *Générateur d'alternatives* »

Le « *Générateur d'alternatives* » est en charge de récupérer un modèle UML fourni par l'utilisateur, de l'analyser et de créer un modèle de caractéristiques afin de fournir un ensemble des alternatives de structuration. Ceci concerne les propositions présentées dans le chapitre 6 sections 6.4 et 6.5 concernant la création d'un modèle de caractéristiques pour alternatives de structuration orientés document.

Nous avons utilisé le Framework *EMF* [88, 89] ainsi que la spécification du méta-modèle d'UML [90] afin d'analyser et d'extraire l'information du modèle UML. Nous avons utilisé Papyrus pour faciliter la création du modèle UML.

Nous avons utilisé le langage *FAMILIAR* pour la création du modèle de caractéristiques. *FAMILIAR* permet de manipuler et raisonner sur les variantes d'un modèle de caractéristiques. L'algorithme *AMISS*, présenté dans la section 6.5, a été implémenté afin de créer le modèle fms à l'aide de *FAMILIAR*. L'opération d'insertion d'une branche dans un modèle fms, en mode OR, a été implémentée car *FAMILIAR* ne la fournit pas.

Chacune des configurations, correspondant aux alternatives de structuration, est fournie sous la forme d'un ensemble de noms des caractéristiques sélectionnées du modèle de caractéristiques.

8.4.3 Composant « *Dérivateur de schémas* »

Le « *Dérivateur de schémas* » est en charge de traduire une configuration du modèle fms, correspondant à une alternative de structuration d'un schéma, sous la forme d'une représentation AJTree et d'une représentation AJSchéma. Ceci concerne les propositions présentées dans le chapitre 6 section 6.6 concernant la dérivation d'une alternative de structuration.

Ce composant utilise notamment des bibliothèques de manipulation de graphes en Java tant dans la construction de la représentation arborescente AJTree que dans la construction de l'AJSchéma.

Pour la construction de l'AJTree, nous avons implémenté l'algorithme *ADJusT*, présenté dans la section 6.6.1. Cet algorithme, utilise l'ensemble des noms des caractéristiques de la configuration et génère le graphe correspondant à l'AJTree.

Concernant l'AJSchéma, l'implémentation de l'algorithme *ADApt*, présenté dans la section 6.6.2, nous permet de le créer. Cet algorithme utilise l'information fournie par l'AJTree et construit l'AJSchéma.

8.4.4 Composant « *Evaluateur de métriques* »

L'*Evaluateur de métriques* est en charge d'évaluer les métriques structurelles d'un schéma en utilisant leur représentation arborescente AJTree. Ces métriques ont été présentées dans le chapitre 7.

Des bibliothèques de manipulation de graphes en Java sont utilisées pour traiter l'AJTree. Nous utilisons également le Framework EMF et la spécification du méta-modèle UML pour l'extraction d'informations du modèle UML quand c'est requis.

Les métriques à évaluer sont identifiées en analysant les types fournis par le modèle UML. Par exemple, si un modèle UML fournit les types $te1$ et $te2$, alors la métrique $colExistence(type)$ sera évaluée deux fois, une pour chaque type.

Nous avons implémenté une fonction par métrique. Il s'agit d'algorithmes de parcours d'arbres afin de calculer les métriques structurelles selon les formules présentées dans le chapitre 7.

8.4.5 Composant « *GUI* »

Ce composant fournit l'interface graphique pour l'interaction avec l'utilisateur et la présentation des résultats (cf. Figure 8.25). Cette interface montre les configurations fournies par le modèle de caractéristiques fms correspondant aux alternatives de structuration. Pour chaque alternative de structuration, l'interface présente l'AJTree et l'AJSchéma correspondant ainsi que les valeurs des métriques structurelles.

Le composant utilise des librairies de la bibliothèque Swing de Java. Le langage *FAMILIAR* est également utilisé afin d'interagir entre les composants « *Générateur d'alternatives* » et « *Evaluateur de métriques* ».

Depuis l'interface, il est possible de choisir si la génération et l'évaluation sont exécutées indépendamment ou dans le cadre d'un processus. Il est également possible de sélectionner un sous-ensemble de métriques à évaluer.

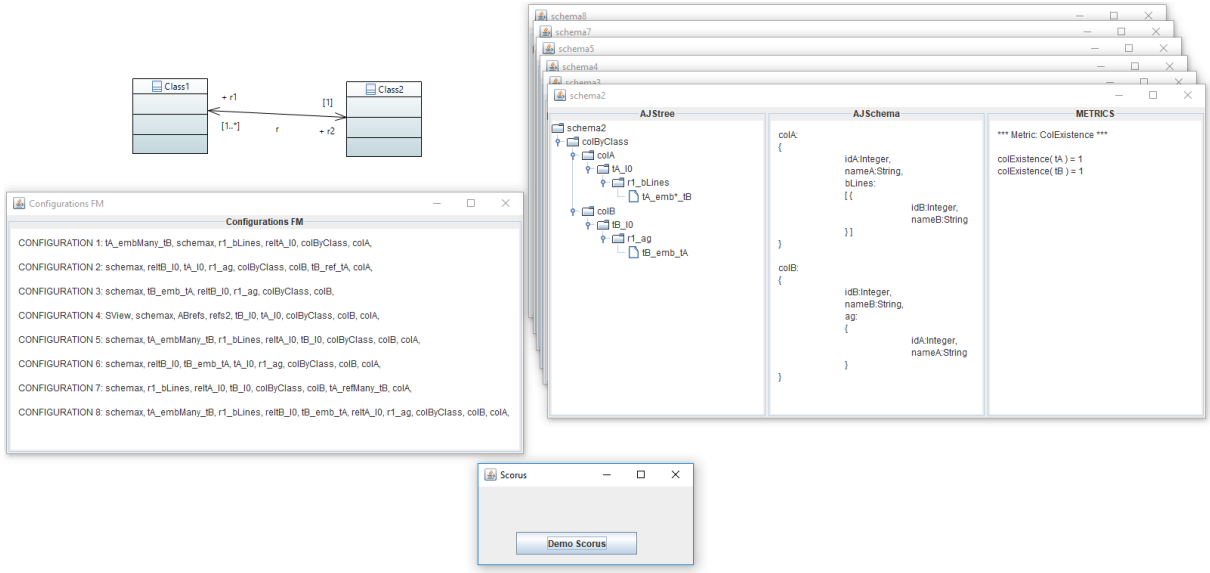


FIGURE 8.25: GUI *ScorusTool*

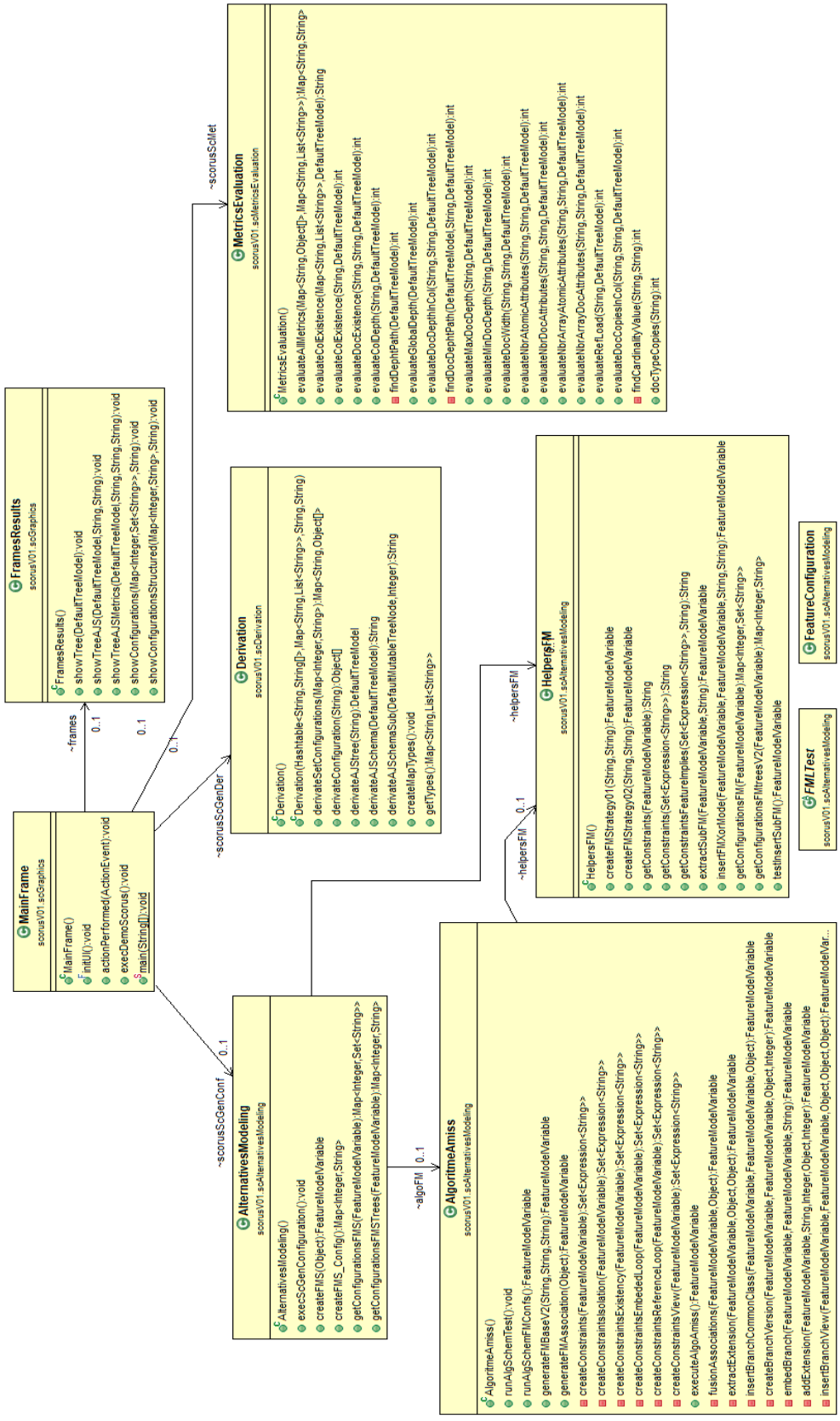


FIGURE 8.24: Diagramme de classes de l'implémentation *ScorusTool*

8.5 Conclusions

Dans ce chapitre nous avons présenté des actions de validation de nos recherches ainsi que le prototype *ScorusTool* mis en œuvre. En effet, nous avons développé les modules de génération d'alternatives de structuration, les outils nécessaires à la manipulation des AJTree et des AJSchéma et l'évaluateur automatique des métriques.

Concernant la validation de nos recherches nous avons présenté un scénario suivant l'approche préconisée dans SCORUS. Cela nous a permis de prouver les concepts et constater la faisabilité et l'intérêt de nos propositions. Nous avons mené une analyse en utilisant des métriques sur plusieurs variantes d'AJSchéma et certains critères et priorités applicatifs. L'analyse a permis d'écarter certains AJSchémas et d'en mettre en avant d'autres. Ces résultats d'analyse structurelle ont été comparés et, sont bien en phase, avec les résultats d'expériences d'évaluation de performances sur des bases opérationnelles. Nos expérimentations ont été réalisées sur des bases MongoDB contenant une taille conséquente de documents. La création des bases et la programmation des requêtes a requis un effort significatif. Il est important de noter, qu'au contraire, l'analyse structurelle est faite à "faible coût" dans le sens des efforts à faire par le développeur. Aidé par l'outil de génération d'AJSchémas et d'évaluation de métriques, un développeur pourra facilement et rapidement apprécier plusieurs alternatives et faire un choix éclairé.

Lors de l'analyse structurelle, nous avons utilisé plusieurs critères et effectué une analyse multi-critères. L'expression des critères et leur pondération pour l'analyse multi-critères n'est pas toujours facile. Les critères à considérer et leur poids dépendent du contexte applicatif mais peuvent aussi correspondre à de bonnes pratiques préconisées pour le développement ou à une priorité générale. Par exemple, adopter des structurations très "compactes" pour limiter l'empreinte mémoire lorsque des données seront peu utilisées. Ou, en une donnant priorité à la qualité logicielle, ou en privilégient les AJSchémas les plus "lisibles". Sachant que les critères peuvent diverger et évoluer, l'utilisation des métriques et des critères pour un choix de AJSchéma peut aider dans un processus continu de monitoring (voir de *tuning*) de la base. Une telle approche peut conduire à des évolutions de la structuration ou à la création de copies des données avec des structures différentes. Pendant un certain temps, une base pourrait avoir, pour les mêmes données, une copie structurée selon l'AJSchéma Sj et une autre copie selon l'AJSchéma Sk.

La proposition d'outils d'aide à l'expression et la formalisation des critères nous parait une perspective intéressante de ce travail. La suite des travaux inclut également un système de recommandations plus complet pour faciliter la prise en compte des critères en utilisant les métriques, les requêtes fréquentes et autres préférences fonctionnelles ou non fonctionnelles des utilisateurs potentiels.

“L’espoir est la chose avec des plumes,
Qui se perche dans l’âme,
Et chante l’air sans les mots,
Et ne s’arrête jamais - du tout - ...”

Emily Dickinson

9

Conclusions et perspectives

.

9.1 Conclusions

Ces travaux de thèse ont été motivés par l'utilisation croissante des systèmes NoSQL et plus précisément par leur impact sur les systèmes d'information actuels et à venir. Après nous être intéressés au développement d'applications en utilisant Cassandra, Neo4J et MongoDB, nous nous sommes focalisés sur ce dernier. En effet, MongoDB est très largement utilisé et le sera probablement pendant longtemps. Cela amène la question de la qualité et la maintenabilité des applications qui l'utilisent.

Au regard des approches traditionnelles de développement d'applications utilisant des SGBD, les applications utilisant des systèmes orientés documents sont différentes à plusieurs égards. Les plus notables ayant motivé nos travaux sont l'absence de schéma conceptuel, la grande flexibilité de représentation de données aux structures complexes et la manière d'accéder aux données. Nous avons pu constater cela de manière empirique par les expérimentations que nous avons réalisées avec MongoDB. Nous avons créé 30 bases de documents avec deux jeux de données de taille différentes, et programmé plus de 60 requêtes adaptées aux diverses structures des documents des bases. Ces expérimentations ont permis de mieux cerner les avantages et les inconvénients pour le développeur et de clarifier les enjeux en ce qui concerne la structuration des bases et les programmes qui l'utilisent.

Au début de nos travaux certaines démarches expérimentales et listes de bonnes pratiques commençaient à émerger sans pour autant arriver à des propositions consolidées. Dans cette thèse, nous préconisons une démarche de modélisation et des éléments objectifs d'analyse. Cela est rendu possible à faible coût pour les utilisateurs ce qui joue en faveur de son adoption. Dans des publications récentes, d'autres auteurs font également des propositions dans ce sens.

De notre point de vue, les contributions conceptuelles les plus importantes de cette thèse concernent la génération d'alternatives de structuration et la proposition de métriques structurelles. Nous avons mis en œuvre les algorithmes proposés et réalisé plusieurs expérimentations qui ont largement contribué à l'identification des problèmes de recherche, leur solution et à la validation des propositions.

Génération d'alternatives de structuration de données
En ce qui concerne la génération d'alternatives de structuration de données, nos conclusions sont très positives. La possibilité de visualiser rapidement les alternatives facilite l'analyse. D'un point de vue démarche et algorithmique pour la mise en œuvre du générateur, nos choix ont été guidés par une approche en génie logiciel. L'utilisation des modèles de caractéristiques, bien qu'un peu complexe à prendre en main, s'est avéré pertinente. Cette approche est, à notre connaissance,

originale pour la conception de bases de données. Elle permet d'explorer et de gérer un grand nombre d'alternatives de structuration et ouvre des possibilités intéressantes pour la prise en compte de contraintes.

Dans la conception de l'algorithme de génération AMISS, nous avons effectué des choix importants pour limiter la variabilité et le nombre de solutions produites. Ce dernier reste néanmoins élevé. Nous avons notamment produit des AJSchémas avec des collections dont la structure des documents est unique. Nous avons travaillé avec l'idée que ces structures peuvent jouer le rôle de guide sachant que les systèmes de stockage n'effectuent pas de vérification de types. Ceci autorise bien sûr qu'en pratique les documents puissent être créés avec une structure qui diffère. L'algorithme AMISS implémente, à l'aide de contraintes, la génération de structures avec plusieurs niveaux d'imbrication de documents et diverses formes de redondance. Les contraintes implantées aujourd'hui reflètent certains choix qui nous ont paru pertinents suite à nos recherches. Nous avons néanmoins opté pour une mise en œuvre qui permet de facilement modifier et étendre les contraintes de génération de variantes pour pouvoir personnaliser la stratégie selon les préférences des utilisateurs. Ces préférences pourraient conduire à réduire le nombre de variantes générées ou à faire explorer des variantes avec des patrons de structuration choisis.

Métriques structurelles
La proposition de métriques structurelles faite dans cette thèse constitue un outil formel et objectif qui peut être utilisé dans l'analyse des structures. Ces métriques mettent en évidence des caractéristiques qui peuvent avoir un impact notable sur plusieurs aspects liés aux données et aux programmes. Nos premières expériences avec les métriques confirment leur utilité et laissent penser que le développement d'outils formels plus complets serait bienvenu.

Rappelons que le positionnement de notre proposition de métriques sur les aspects structurels est à considérer comme un complément à d'autres mesures de performances opérationnelles si elles sont disponibles. Notre objectif initial était d'aider les développeurs dans la prise de décision lorsque la base de documents n'est pas encore créée et il n'y a donc pas de données statistiques de performance sur son utilisation. Le coût de création et d'exploitation d'une base de documents est important. Bien qu'intéressant pour apprécier les performances, il ne sera pas forcément possible pour les développeurs d'expérimenter avec plusieurs bases de documents avec structures alternatives dans une phase de prise de décision de la structure. L'analyse structurelle reste partielle mais peut être faite à faible coût et contribue à éclairer les choix des analystes-développeurs d'applications.

Les propositions de génération d'alternatives de structuration et l'évaluation automatique des métriques sont opérationnelles dans le prototype *ScorusTool*. Pour un développeur, il suffit de donner un modèle UML et en un clic il peut visualiser l'ensemble de variantes de structure et

leur métriques. En très peu de temps, il est possible de considérer de nombreuses alternatives qui permettraient une analyse poussée.

9.2 Perspectives

De ce travail nous identifions des perspectives à court et moyen terme. Sur le court terme nous souhaitons investir dans la finalisation et fiabilisation du prototype *ScorusTool*. Ils nous semble également important de réaliser une expérimentation avec le prototype en le proposant à des développeurs d'applications utilisant MongoDB. Une telle expérimentation permettra aussi de mieux juger l'ensemble des métriques structurelles proposées et si nécessaire envisager leur évolution. Sur le plan des métriques, nous avons commencé un travail sur les requêtes qui n'a pas été présenté ici mais qui nous paraît intéressant à explorer. Ce travail porte sur l'analyse de code des requêtes pour donner des indicateurs de la complexité des programmes nécessaires selon les cas.

Nous souhaitons également travailler sur un outil de rétro-ingénierie de bases de documents, consistant à déduire l'AJSchéma correspondant à une base MongoDB existante. Cela permettrait d'évaluer les métriques sur ces AJSchémas et contribuer à une meilleure compréhension des bases et des applications.

Concernant le générateur d'AJSchémas, une perspective que nous souhaitons explorer porte sur l'optimisation et la personnalisation de l'algorithme de génération de variantes. L'algorithme proposé dans cette thèse est principalement inspiré d'expériences d'utilisation de bases de documents. Cet algorithme pourrait être enrichi en permettant à l'utilisateur de donner des préférences ou contraintes structurelles à respecter par les AJSchémas créés. Ces contraintes pourraient être, par exemple, des métriques qui joueraient le rôle de bornes tel qu'un nombre maximum de copies ou de niveaux d'imbrication de documents.

Sur un plan plus global, une autre perspective importante pour ce travail est une meilleure prise en compte des collections avec des documents hétérogènes. Cette hétérogénéité peut se situer à plusieurs niveaux tels que le type des propriétés qui diffère d'un document à l'autre ou le caractère optionnel des propriétés d'un document. La prise en compte des collections hétérogènes peut être reflétée sur divers aspects, notamment dans la génération d'alternatives de structuration. Nous considérons qu'une option intéressante à explorer est l'utilisation d'annotations sur le modèle UML afin de permettre l'expression du niveau de liberté souhaité pour les instances d'une classe. Une classe avec un typage fort donne lieu à des structures homogènes où les instances sont toutes attendues avec les mêmes attributs. Au contraire, si un typage faible est envisagé, ceci peut être

précisé avec des annotations indiquant le caractère obligatoire ou optionnel d'un attribut et si son type est strict ou non. Ces indications seront reflétées dans les AJSchéma mis à disposition de l'utilisateur de manière à servir de guide pour la création des bases.

L'extension pour une prise en compte de l'hétérogénéité des structures nécessite aussi une extension de la définition des métriques et de l'analyse pour sélectionner une structuration pour un cas précis. Il serait notamment intéressant de mesurer les niveaux d'hétérogénéité, de les comprendre et d'estimer la difficulté et le surcoût engendré par l'utilisation de collections hétérogènes.

Pour finir, nos perspectives à plus long terme concernent un système de recommandation des structures. Il s'agit d'enrichir SCORUS avec l'automatisation de la phase d'analyse mais aussi chercher à assister le développeur plus largement dans ces choix de modélisation. Il pourrait l'assister notamment dans l'explicitation des critères prioritaires en se basant sur l'expression de requêtes dans des langages de haut niveau indépendants de la structure de la base. Nous envisageons également un module d'aide à l'administrateur qui permettrait de suivre leur évolution lorsque la base est déjà créée et en exploitation. On disposerait alors d'une démarche d'ingénierie outillée dans ce contexte de bases de documents.

Bibliographie

[1] 451Research. *Data Platforms Map*. https://451research.com/state-of-the-database-landscape. Accessed: 2018-03-26.

[2] A. Davoudian, L. Chen et M. Liu. “A Survey on NoSQL Stores”. Dans : *Journal ACM Computing Surveys (CSUR)*. Vol. 51, n° 2, p. 40. ACM, 2018.

[3] M. Chen, S. Mao et Y. Liu. “Big data: A survey”. Dans : *Mobile networks and applications, Journal*. Vol. 19, n° 2, p. 171–209. Springer, 2014.

[4] P. J. Sadalage et M. Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Livre. Pearson Education, 2012.

[5] N. Leavitt. “Will NoSQL Databases Live Up to Their Promise?”. Dans : *Journal Computer*. Vol. 43, n° 2, p. 12–14. IEEE, 2010.

[6] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca et R. Torlone. “The relational model is dead, SQL is dead, and I don’t feel so good myself”. Dans : *ACM SIGMOD Record*. Vol. 42, n° 2, p. 64–68. ACM, 2013.

[7] E. Gallinucci, M. Golfarelli et S. Rizzi. “Schema profiling of document-oriented databases”. Dans : *Information Systems Journal*. Vol. 75, p. 13–25. Elsevier, 2018.

[8] P. Atzeni, F. Bugiotti, L. Cabibbo et R. Torlone. “Data modeling in the NoSQL world”. Dans : *Computer Standards & Interfaces Journal*. Elsevier, 2016.

[9] E. F. Codd. “A relational model of data for large shared data banks”. Dans : *Communications of the ACM*. Vol. 13, n° 6, p. 377–387. 1970.

[10] A. Makinouchi. “A consideration on normal form of not-necessarily-normalized relation in the relational data model”. Dans : *Proceedings of the International conference on Very Large Data Base (VLDB)*. Vol. 1977, p. 447–453. 1977.

[11] R. Agrawal et N. H. Gehani. “Ode (object database and environment): the language and the data model”. Dans : *ACM SIGMOD Record*. Vol. 18, p. 36–45. ACM, 1989.

[12] J. Han, E. Haihong, G. Le et J. Du. “Survey on NoSQL database”. Dans : *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, p. 363–366. 2011.

[13] MongoDB. https://www.mongodb.com. Accessed: 2018-09-25.

[14] neo4J. https://neo4j.com. Accessed: 2018-09-25.
[15] cassandra. cassandra.apache.org. Accessed: 2018-09-25.
[16] A. Kanade, A. Gopal et S. Kanade. "A study of normalization and embedding in MongoDB". Dans : *International Advance Computing Conference (IACC)*, p. 416–421. 2014.
[17] DB-Engines. https://db-engines.com. Accessed: 2018-09-25.
[18] K. C. Kang, J. Lee et P. Donohoe. "Feature-oriented product line engineering". Dans : *IEEE Software Magazine*. Vol. 19, nº 4, p. 58–65. IEEE, 2002.
[19] H. Gomaa. *Designing software product lines with UML*. Livre. IEEE, 2005.
[20] P. Gómez, R. Casallas et C. Roncancio. "Data schema does matter, even in NoSQL systems!". Dans : *Research Challenges in Information Science (RCIS) Tenth International Conference on*, p. 1–6. IEEE, 2016.
[21] P. Gómez, C. Roncancio et R. Casallas. "Métriques structurelles pour l'analyse de bases orientées documents". Dans : *Actes du XXXVIème Congrès INFORSID, May 28 - 31, 2018*.
[22] P. Gómez, C. Roncancio et R. Casallas. "Towards quality analysis for document oriented bases". Dans : *International Conference on Conceptual Modeling (ER) October 22-25, 2018*. Springer,
[23] D. Wood. "Standard generalized markup language: Mathematical and philosophical issues". Dans : *Computer Science Today*. Springer, 1995, p. 344–365.
[24] R. G. G. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland et D. Wade. *The object database standard: ODMG 2.0*. Livre. Morgan Kaufmann Publishers Los Altos, CA, 1997.
[25] Y. Papakonstantinou, H. Garcia-Molina et J. Widom. "Object exchange across heterogeneous information sources". Dans : *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, p. 251–260. IEEE, 1995.
[26] S. Abiteboul. "Querying semi-structured data". Dans : *International Conference on Database Theory*, p. 1–18. Springer, 1997.
[27] Amazon. *DynamoDB*. https://aws.amazon.com/fr/dynamodb/. Accessed: 2018-10-09.
[28] Redis. https://redis.io/. Accessed: 2018-10-09.
[29] Voldemort. https://www.project-voldemort.com/voldemort/. Accessed: 2018-10-09.
[30] google. *Bigtable*. https://cloud.google.com/bigtable/. Accessed: 2018-10-09.
[31] Apache. *HBase*. http://hbase.apache.org/. Accessed: 2018-10-09.
[32] Titan. http://titan.thinkaurelius.com/. Accessed: 2018-10-09.
[33] OrientDB. https://orientdb.com/. Accessed: 2018-10-09.
[34] ArangoDB. https://www.arangodb.com/. Accessed: 2018-10-09.
[35] T. Bray. "The javascript object notation (json) data interchange format". Rapp. tech. 2017.
[36] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler et F. Yergeau. "Extensible Markup Language (XML)." Dans : *World Wide Web Journal*. Vol. 2, nº 4, p. 27–66. 1997.
[37] Apache. *CouchDB*. http://couchdb.apache.org/. Accessed: 2018-10-09.
[38] Apache. *AsterixDB*. https://asterixdb.apache.org/. Accessed: 2018-10-09.

[39] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz et al. "AsterixDB: A scalable, open source BDMS". Dans : *Proceedings of the International conference on Very Large Data Base (VLDB)*. Vol. 7, n° 14, p. 1905–1916. VLDB Endowment, 2014.

[40] R. Grover et M. J. Carey. "Data Ingestion in AsterixDB." Dans : *International Conference on Extending Database Technology (EDBT)*, p. 605–616. OpenProceedings, 2015.

[41] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler et C. Li. "Storage management in AsterixDB". Dans : *Proceedings of the International conference on Very Large Data Base (VLDB)*. Vol. 7, n° 10, p. 841–852. VLDB Endowment, 2014.

[42] V. Borkar, Y. Bu, E. P. Carman Jr, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey et V. J. Tsotras. "Algebricks: a data model-agnostic compiler backend for Big Data languages". Dans : *Proceedings of the Sixth ACM Symposium on Cloud Computing*, p. 422–433. 2015.

[43] R. Copeland. *MongoDB Applied Design Patterns*. Livre. Oreilly, 2013.

[44] *RDBMS to MongoDB Migration Guide*. White Paper. MongoDB, nov. 2017., adresse : `https://www.mongodb.com/collateral/rdbms-mongodb-migration-guide`.

[45] G. Zhao, Q. Lin, L. Li et Z. Li. "Schema Conversion Model of SQL Database to NoSQL". Dans : *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, p. 355–362. Nov. 2014.

[46] M. J. Mior, K. Salem, A. Aboulnaga et R. Liu. "NoSE: Schema design for NoSQL applications". Dans : *IEEE Transactions on Knowledge and Data Engineering*. Vol. 29, n° 10, p. 2275–2289. IEEE, 2017.

[47] S. Lombardo, E. D. Nitto et D. Ardagna. "Issues in Handling Complex Data Structures with NoSQL Databases". Dans : *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, p. 443–448. 2012.

[48] T. Vajk, P. Feher, K. Fekete et H. Charaf. "Denormalizing data into schema-free databases". Dans : *Cognitive Infocommunications (CogInfoCom), 4th International Conference on*, p. 747–752. IEEE, 2013.

[49] M. J. Mior. "Automated schema design for NoSQL databases". Dans : *Proceedings of the 2014 SIGMOD PhD symposium*, p. 41–45. ACM, 2014.

[50] T. Vajk, L. Deák, K. Fekete et G. Mezei. "Automatic NoSQL schema development: A case study". Dans : *Journal of Artificial Intelligence and Applications*, p. 656–663. Actapress, 2013.

[51] J. B. Warmer et A. G. Kleppe. "The object constraint language: Precise modeling with uml (addison-wesley object technology series)". Addison-Wesley Longman Publishing Co., Inc., 1998.

[52] F. Abdelhedi, A. A. Brahim, F. Atigui et G. Zurfluh. "MDA-Based Approach for NoSQL Databases Modelling". Dans : *International Conference on Big Data Analytics and Knowledge Discovery*, p. 88–102. Springer, 2017.

[53] A. G. Kleppe, J. Warmer, W. Bast et M. Explained. *The model driven architecture: practice and promise*. 2003.

[54] F. Bugiotti, L. Cabibbo, P. Atzeni et R. Torlone. “Database design for NoSQL systems”. Dans : *International Conference on Conceptual Modeling*, p. 223–231. Springer, 2014.

[55] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou et C. Wangz. “Schema management for document stores”. Dans : *Proceedings of the International conference on Very Large Data Base (VLDB)*. Vol. 8, n° 9, p. 922–933. 2015.

[56] D. S. Ruiz, S. F. Morales et J. G. Molina. “Inferring Versioned Schemas from NoSQL Databases and Its Applications”. Dans : *Conceptual Modeling*. Springer, 2015, p. 467–480.

[57] D. C. Schmidt. “Model-driven engineering”. Dans : *Journal Computer*. Vol. 39, n° 2, p. 25. IEEE, Citeseer, 2006.

[58] M. Klettke, U. Störl et S. Scherzinger. “Schema extraction and structural outlier detection for JSON-based NoSQL data stores”. Dans : *Datenbanksysteme für Business, Technologie und Web (BTW)*. Gesellschaft für Informatik eV, 2015.

[59] MongoDBCompass. https://docs.mongodb.com/compass/master/. Accessed: 2018-02-12.

[60] jsonschema.net. https://jackwootton.github.io/json-schema/. Accessed: 2018-10-03.

[61] jsonSchema. *JSON Schema*. http://json-schema.org/. Accessed: 2018-03-26.

[62] O. Herden. “Measuring Quality of Database Schemas by Reviewing–Concept, Criteria and Tool”. Dans : *Oldenburg Research and Development Institute for Computer Science Tools and Systems, Escherweg*. Vol. 2, p. 26 121. 2001.

[63] R. Cattell. “Scalable SQL and NoSQL data stores”. Dans : *ACM SIGMOD Record*. Vol. 39, n° 4, p. 12–27. ACM, 2011.

[64] A. Boicea, F. Radulescu et L. I. Agapin. “MongoDB vs Oracle–Database Comparison”. Dans : *2012 Third International Conference on Emerging Intelligent Data and Web Technologies*, p. 330–335. IEEE, 2012.

[65] W.-C. Chung, H.-P. Lin, S.-C. Chen, M.-F. Jiang et Y.-C. Chung. “JackHare: a framework for SQL to NoSQL translation using MapReduce”. Dans : *Automated Software Engineering, Journal*. Vol. 21, n° 4, p. 489–508. 2014.

[66] R. Angles. “A comparison of current graph database models”. Dans : *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, p. 171–177. IEEE, 2012.

[67] R. Angles et C. Gutierrez. “Survey of graph database models”. Dans : *Journal ACM Computing Surveys (CSUR)*. Vol. 40, n° 1, p. 1. ACM, 2008.

[68] P. J. Carrington, J. Scott et S. Wasserman. *Models and methods in social network analysis*. Livre. Cambridge university press, 2005. Vol. 28.

[69] J. A. Bondy, U. S. R. Murty et al. *Graph theory with applications*. Livre. Citeseer, 1976. Vol. 290.

[70] D. B. West et al. *Introduction to graph theory. Livre*. Prentice hall Upper Saddle River, 2001. Vol. 2.

[71] B. Gallagher. “Matching structure and semantics: A survey on graph-based pattern matching”. Dans : *American Association for Artificial Intelligence, Fall Symposium (AAAI FS)*. Vol. 6, p. 45–53. 2006.

[72] M. Klettke, L. Schneider et A. Heuer. “Metrics for XML document collections”. Dans : *International Conference on Extending Database Technology*, p. 15–28. Springer, 2002.

[73] M. Pusnik, M. Heričko, Z. Budimac et B. Šumak. “XML Schema metrics for quality evaluation”. Dans : *Computer science and information systems, Journal*. Vol. 11, n° 4, p. 1271–1289. 2014.

[74] N. Fenton et J. Bieman. *Software metrics: a rigorous and practical approach. Livre*. CRC Press, 2014.

[75] A. L. Timóteo, A. Álvaro, E. S. De Almeida et S. R. de Lemos Meira. *Software Metrics: A Survey*. Citeseer. 2008.

[76] T. J. McCabe. “A complexity measure”. Dans : *IEEE Transactions on software Engineering*. N° 4, p. 308–320. IEEE, 1976.

[77] S. R. Chidamber et C. F. Kemerer. *Towards a metrics suite for object oriented design*. 11. Livre. ACM, 1991. Vol. 26.

[78] W. Li et S. Henry. “Object-oriented metrics that predict maintainability”. Dans : *Journal of systems and software*. Vol. 23, n° 2, p. 111–122. Elsevier, 1993.

[79] Sonarqube. https://www.sonarqube.org/. Accessed: 2018-10-09.

[80] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak et A. S. Peterson. “Feature-oriented domain analysis (FODA) feasibility study”. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. Rapp. tech. 1990.

[81] K. Czarnecki, U. W. Eisenecker et K. Czarnecki. *Generative programming: methods, tools, and applications*. Livre. Addison Wesley Reading, 2000. Vol. 16.

[82] D. Benavides, S. Segura et A. Ruiz-Cortés. “Automated analysis of feature models 20 years later: A literature review”. Dans : *Information Systems Journal*. Vol. 35, n° 6, p. 615–636. Elsevier, 2010.

[83] M. Mendonca, M. Branco et D. Cowan. “SPLOT: software product lines online tools”. Dans : *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, p. 761–762. ACM, 2009.

[84] M. Acher, P. Collet, P. Lahire et R. France. “Comparing approaches to implement feature model composition”. Dans : *European Conference on Modelling Foundations and Applications*, p. 3–19. Springer, 2010.

[85] M. Acher. “Managing, multiple feature models: foundations, languages and applications”. Thèse de doct. Nice. 2011.

[86] J. Chavarriaga. “Using multiple Feature Models of Domains and Regulations to develop Configuration Systems”. Thèse de doct. Vrije Universiteit Brussel. 2017.

[87] M. Acher, P. Collet, P. Lahire et R. B. France. “FAMILIAR: A domain-specific language for large scale management of feature models”. Dans : *Science of Computer Programming (SCP) , Journal*. Vol. 78, n° 6, p. 657–681. Elsevier, 2013.

[88] D. Steinberg, F. Budinsky, E. Merks et M. Paternostro. *EMF: eclipse modeling framework*. Livre. Pearson Education, 2008.

[89] Eclipse. *Eclipse Modeling Framework Project (EMF)*. http://www.eclipse.org/modeling/emf/. Accessed: 2018-09-21.

[90] OMG. *Unified Modeling Language (UML) Version 2.0*. http://www.omg.org/spec/UML/2.0/. Accessed: 2018-09-21.

Résumé

De nos jours, les applications et systèmes d'information doivent gérer des masses de données hétérogènes tout en répondant à des exigences fonctionnelles variées et à des besoins de performance et de passage à l'échelle. Les systèmes de gestion de données NoSQL apportent diverses solutions et offrent, pour la plupart, beaucoup de souplesse dans la structuration des données. Ils permettent une structuration des données avec une grande flexibilité et sans création préalable d'un schéma (contrairement aux SGBD relationnels). Dans ces solutions il n'y a pas de séparation claire des couches logiques et physiques.

La flexibilité et l'absence de schéma dans les systèmes NoSQL orientés documents, telle que MongoDB, permettent diverses alternatives de structuration. Le choix de la structuration reste important et critique par ses impacts sur la qualité de la base et de ses applications. Dans cette thèse nous proposons de revenir sur une phase de conception dans laquelle des aspects de qualité et les impacts de la structure sont pris en compte afin de prendre une décision d'une manière plus avertie.

Nous proposons SCORUS, un système pour l'analyse et l'évaluation de structures orientées document qui vise à faciliter l'étude des possibilités de semi-structuration des données, et à fournir des métriques objectives pour mieux faire ressortir les avantages et les inconvénients de chaque solution par rapport aux besoins des utilisateurs. Pour cela, une séquence de trois phases peut composer un processus de conception. Chaque phase peut être aussi effectuée indépendamment à des fins d'analyse et de réglage des bases existantes. La stratégie générale de SCORUS est composée par :

— Génération d'un ensemble d'alternatives de structuration : dans cette phase nous proposons de partir d'une modélisation UML des données et de produire automatiquement un ensemble de variantes de structuration orienté document.

— Évaluation d'alternatives en utilisant un ensemble de métriques structurelles : cette évaluation prend un ensemble de variantes de structuration et calcule les métriques au regard des données modélisées.

— Analyse des alternatives évaluées : utilisation des métriques afin d'analyser les alternatives considérées et de choisir la ou les plus appropriées.

Cette thèse présente les outils théoriques et logiciels pour SCORUS ainsi que des expérimentations avec MongoDB.

Mots clés : NoSQL, modèle orienté document, structuration, variabilité, modèles de caractéristiques, métriques, évaluation, analyse.

Abstract

Nowadays, applications and information systems must handle large heterogeneous datasets while responding to many functional requirements as well as performance and scalability needs. NoSQL data management systems provide a variety of solutions. Most of them allow a data structuring with high flexibility and without prior creation of a schema (unlike relational DBMS). In such schema-less systems there is no clear separation of logical and physical layers.

Document-oriented NoSQL systems, such as MongoDB, allow many data structuring alternatives without facing the constraints of a schema. Nevertheless, the data structuring choice remains important and critical by its impacts on the quality of the base and its applications. In this thesis, we propose to return to a design phase to integrate aspects of quality and the impact of the data structure in order to improve the decision making.

In this context, we propose SCORUS, a system for the analysis and evaluation of document-oriented structures that aims to facilitate the study of data semi-structuring possibilities, and to provide objective metrics for better highlight the advantages and disadvantages of each solution in relation to the needs of the users. For this, a sequence of three phases can compose a design process. Each phase can also be performed independently for analysis and tuning purposes. The general strategy of SCORUS is composed by :

— Generation of a set of structuration alternatives : in this phase we propose to start from UML modeling of the data and to automatically produce a set of document-oriented structuring variants.

— Evaluation of alternatives using a set of structural metrics : This evaluation takes a set of structuring variants and calculates the metrics against the modeled data.

— Analysis of the evaluated alternatives : use of the metrics to analyze the alternatives and to choose the most appropriate one (s).

This thesis presents the theoretical and software tools for SCORUS as well as experiments with MongoDB.

Keywords : NoSQL, document-oriented model, structuring, variability, feature models, metrics, evaluation, analysis.