N° d'ordre

Résumé

L'évolution rapide des technologies d'information et l'augmentation massive des données dans les dernières années ont poussé les organisations à adopter les nouvelles techniques pour le stockage des différentes informations, ce qui a rendu les méthodes et les approches de reverse engineering et de la migration des bases de données un domaine de recherche riche et active. Pour gérer ce grand changement et améliorer la performance des systèmes actuels, le processus de migration passe par plusieurs étapes, l'une des étapes les plus importantes c'est l'identification des différents composants de la base de données. La compréhension de la base de données est une étape cruciale et critique car c'est le point d'entrée du processus de migration ou on peut identifier et extraire toutes les informations en relation avec (les tables, attributs, type de données, relations, contraintes, …)

Pour avoir un système solide et ouvert à l'extension; il faut donner aux administrateurs, les développeurs et les concepteurs la main pour manipuler la base de données et faire des modifications avec le strict minimum des régressions; pour cela un schéma conceptuel avec un niveau d'abstraction élevé qui représente le schéma physique de la base de données est indispensable, on propose dans notre approche une méthode de reverse engineering qui prend en entrée un schéma physique de la base de données et génère un schéma conceptuel avec une représentation graphique compréhensible sous forme d'un diagramme de classes UML, qui présente les différentes tables de la base de données avec les relations et les attributs associés.

Une source de données comme XML joue un rôle très important dans les nouvelles technologies, notamment dans les applications web modernes. Avec XML Schema on peut définir des contraintes sur les données et avec le langage de requête XQuery on peut manipuler et récupérer les informations facilement. Malgré ces points forts, les utilisateurs trouvent des difficultés pour comprendre le code de ces deux langages surtout les analystes concepteurs, pour cela nous avons proposé une méthode de reverse engineering de l'XML Schema et XQuery pour présenter les requêtes et les contraintes avec un langage facile et compréhensible. Ce langage est l'OCL (Object Constraint Language) un langage standard de W3C.

Une masse considérable de données est stockée dans des bases de données relationnelles (BDR), des bases de données objet relationnelle (BDOR) ou des bases de données orientées objets (BDOO), mais ces bases de données ont des limites par rapport à des bases de données relativement récentes, telles que des bases de données NoSQL (MongoDB, Cassandra, HBase, …). En conséquence nous avons proposé une approche de migration des données des bases de données objet relationnelle vers les bases de données NoSQL basées sur les documents (MongoDB).

Fouad TOUFIK

Transposition, comparaison et évaluation des méthodologies pour transformer les sources de données XML et objet relationnel en Big Data (NoSQL) et UML/OCL

2020/2021, MAI

Université Hassan 1er
Centre d'Études Doctorales

Faculté des Sciences et Techniques
Settat

THÈSE DE DOCTORAT

Pour l'obtention de grade de Docteur en informatique

Formation Doctorale: Mathématiques Appliquées et Informatique

Spécialité: Informatique

Sous le thème

Transposition, comparaison et évaluation des méthodologies pour transformer les sources de données XML et objet relationnel en Big Data (NoSQL) et UML/OCL

Présentée par :
Fouad TOUFIK

Soutenue le: *Jeudi 24 Juin 2021*
A la Faculté des Sciences et Techniques de Settat devant le jury composé de :

Pr. Mohamed ZAKARI	**PES**	**FST Settat**	Président
Pr. Mohammed Chaouki ABOUNAIMA	**PH**	**FST Fes**	Rapporteur
Pr. Noreddine GHERABI	**PES**	**ENSA Khouribga**	Rapporteur
Pr. Ilias CHERTI	**PES**	**FST Settat**	Rapporteur
Pr. Abdellah EZZATI	**PES**	**FST Settat**	Examinateur
Pr. Mohamed BAHAJ	***PES***	***FST Settat***	Directeur de thèse

Année Universitaire: 2020/2021

Dédicaces

A mes chers parents :

Nulle dédicace n'est susceptible de vous exprimer ma profonde affection, mon immense gratitude, pour tous les sacrifices que vous avez consentis pour mon éducation et mes études.

A mon épouse et mon fils :

Qu'ils trouvent ici l'expression de ma profonde gratitude et affection. Que dieu réunisse nos chemins pour un long commun serein et que ce travail soit témoignage de ma reconnaissance et de mon amour sincère et fidèle.

A mon frère à mes sœurs et à mes chers amis :

Avec toute mon affection, veuillez trouver ici l'expression de mon profond sentiment de respect et de reconnaissance, pour le soutien que vous n'avez cessé de me prodiguer.

A Mme Nainia Amina, Mme Essadik Mounia et à tous mes Enseignants:

J'ai eu l'honneur et la chance de bénéficier de vos connaissances et compétences, de vos précieux conseils et de votre suivi tout au long de mon parcours académique. Votre sens élevé du devoir, le fait d'être toujours montré à l'écoute ainsi que votre rigueur scientifique impose l'estime et le respect. Je vous remercie infiniment.

Et avant tout à l'Eternel, mon Dieu, le Tout puissant de m'avoir aidé à arriver au bout de mes études, lui qui m'a accompagné dès le début jusqu'à la fin, il est mon ombre à ma main droite!

Remerciements

Tout d’abord, je tiens à remercier mon directeur de thèse, monsieur Mohamed BAHAJ, pour la confiance qu'il m'a accordée en acceptant d'encadrer ce travail doctoral, pour ses multiples conseils et pour son précieux temps qu'il a consacré à diriger cette recherche. J'aimerais également lui dire à quel point j’ai apprécié sa grande disponibilité et son respect sans faille des délais serrés de relecture des documents que je lui ai adressés. Enfin, j’ai été extrêmement sensible à ses qualités humaines d'écoute et de compréhension tout au long de ce travail doctoral.

Je suis très honorée de remercier tous les membres de mon jury d’avoir accepté d’assister à la soutenance de ma thèse :

Monsieur Mohamed ZAKARI, Professeur à la Faculté des Sciences et Techniques de Settat, d’avoir accepté de participer au jury de cette thèse en tant que président.

Messieurs Mohammed Chaouki ABOUNAIMA (Faculté des Sciences et Techniques de Fes), Noreddine GHERABI (Professeur à l’Ecole Nationale des Sciences Appliquées de Khouribga) et Ilias CHERTI (Professeur à la Faculté des Sciences et Techniques de Settat), pour l’honneur qu’ils m’ont fait pour leurs participations à mon jury de thèse en tant que rapporteurs de mon travail, pour le temps consacré à la lecture de cette thèse, et pour les suggestions et les remarques pertinentes qu’ils m’ont prodiguées.

Monsieur Abdellah EZZATI, Professeur à la Faculté des Sciences et Techniques de Settat, d’avoir accepté de participer au jury de cette thèse en qualité d’examinateur.

Ces remerciements ne peuvent s'achever sans une pensée pour mes chers parents. Leurs présence et encouragements étaient pour moi les piliers fondateurs de ce que je suis et de ce que je fais. Je tiens à les remercier vivement surtout pour leur soutien moral permanent et leurs nombreux conseils tout le long de ma thèse.

Résumé

L'évolution rapide des technologies d'information et l'augmentation massive des données dans les dernières années ont poussé les organisations à adopter les nouvelles techniques pour le stockage des différentes informations, ce qui a rendu les méthodes et les approches de reverse engineering et de la migration des bases de données un domaine de recherche riche et active. Pour gérer ce grand changement et améliorer la performance des systèmes actuels, le processus de migration passe par plusieurs étapes, l'une des étapes les plus importantes c'est l'identification des différents composants de la base de données. La compréhension de la base de données est une étape cruciale et critique car c'est le point d'entrée du processus de migration ou on peut identifier et extraire toutes les informations en relation avec (les tables, attributs, type de données, relations, contraintes, …)

Pour avoir un système solide et ouvert à l'extension; il faut donner aux administrateurs, les développeurs et les concepteurs la main pour manipuler la base de données et faire des modifications avec le strict minimum des régressions; pour cela un schéma conceptuel avec un niveau d'abstraction élevé qui représente le schéma physique de la base de données est indispensable, on propose dans notre approche une méthode de reverse engineering qui prend en entrée un schéma physique de la base de données et génère un schéma conceptuel avec une représentation graphique compréhensible sous forme d'un diagramme de classes *UML*, qui présente les différentes tables de la base de données avec les relations et les attributs associés.

Une source de données comme *XML* joue un rôle très important dans les nouvelles technologies, notamment dans les applications web modernes. Avec *XML Schema* on peut définir des contraintes sur les données et avec le langage de requête *XQuery* on peut manipuler et récupérer les informations facilement. Malgré ces points forts, les utilisateurs trouvent des difficultés pour comprendre le code de ces deux langages surtout les analystes concepteurs, pour cela nous avons proposé une méthode de reverse engineering de *l'XML Schema* et *XQuery* pour présenter les requêtes et les contraintes avec un langage facile et compréhensible. Ce langage est l'*OCL (Object Constraint Language)* un langage standard de *W3C*.

Une masse considérable de données est stockée dans des bases de données relationnelles *(BDR)*, des bases de données objet relationnelle *(BDOR)* ou des bases de données orientées objets *(BDOO)*, mais ces bases de données ont des limites par rapport à des bases de données relativement récentes, telles que des bases de données *NoSQL (MongoDB, Cassandra, HBase, ...).* En conséquence nous avons proposé une approche de migration des données des bases de données objet relationnelle vers les bases de données NoSQL basées sur les documents (MongoDB).

Mot Clés : *base de données objet relationnel, XML, XML Schema, XQuery, UML, OCL, NoSQL.*

Abstract

The rapid evolution of information technologies and the massive increase of data in recent years have pushed organizations to adopt new techniques for the storage of different data, which has made the methods and approaches of reverse engineering and database migration a rich and active research area. To manage this big change and improve the performance of current systems, the migration process goes through several stages, one of the most important stages is the identification of the different components of the database. Understanding the database is a crucial and critical step because it is the entry point of the migration process where we can identify and extract all the information related to (tables, attributes, type of data, relationships, constraints ...)

To have a solid system open to extension; administrators, developers and designers must manipulate the database and make changes with the minimum of regressions; for this a conceptual schema with a high level of abstraction which represents the physical schema of the database is essential, we propose in our approach a reverse engineering method which takes as input a physical schema of the database and generates a conceptual schema with an understandable graphical representation in the form of a UML class diagram, which presents the different tables of the database with the relationships and associated attributes.

A data source like XML plays very important role in new technologies, especially in modern web applications. With XML Schema we can define constraints on data and with the XQuery query language for XML we can easily manipulate and retrieve information. Despite these strengths, users find it difficult to understand the code of these two languages, especially designer, therefore, we have proposed a method of transforming XML Schema and XQuery to an easy and understandable language. This language is the OCL (Object Constraint Language) a standard language of W3C.

A considerable amount of data is stored in relational databases, object relational databases or object-oriented databases, but these databases have limitations compared to recent databases, such as NoSQL databases (MongoDB, Cassandra, HBase,…). therefore, we have proposed an approach for data migration from object relational databases to document-oriented NoSQL databases (MongoDB).

Liste des Abréviations

	Abréviation	Désignation
A	API	Application Programming Interface
B	BDR	Base de Données Relationnelle
	BDOR	Base de Données Objet Relationnelle
	BDOO	Base de Données Orientée Objet
	BLOB	Binary Large Object
	BSON	Binary Javascript Object Notation
C	CMS	Content Management System
D	DTD	Document Type Definition
	DML	Data Manipulation Language
	DOM	Document Object Model
E	ER	Entity Relationship
	ETL	Extract Transform Load
H	HTML	Hypertext Markup Language
	HTTP	Hypertext Transfer Protocol
J	JSON	Javascript Object Notation
M	MCD	Modèle Conceptuel des Données
	MERISE	Méthode d'Étude et de Réalisation Informatique par les Sous-Ensembles ou pour les Systèmes d'Entreprise
	MDE	Model Driven Engineering
N	NoSQL	Not Only SQL
O	OCL	Object Constraint Language
	OQL	Object Query Language
	OMT	Object Modeling Technique
	OMG	Object Management Group
	ORM	Object Relationship Mapping
	ORDB	Object Relational DataBase
	OID	Object Identifier

P	PL	Procedural Language
R	RELAX	Regular Language for XML
	RELAX-NG	Regular Language for XML Next Generation
	REST	Representational State Transfer
S	SGBD	Système de Gestion des Base de Données
	SGBDR	Système de Gestion des Base de Données Relationnel
	SGBDRO	Système de Gestion des Base de Données Relationnel Objet
	SQL	Structured Query Language
	SGML	Standard Generalized Markup Language
	SRS	Software Requirement Specification
T	TREX	Tree Regular Expression
U	UML	Unified Modeling Language
	UDT	User Definition Type
	URI	Uniform Resource Identifier
X	XML	eXtensible Markup Language
	XQuery	XML Query
	XSLT	Extensible Stylesheet Language Transformations

Liste des figures

Liste des Tableaux

Table des matières

Introduction générale

I. Contexte général

Les bases de données représentent un ensemble de collections pour une gestion des données d'une manière efficace et utile, qui doit assurer un stockage approprié pour des grandes quantités de données, avec un accès facile et rapide qui facilite le traitement des données.

Les systèmes de bases de données ont été développés à l'origine dans les grandes sociétés et les grandes organisations comme la base des grands systèmes de traitement des transactions par les bureaux gouvernementaux, les bibliothèques, les hôpitaux. Après l'émergence des micro-ordinateurs, ils ont été connectés ensemble dans des groupes de travail, qui ont provoqué le déménagement de la technologie des bases de données vers la création des groupes de travail pour être utilisé aujourd'hui dans l'internet et les intranets.

Au cours des dernières années et avec l'évolution rapide des applications des différent types de données sont générées tels que les données des réseaux sociaux (audio, vidéo, images, …) sous forme de données semi-structuré et non structuré. Par conséquent de nouvelles bases de données qui peuvent assurer la gestion de ces données sont apparues comme les bases de données *NoSQL*.

Face au grand changement des données et à la masse importante des informations générées, de nombreuses entreprises transforment les schémas et migrent leurs données en remplaçant les bases de données classique *(relationnel, objet relationnel, xml)* par des bases de données *NoSQL*. La migration est une opération cruciale et couteuse en termes de perte de données pour cela la compréhension de la base de données cible et la génération d'un modèle conceptuel est une étape indispensable pour maintenir la base de données ou effectuer une migration sans erreur.

Le reverse engineering représente l'étude et l'analyse d'un système pour en déduire son fonctionnement interne, et se retrouve dans de nombreux domaines de l'ingénierie. Dans le domaine des bases données, faire du reverse engineering revient à élaborer un processus de transformation où tout composant doit se transformer. La transformation est équivalente au processus de migration d'une base de données source à une base de données cible, respectant la transformation du schéma physique, le mapping des données, les requêtes, l'intégrité des données, la sécurité des données, et l'indépendance des données.

Avec la croissance des données des 4V (Volume, Variété, Vélocité, Value), la gestion des données évolue de la scalabilité verticale vers la scalabilité horizontale pour une augmentation en performance et puissance. Des centaines de milliers de petits serveurs créent l'informatique répartie au lieu d'une seule machine puissante. Les données doivent être transformées en un modèle différent pour soutenir l'informatique distribuée. Les données devraient être plus variées, donc la flexibilité est nécessaire pour correspondre au format natif de l'information, elles peuvent être stockées dans des bases de données (orientées documents, graphiques, clé/valeur ou orientées colonne). La structure des données change avec les entreprises agiles. Mais un schéma prédéfini fort est limité par ce genre de scénario d'entreprise. Modification d'une colonne existante ou l'ajout des nouvelles colonnes nécessite de recréer la table dans les bases de données relationnelles ou objet relationnelles, mais il est flexible pour ajouter un nouvel attribut ou un objet composite dans une base de données *NoSQL*.

L'objectif de cette thèse est d'apporter une contribution en proposant des méthodes et des approches permettant de :

- Reverse engineering d'une base de données objet relationnelle pour obtenir un modèle conceptuel dans un haut niveau d'abstraction exprimé en un diagramme de classes *UML* et enrichi avec des expressions *OCL*, le modèle obtenu permet aux concepteurs, développeurs et administrateurs de base de données de bien comprendre la structure de la base de données puisqu'elle présente en langage naturel compréhensible par les humains.
- Toujours dans le but de bien comprendre la structure et les règles de gestion appliquées dans une source de donnée, la deuxième contribution consiste à rétro-concevoir un document *XML Schema* en transformant les restrictions définies en tant que *constraining facets* à des expressions *OCL* compréhensible qui aident à bien maintenir ce type de source de données. L'approche proposée couvre aussi le mapping des requêtes *XQuery* à des expressions *OCL*.
- Convertir le schéma et migrer les données d'une base de données objet relationnelle à la base de données *NoSQL* orienté document *MongoDB*.

II. Organisation du manuscrit

Ce manuscrit est organisé en deux grandes parties :

- **Partie I : État de l'art**

  Elle a comme objectif d'introduire les principaux concepts que nous aborderons le long de notre manuscrit, elle est composée de deux chapitres.

  - Chapitre 1 : présente une vue globale sur les différentes sources de données utilisées dans les différents axes de notre contribution à savoir les bases de données relationnelles, objet relationnelles, *XML* et les bases de données *NoSQL* ainsi qu'une petite introduction sur le langage de modélisation *UML* avec *OCL*.
  - Chapitre 2 : synthétise une panoplie de travaux connexes à nos trois axes de contribution en mettant en exergue les avantages et les limites de chaque approche pour mieux positionner notre travail.

- **Partie II : Contribution**
  - Chapitre 3 : décrit notre méthode de *reverse engineering* d'une base de données objet relationnelle à un modèle conceptuel en langage naturel compréhensible exprimé en diagramme de classes *UML* et enrichi avec des expressions *OCL*. Notre méthode transforme les restrictions définies en tant que contraintes déclaratives ou implémentées sous forme de *trigger* à des expressions *OCL*.
  - Chapitre 4 : présente le deuxième axe de notre contribution permettant d'assurer la transformation des *constraining facets* définies dans un document *XML* schéma vers des expressions *OCL* facile à lire et à comprendre, dans cette approche nous avons proposé la transformation des requêtes XQuery vers des expressions *OCL*.
  - Chapitre 5 : décrit l'approche proposée pour effectuer la transformation du schéma et la migration des données d'une base de données objet relationnel à une base de données *NoSQL* orientée document *MongoDB*.

Ce manuscrit se termine par une conclusion générale des travaux réalisés et par les perspectives que nous envisagions pour compléter et améliorer notre approche.

III. Publication

La liste suivante présente les publications concernant les travaux effectués dans cette thèse.

1. Article Publiés

- Toufik, F., & Bahaj, M. Transforming XML schema constraining facets and XML queries to object constraint language (OCL). Journal of Theoretical and Applied Information Technology. Vol 87 No 3 469-478 May 2016.

- Toufik, F., & Bahaj, M. Extracting UML Models and OCL integrity constraints from Object Relational Database. Journal of Theoretical and Applied Information Technology. Vol 96 No 4 1138-1149 February 2018.

- Ain El Hayat, S., Toufik, F., Bahaj, M. UML/OCL based design and the transition towards temporal object relational database with bitemporal data, Journal of King Saud University - Computer and Information Sciences, Volume 32, Issue 4, Pages 398-407, 2020.

2. Article soumis

- Toufik, F., & Bahaj, M. Model Transformation And Data Migration From Object Relational Database to NoSQL Document Database. Journal of Applied Research and Technology.

- Ain El Hayat, S., Toufik, F., Bahaj, M. Transforming TRDB into TORDB Including Bitemporal data: Rules. Advances in Science, Technology and Engineering Systems Journal.

3. Conférences Internationales Indexées

- Toufik, F., & Bahaj, M. Reverse Engineering of Object Relational Database. International Conference on Software Engineering and Information Management 2018, pp. 73–76. Association for Computing Machinery. ACM.

- Toufik, F., & Bahaj, M. Extracting OCL Integrity Constraints from Object Relational Database. International Conference on Control, Automation and Diagnosis 2018, pp. 1-6. IEEE
- Toufik, F., & Bahaj, M. Model Transformation From Object Relational Database to NoSQL Document Database. International Conference on Networking, Information Systems & Security 2019, Article No. 49. Association for Computing Machinery. ACM.
- Toufik, F., & Bahaj, M. Model Transformation From Object Relational Database to NoSQL Column Database. International Conference on Networking, Information Systems & Security 2020, Article No. 62. Association for Computing Machinery. ACM.

4. Conférences internationales avec comités de lecture

- Toufik, F., & Bahaj, M. Transforming XML Schema Constraining Facet to Object Constraint Language. Advanced Information technology, Services and Systems 2015.

PARTIE 1 :

ETAT DE L'ART

Chapitre1 : Les Sources de données et Langage de Modélisation

Ce chapitre présente une introduction générale sur les différentes composantes et technologies des bases de données, les langages de modélisation objet ainsi que les principales techniques de reverse engineering.

I. Introduction

Une base de données est un ensemble ou une collection de données homogènes, organisées et structurées qui permettent de décrire des personnes ou des objets complexe du monde réel. Son rôle principal est de faciliter la manipulation des données à l'aide des moyens informatiques afin de récupérer ou produire l'information cherchée. Pour gérer ces données, il existe une couche intermédiaire entre les données sauvegardées physiquement dans des disques dure ou des bandes de stockage qui facilite le stockage, le traitement et la recherche des données, cette couche est un logiciel appelé un Système de Gestion de Base de Données *(SGBD).*

Un *SGBD* est un ensemble de programmes qui contient une ou plusieurs bases de données et permet aux différents utilisateurs de manipuler leurs données (création de nouvelle base de données, insertion modification et suppression). Plus les commandes basiques des *SGBDs*, en même temps ils offrent des mécanismes robustes et très puissants pour garder la cohérence de la base de données et éliminer la perte des données à l'aide de (sauvegarde automatique, maintenance des données, récupération et contrôle des données).

Avec l'évolution rapide du web, et le grand nombre des utilisateurs qui créent les données, on trouve différent type des bases de données qui peuvent servir et répondre au besoin élevé de stockage. Parmi les types de données les plus connues, on trouve par exemple les données *(Relationnelles, Orientée Objet, Objet Relationnel, NoSQL, XML …)* et chaque base de données nécessite un langage spécifique d'interrogation pour permettre aux utilisateurs d'exploiter ces données *(SQL, XQuery, OQL, …).*

Dans cette thèse, nos travaux se concentre essentiellement sur les bases de données structurées Objet Relationnel, les bases de données semi structurées *XML* et finalement les bases de données non structurées *NoSQL*.

Dans ce chapitre nous allons présenter des généralités sur les bases de données faisant parti de notre sujet de thèse.

II. Les Bases de Données Relationnelles

Les bases de données relationnelles sont les plus répandues et utilisés. Sont des bases de type tabulaire, elles se basent sur le modèle relationnel définit par le mathématicien britannique *Edgar Frank Codd* du centre de recherche IBM. Basés sur des concepts mathématiques solide comme la logique et la théorie des ensembles, le modèle relationnel est le plus adopté jusqu'à nos jours par une grande majorité des systèmes d'information des entreprises, en plus de son utilisation comme un modèle par défaut sur la majorité des sites web.

Codd a défini une séparation entre la représentation logique des données et la technique adoptée pour leur stockage physique. Avec la contribution de *P.Chen*, le modèle relationnel s'est enrichi par le modèle entité-relation avec l'ajout d'un niveau conceptuel aux deux niveaux logique et physique définit par le modèle relationnel. Le modèle conceptuel présente le schéma physique de la base de données à un haut niveau d'abstraction et son rôle principal est de fournir une modélisation graphique des données sous forme des entités et des dépendances.

Les bases de données présentent le noyau du système d'information. Pour bien mettre en place un système d'information solide et puissant il faudra une conception détaillée de la base de données, ce qui rend la phase de conception la plus difficile et critique dans le développement d'un tel système.

Il existe plusieurs langages et méthodes de conception des bases de données, ces langages ou méthodes se basent sur plusieurs étapes en utilisant des diagrammes ou des modèles pour représenter les objets qui constituent les systèmes d'information ainsi que les relations entre ces objets, comme le modèle conceptuel de données *(MCD)* de la méthode *MERISE (méthode d'informatisation de projet informatique des organisations)* ou le diagramme de classes du langage de modélisation graphique *UML*. Malgré que le diagramme de classes a pour rôle la conception des bases de données objet, mais on peut l'utiliser pour concevoir les bases de données relationnelles.

III. Les Bases de Données Objet Relationnelles

Une approche objet-relationnelle pour gérer les données à l'aide des Système de gestion de base de données Objet Relationnel, comprend des objets qui ont besoin de persistance, un modèle de données, un langage de requête pour manipuler, récupérer et stocker des données. La persistance est le processus de stockage des informations qui sont conservées après la fin d'une application.

Un *Data Model* est une organisation logique des objets du monde réel avec leurs relations et leurs contraintes. Les objets et les relations entre eux peuvent être représentés dans des diagrammes de *Entity Relationship (ER)* en utilisant des techniques de modélisation telles que la notation *Coad / Yourdon*, la notation *Shlaer / Mellor*, la notation *Booch* ou le langage de modélisation le plus utilisé *UML*. La modélisation est très importante car elle permet aux développeurs de représenter les relations entre les entités à l'aide d'une représentation standard, cette représentation peut être transformée par la suite à un schéma physique d'une base de données.

Une base de données Objet Relationnel est une collection d'objets dont le comportement, l'état et les relations peuvent être manipulés à l'aide des méthodes définies au sein de l'objet.

Pourquoi utiliser le modèle objet relationnel ?

Si le modèle relationnel a répondu aux besoins de stockage de données au cours des trente dernières années, qu'est-ce qui a changé pour que les développeurs doivent stocker des types de données et des relations complexes ? Pour répondre à cela, il faut identifier pourquoi des données et des relations complexes existent, est ce que les ingénieurs logiciels ont créé ce nouveau phénomène, ou cela indique que le modèle relationnel a toujours été inadéquat pour stocker ces types de données et de relations nécessaires au développement des applications.

Avec l'augmentation massive des données non structurées et numérique comme les photos, la voix et les vidéos en raison de la réduction des coûts de stockage et l'augmentation des technologies d'enregistrement et transfert numérique, le besoin de stockage des données complexe a augmenté. Par conséquent, en raison de l'incapacité du modèle relationnel à représenter les concepts de la programmation orientée objet et à stocker des données complexes, les développeurs ont besoin d'un nouveau modèle de données pour stocker les objets du monde réel des applications.

Le modèle objet relationnel à certainement des avantages par rapport au modèle relationnel à savoir :

- Facilité de modélisation des objets et des relations du monde réel.
- Possibilité de créer des nouveaux types définis par les utilisateurs.
- Encapsulation des objets.
- Référencement des objets.

1. Facilité de modélisation des objets et des relations du monde réel

L'utilisation des systèmes de gestion de base de données objet relationnel donne aux développeurs des applications la possibilité de présenter les entités et les relations du monde réel. Non seulement cela améliore les capacités des concepteurs à représenter les problèmes dans les applications, mais l'utilisation d'un système de gestion de base de données objet relationnel permet également aux concepteurs de penser aux problèmes à un niveau d'abstraction plus élevé sans avoir la charge de mapper les données d'application aux tables relationnelles. Donc les développeurs ne sont pas besoin de contourner les limites du modèle relationnel ce qui diminue le temps de développement.

La qualité globale de l'application pourrait être définie comme la façon dont la conception globale modélise le monde réel, sa résistance au changement et la maintenabilité de l'application tout au long du cycle de vie du logiciel.

Par conséquent, l'utilisation d'une approche orientée objet pour résoudre la persistance des données des applications pourrait permettre aux développeurs de concevoir des applications de meilleure qualité.

2. Possibilité de créer des nouveaux types définis par les utilisateurs

La possibilité de créer de nouveaux types de données augmente l'expressivité et la maintenabilité dans les applications orientées objet, car les objets peuvent être stockés directement dans une base de données sans les convertir en types de données relationnels.

Les *UDTs (User Definition Type)* sont nécessaires pour résoudre le problème de stockage des données complexes. Sans *UDT*, une colonne de table est limitée aux types de données pris en charge par *SQL*. Ce qui oblige les développeurs à créer un code

d'application complexe qui augmente le temps d'exécution et pourrait être éliminé si un *UDT* était créé.

3. Encapsulation des objets

Dans les *SGBDR* traditionnels, les données sont stockées dans des tables relationnelles tandis que les procédures stockées sont stockées dans le schéma de la base de données, par contre dans les systèmes de base de données objet relationnel on peut stocker les données avec les méthodes qui manipule ces données ensemble. Dans les *SGBDRs* les données sont manipulées par les procédures stockées ou des commandes SQL, dans les *SGBDROs* les données peuvent être manipulées par des méthodes stockées avec des attributs dans les *UDTs*.

L'utilisation des méthodes réduit le nombre de méthode défini au niveau applicatif par les développeurs et garantit aux administrateurs de base de données une manipulation correcte des données. Par exemple, si le *salaire* est un attribut de l'objet *employé* stocké en tant que valeur annuelle, la base de données peut fournir des méthodes à *getMonthlySal()* et *getAnnualSal()*, pour garantir que les valeurs correctes sont récupérées. La technique de stocker les données et les méthodes en un seul ensemble c'est le principe d'encapsulation de la programmation orienté objet.

4. Référencement des objets

Dans les *SGBDRs*, les requêtes de jointures sont considérées comme l'une des opérations les plus coûteuses, tandis que dans les *SGBDROs*, les tables associées sont accessibles en utilisant des références d'objet au lieu des jointures.

Les références aux objets permettent de naviguer facilement entre les tables. L'utilisation d'une référence d'objet à une table associée peut être plus efficace que l'utilisation d'une clé étrangère relationnelle.

IV. XML, DTD et XML Schema

XML signifie *eXtensible Markup Language*, il s'agit d'un langage de balisage textuel dérivé du langage *SGML (Standard Generalized Markup Language)*. *XML* est libre et indépendant de toute plateforme logicielle ou matérielle, il est devenu une recommandation de W3C dès 1998. Le point fort de *XML* est un langage générique

extensible, sa structure et syntaxe permet de définir d'autres langages avec vocabulaire et grammaire différent.

XML est très utilisé dans la majorité des applications web et joue un rôle très important pour assurer la communication entre les applications hétérogènes. A l'aide de sa structure et sa manière de stockage des données.

1. Description du Langage XML

XML est un langage de balisage qui définit un ensemble de règles pour l'encodage de documents dans un format à la fois lisible par l'homme et lisible par la machine. Alors, quel est exactement un langage de balisage ? Le balisage est une information ajoutée à un document qui améliore sa signification à certains égards, en ce qu'elle identifie les parties et leur relation les unes avec les autres. Plus précisément, un langage de balisage est un ensemble de symboles qui peuvent être placés dans le texte d'un document pour délimiter et étiqueter les parties de ce document.

Les balises *XML* identifient les données et sont utilisées pour stocker et organiser les données, plutôt que de spécifier comment les afficher comme les balises *HTML*. *XML* ne remplacera pas *HTML*, mais il introduit de nouvelles possibilités en adoptant de nombreuses fonctionnalités réussies de *HTML*.

L'extraction du contenu des documents *XML* se fait par des parseurs analyseur *(Parser).* Un analyseur *XML* est une bibliothèque de logiciels ou un package qui fournit une interface permettant aux applications clientes de travailler avec des documents *XML*. Il vérifie le format approprié du document *XML* et peut également valider les documents *XML*. Les navigateurs modernes ont des analyseurs *XML* intégrés.

Un document *XML* est considéré comme valide si son contenu correspond aux éléments, attributs et déclaration de type de document associé *(DTD),* et si le document est conforme aux contraintes qui y sont exprimées. La validation est traitée de deux manières par le parseur *XML* :

- Document *XML* bien formé.
- Document *XML* Valide.

Un document *XML* est dit bien formé s'il respecte les règles suivantes spécifiées par le W3C :

- Le document *XML* doit commencer par une déclaration *XML*.
- Le document *XML* doit avoir un unique élément racine.
- Tous les éléments *XML* doivent avoir une balise de fermeture.
- Les balises sont sensibles à la casse.
- Les éléments *XML* doivent être bien positionnés (pas d'imbrication sans inclusion).
- Les valeurs d'attributs *XML* doivent toujours être entre guillemets (simple ou double).

Selon le W3C, un document *XML* valide, est un document *XML* bien formé et respecte aussi les règles définies dans un modèle ou un schéma (une grammaire). La grammaire d'un document est un ensemble de règles qui décrivent la structure d'un document *XML*.

Un document *XML* peut contenir sa grammaire dans le même fichier, mais dans la majorité des cas, la grammaire est définie dans un fichier séparé, ce qui offre la réutilisation de la même grammaire dans plusieurs documents *XML*.

Pour définir la grammaire d'un document *XML*, plusieurs langages sont introduits à savoir :

- *DTD (Document Type Definition).*
- *XML Schema.*
- *TREX (Tree Regular Expression).*
- *RELAX (Regular Language For XML).*
- *RELAX-NG (Regular Language For XML Next Generation).*

Par la suite on va présenter une description des deux premiers langages.

2. Document Type Definition (DTD)

DTD est un document permettant de décrire un document *XML*, les *DTDs* vérifient le vocabulaire et la validité de la structure des documents *XML* par rapport aux règles grammaticales du langage *XML* approprié. Tout document faisant référence à une *DTD* doit respecter le modèle associé.

La *DTD* peut être classée sur sa base de déclaration dans le document *XML*, telle que *DTD* interne ou externe. Lorsqu'une *DTD* est déclarée dans le fichier, elle est appelée *DTD* interne et si elle est déclarée dans un fichier séparé, elle est appelée *DTD* externe.

La figure suivante présente une syntaxe basique d'une *DTD* classique

```
1
2   <!DOCTYPE element DTD identifier
3   [
4       declaration1
5       declaration2
6       ........
7   ]>
8
```

Figure 1: Exemple d'une DTD classique

Dans la figure ci-dessus,

- La *DTD* commence par le délimiteur *<! DOCTYPE*.
- *element* demande au parseur d'analyser le document à partir de l'élément racine spécifié.
- *DTD identifier* est un identifiant pour la définition du type de document, qui peut être le chemin d'accès à un fichier sur le système ou l'URL d'un fichier sur Internet. Si la *DTD* pointe vers un chemin externe, elle est appelée sous-ensemble externe.
- *Les crochets []* entourent une liste facultative de déclarations d'entité appelée sous-ensemble interne.

Une *DTD* est appelée *DTD* interne si les éléments sont déclarés dans le fichier *XML*. Pour le référencer en tant que *DTD* interne, l'attribut *standlone* dans la déclaration *XML* doit être *yes*. Cela signifie que la déclaration fonctionne indépendamment d'une source externe.

L'exemple suivant présente un fichier *XML* avec une *DTD* interne :

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "yes" ?>
<!DOCTYPE person [
   <!ELEMENT person (name,age,phone)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT age (#PCDATA)>
   <!ELEMENT phone (#PCDATA)>
]>

<person>
   <name>TOUFIK Fouad</name>
   <age>29</age>
   <phone>0644550028</phone>
</person>

```

Figure 2: Exemple d'une DTD interne

La déclaration *DOCTYPE* a un point d'exclamation *(!)* Au début du nom de l'élément. Le *DOCTYPE* informe le parseur qu'une *DTD* est associée à ce document *XML*. La déclaration *DOCTYPE* est suivie du corps de la *DTD*, où on déclare les éléments, les attributs et les entités. *DOCTYPE person* indique que l'élément *person* est la racine du document.

Plusieurs éléments sont déclarés au sein de l'élément *person* qui compose le vocabulaire du document. *<!ELEMENT name (#PCDATA)>* définit l'élément *name* comme étant de type *"#PCDATA"*. Ici, *#PCDATA* signifie des données texte analysables et ne doit pas avoir des éléments imbriqués. Le même principe s'applique aux autres éléments *(age, phone).*

Dans une *DTD* externe les éléments sont déclarés en dehors du fichier *XML*. On y accède en spécifiant l'attribut *system* qui peut être soit le fichier *.dtd*, soit une URL valide. Pour référencer une *DTD* externe, l'attribut *standlone* dans la déclaration *XML* doit être *no*. Cela signifie que la déclaration comprend des informations provenant d'une source externe. L'exemple suivant montre un document *XML* avec une *DTD* externe.

```
1
2   <!-- person.dtd -->
3   <!ELEMENT person (name,age,phone)>
4   <!ELEMENT name (#PCDATA)>
5   <!ELEMENT age (#PCDATA)>
6   <!ELEMENT phone (#PCDATA)>
7
```

```
1  <?xml version = "1.0" encoding = "UTF-8" standalone = "no" ?>
2  <!DOCTYPE person SYSTEM "person.dtd">
3  <person>
4     <name>TOUFIK  Fouad</name>
5     <age>29</age>
6     <phone>0644550028</phone>
7  </person>
8
```

Figure 3: Exemple d'une DTD externe

Limites de DTD

Malgré la grande contribution de la *DTD*, mais elle reste limitée par rapport à l'évolution rapide du web, avec des exigences grammaticales de plus en plus compliquées. Les limites syntaxiques de la *DTD* ne permettent plus d'accompagner cette évolution, ce qui rend l'utilisation des autres langages est une nécessité pour résoudre les problèmes et les limites de *DTD*, parmi ces limites on cite :

- Il ne prend pas en charge les espaces de nom. Les espaces de nom est un mécanisme par lequel les noms d'éléments et d'attributs peuvent être attribués à des groupes. Cependant, dans une *DTD*, les espaces de noms doivent être définis dans la *DTD*, ce qui viole l'objectif d'utilisation des espaces de noms.
- Il prend uniquement en charge le type de données de chaîne de texte.
- Il n'est pas orienté objet. Par conséquent, le concept d'héritage ne peut pas être appliqué sur les *DTD*.
- Possibilités limitées d'exprimer la cardinalité des éléments.

3. XML Schema

XML Schéma a été développé pour résoudre et remédier aux limites des *DTD* tout en assurant les fonctionnalités de l'ancien langage. *XML Schéma* est une recommandation de W3C dès Mai 2001.

XML Schéma ou (XML Schema Definition), il est utilisé pour décrire et valider la structure et le contenu des documents *XML*. Le *XML* Schéma définit les éléments, les attributs et les types de données. L'élément de schéma prend en charge les espaces de nom. Il est similaire à un schéma de base de données qui décrit les données d'une base de données.

Les Avantages de XML Schéma

Les espaces de nom

Un espace de nom est un mécanisme par lequel l'élément et le nom d'attribut peuvent être attribués à un groupe. L'espace de noms est identifié par *URI (Uniform Resource Identifier)*. Ce qui permet d'utiliser plusieurs vocabulaires dans le même document.

Les Types de données

Au contraire des *DTD*, dans *XML* Schéma on peut gérer nos données et déclarer chaque attribut avec le type adéquat (chaine de caractères, nombres entiers et réels, booléens, dates, …) et aussi on peut créer d'autres types à partir des types prédéfinis.

L'héritage

Dans *XML* Schéma on peut bénéficier de l'héritage, ce qui permet à un élément d'hériter les éléments et les attributs d'un autre élément ce qui rend la réutilisation des différents éléments possible.

Les indicateurs

Les indicateurs contrôlent la façon dont les éléments doivent être organisés dans un document *XML*. Il existe sept types d'indicateurs, classés en trois grandes catégories

1) Les indicateurs d'ordre
   a) *all* : Les éléments enfants peuvent apparaître dans n'importe quel ordre.
   b) *choice* : Un seul élément enfant peut se produire.
   c) *sequence* : L'élément enfant ne peut apparaître que dans l'ordre spécifié.

1) Les indicateurs d'occurrence
   a) *maxOccurs :* L'élément enfant ne peut apparaître que *maxOccurs* nombre de fois.
   b) *minOccurs :* L'élément enfant doit apparaître *minOccurs* nombre de fois.
1) Les indicateurs de groupe
   a) *group* : Définit un ensemble d'éléments associés.
   b) *attributGroup* : Définit un ensemble d'attributs associés.

Les exemples suivants présentent le mécanisme de travail de chaque indicateur ainsi que l'utilisation des attributs, des types simples et complexes.

```
1   <xs:complexType name = "PersonneType" mixed = "true">
2      <xs:all> or <xs:choice> or <xs:sequence>
3         <xs:element name = "nom" type = "xs:string"/>
4         <xs:element name = "prenom" type = "xs:string"/>
5         <xs:element name = "surnom" type = "xs:string"/>
6         <xs:element name = "age" type = "xs:positiveInteger"/>
7      </xs:all> or </xs:choice> or </xs:sequence>
8      <xs:attribute name = 'dateNaissance' type = 'xs:date'/>
9   </xs:complexType>
10
11  <xs:element name = 'personne' type = 'PersonneType' />
12
```

Figure 4: XML schéma Exemple de all, choice, sequence

En utilisant l'indicateur *all*, un élément *personne* peut avoir *nom*, *prenom*, *surnom*, *age* et marque l'élément fils dans n'importe quel ordre dans le document XML.

En utilisant l'indicateur *choice* un élément *personne* ne peut avoir qu'un seul élément *nom*, *prenom*, *surnom*, *age* et marque l'élément fils dans le document XML.

En utilisant l'indicateur *sequence* un élément *personne* peut avoir *nom*, *prenom*, *surnom*, *age* et marque l'élément fils dans l'ordre spécifié uniquement dans le document XML.

```xml
<xs:complexType name = "PersonneType" mixed = "true">
   <xs:all>
      <xs:element name = "nom" type = "xs:string"/>
      <xs:element name = "prenom" type = "xs:string"/>
      <xs:element name = "surnom" type = "xs:string" maxOccurs="5"  minOccurs="2"/>
      <xs:element name = "age" type = "xs:positiveInteger"/>
   </xs:all>
   <xs:attribute name = 'dateNaissance' type = 'xs:date'/>
</xs:complexType>

<xs:element name = 'personne' type = 'PersonneType' />
```

Figure 5: XML schéma Exemple de minOccurs et maxOccurs

En utilisant les indicateurs *minOccurs* et *maxOccurs*, un élément *personne* doit avoir deux *surnoms* et peut avoir au maximum cinq dans le document XML.

```xml
<xs:group name = "infogroup">
   <xs:sequence>
      <xs:element name = "nom" type = "xs:string"/>
      <xs:element name = "prenom" type = "xs:string"/>
      <xs:element name = "dateNaissance" type = "xs:date"/>
   </xs:sequence>
</xs:group>

<xs:element name = "personne" type = "personneType"/>

<xs:complexType name = "personneType">
   <xs:sequence>
      <xs:group ref = "infogroup"/>
      <xs:element name = "age" type = "xs:integer"/>
   </xs:sequence>
</xs:complexType>
```

Figure 6: XML schéma Exemple de group

En utilisant l'indicateur *group* on a la possibilité de regrouper un ensemble d'élément. Ici nous avons créé un groupe *infogroupe* qui contient l'ensemble (*nom*, *prenom*, *dateNaissance*) et puis l'utilisé pour définir l'élément *personne*.

```xml
<xs:attributeGroup name = "infogroup">
   <xs:sequence>
      <xs:attribute name = "nom" type = "xs:string"/>
      <xs:attribute name = "prenom" type = "xs:string"/>
      <xs:attribute name = "dateNaissance" type = "xs:date"/>
   </xs:sequence>
</xs:attributeGroup>

<xs:element name = "personne" type = "personneType"/>

<xs:complexType name = "personneType">
   <xs:sequence>
      <xs:attributeGroup ref = "infogroup"/>
      <xs:element name = "age" type = "xs:integer"/>
   </xs:sequence>
</xs:complexType>
```

Figure 7: XML schéma Exemple de attributeGroup

En utilisant l'indicateur *attributeGroup* on a la possibilité de regrouper un ensemble d'attribut. Ici nous avons créé un groupe *infogroupe* qui contient l'ensemble (*nom*, *prenom*, *dateNaissance*) et puis utilisé ce groupe pour définir les attributs de l'élément *personne*.

V. Les bases de données NoSQL

NoSQL est un système de gestion de base de données non relationnel, qui ne nécessite pas un schéma prédéfini et fixe, évite les jointures et facilement évolutif. Les bases de données *NoSQL* est utilisé pour les données distribuées avec besoin de stockage énorme. *NoSQL* est utilisé pour le *Big Data* et les applications web en temps réel. Par exemple des entreprises comme *Google, Twitter, Amazon et Facebook* qui collectent chaque jour des téraoctets de données des utilisateurs.

Les *SGBDRs* traditionnels utilisent le langage *SQL* pour stocker et récupérer les données, par contre les bases de données *NoSQL* comprennent un ensemble large des technologies qui peuvent traiter les données structurées, semi-structurées et non structurées.

Le concept de bases de données *NoSQL* est devenu populaire auprès des géants acteurs du web comme *Google, Facebook, Amazon*, etc. qui traitent énorme volume de données. Le temps de réponse du système devient très lent lors de l'utilisation des *SGBDRs* pour manipuler un ensemble de données volumineux. Pour résoudre ce problème, on peut

faire un «*Scale UP*» ou faire évoluer nos systèmes en mettant à niveau notre matériel existant. Mais ce processus est très coûteux. L'alternative à ce problème consiste à répartir la charge de la base de données sur plusieurs hôtes chaque fois que la charge augmente. Cette méthode est connue sous le nom de «*Scaling out*» ou la mise en échelle.

Les bases de données *NoSQL* possèdent des caractéristiques communes telles que le partitionnement horizontal sur plusieurs serveurs, l'évolutivité, la flexibilité ainsi que la sclabilité (montée en charge en gardant la performance). En outre, ces bases de données respectent aussi le théorème du ***CAP*** :

Consistance : Les données doivent rester cohérentes même après l'exécution d'une opération. Cela signifie qu'une fois les données écrites, toute future demande de lecture devrait contenir ces données.

Availibility : La base de données doit toujours être disponible et réactive. Il ne devrait pas avoir de temps d'arrêt même en cas de panne.

Tolérance au Partitionnement : La tolérance de partition signifie que le système doit continuer à fonctionner même si la communication entre les serveurs n'est pas stable.

Il existe principalement quatre catégories de bases de données *NoSQL*. Chacune de ces catégories a ses avantages et ses limites. Aucune base de données spécifique n'est meilleure pour résoudre tous les problèmes. La sélection d'un type de base de données se fait en fonction des besoins du système.

1. Les bases de données Clé/valeur

Les données sont stockées dans des paires clé / valeur. Il est conçu de manière pour gérer de nombreuses données avec une charge importante.

Les bases de données de paires clé-valeur stockent les données sous forme de table de hachage où chaque clé est unique, et la valeur peut être un *JSON*, un *BLOB (Binary Large Objects)*, une chaîne de caractère, etc. Il s'agit de l'un des types de bases de données *NoSQL* les plus basiques. Ce type de base de données est utilisé comme une collection, des dictionnaires, des tableaux associatifs (Exemple : *Redis, Dynamo, Riak*, …).

Clé	Valeur
Nom	TOUFIK
Prenom	Fouad
Age	29
Pays	Maroc

Figure 8: Exemple de données clé/valeur

2. Les bases de données orientées colonnes

Les bases de données orientées colonnes fonctionnent sur des colonnes et sont basées sur le principe de *BigTable* de *Google*. Chaque colonne est traitée séparément. Ils offrent des performances élevées sur les requêtes d'agrégation comme *SUM, COUNT, AVG, MIN* car les données sont facilement disponibles dans une colonne.

Les bases de données *NoSQL* orientées colonnes sont largement utilisées pour gérer les entrepôts de données, le Business Intelligence, le CRM (Exemple : *HBase, Cassandra,* …)

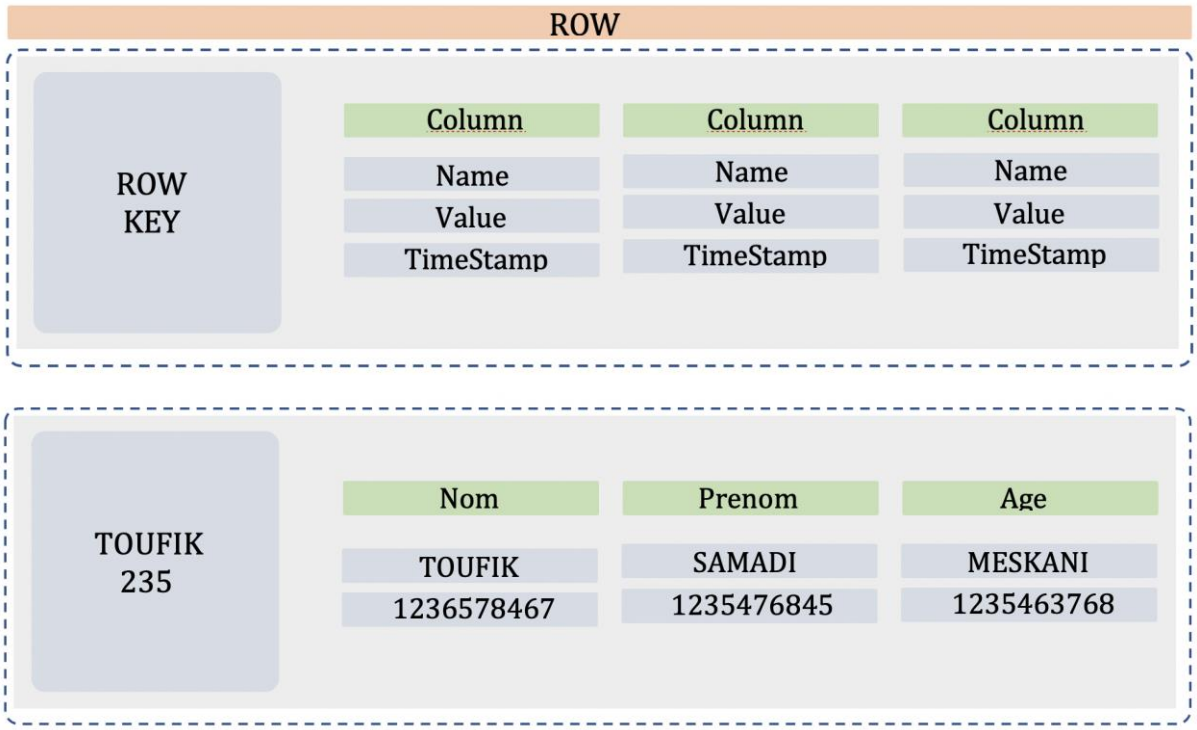


Figure 9: Exemple de données basées sur des colonnes

3. Les bases de données orientées documents

Les bases de données orientée document stocke et récupère les données sous forme de pair clé valeur, mais la partie valeur est stockée sous forme de document. Le document peut être sous format *JSON* ou *XML*.

Ce type de base de données est principalement utilisé pour les systèmes *CMS*, les plates-formes de blogs et les applications de commerce électronique. Il ne doit pas être utilisé pour les transactions complexes qui nécessitent plusieurs opérations ou requêtes sur différentes structures (Exemple : *MongoDB, CouchDB, Amazon SimpleDB*, …).

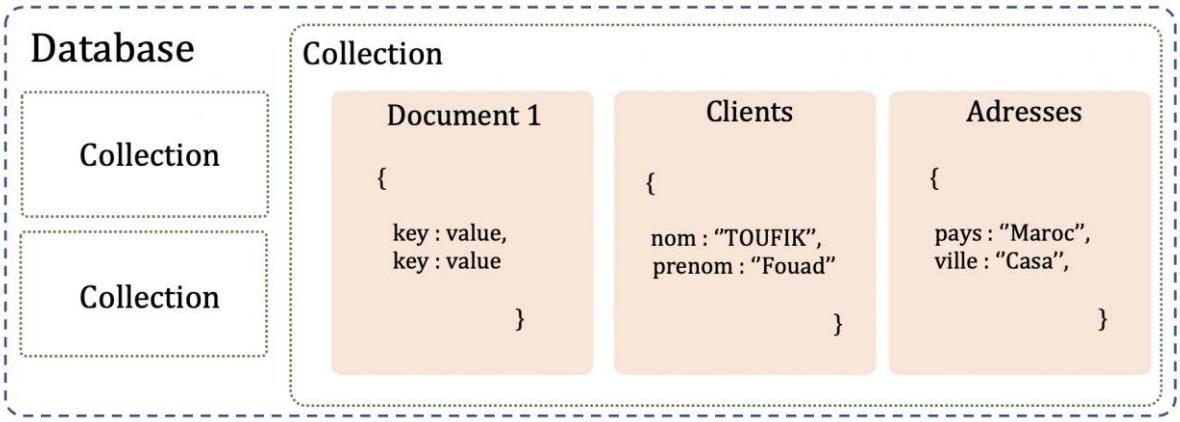


Figure 10: Exemple de données basé sur des documents

4. Les bases de données orientées graphes

Ce type de base de données stocke les entités ainsi que les relations entre ces entités. L'entité est stockée en tant que nœud avec la relation en tant qu'arêtes. Une arête donne une relation entre les nœuds. Chaque nœud et arête a un identifiant unique.

Les bases de données orientées graphes sont souvent et principalement utilisées pour les réseaux sociaux, la logistique et les données spatiales.

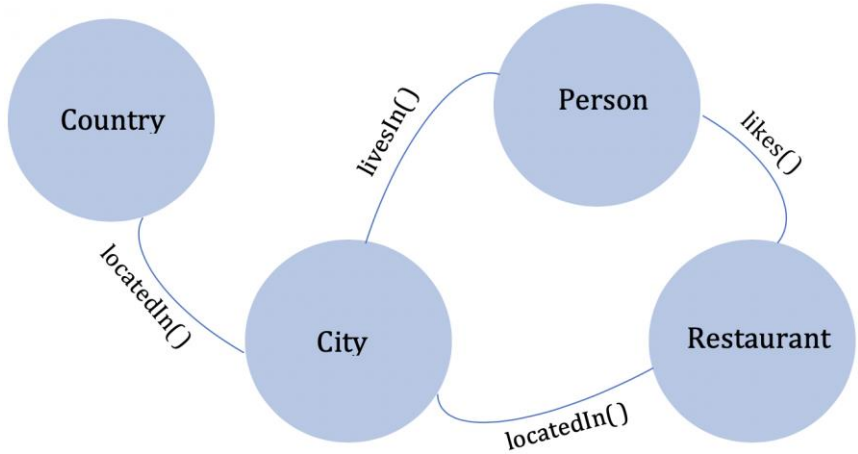


Figure 11: Exemple de données basé sur des graphes

Dans cette thèse, on s'intéresse spécialement aux bases de données orientées document *(JSON/BSON)* ainsi qu'aux bases de données orientées colonnes.

VI. La modélisation avec UML et OCL

1. UML

Le langage de modélisation unifié *(UML)* est un langage de modélisation visuelle à usage général pour les systèmes. Bien *qu'UML* soit le plus souvent associé à la modélisation de logiciels orientés objet, il a une application beaucoup plus large que cela en raison de ses mécanismes d'extensibilité intégrés. *UML* a été conçu pour intégrer les meilleures pratiques actuelles en matière de technologies de modélisation et d'ingénierie logicielle.

Avant 1994, le monde des méthodes orientées objet était un peu en désordre. Il y avait plusieurs langages et méthodologies de modélisation visuelle concurrents, tous avec leurs forces et leurs faiblesses et tous avec leurs avantages et inconvénients. En termes de langages de modélisation visuelle, les leaders clairs étaient Booch (la méthode Booch) et Rumbaugh *(Object Modeling Technique ou OMT)*, qui à eux seuls détenaient plus de la moitié du marché. Du côté des méthodologies, Jacobson avait de loin le cas le plus fort car si de nombreux auteurs prétendaient avoir une «méthode», tout ce que beaucoup d'entre eux avaient réellement était une syntaxe de modélisation visuelle et un recueil de directives plus ou moins utiles.

En 1996, *l'Object Management Group (OMG)* a produit une demande de propositions *(RFP request For Proposition)* pour un langage de modélisation visuelle OO, et *UML* a été soumis. En 1997, *OMG* a accepté *l'UML* et le premier langage de modélisation OO ouvert et standard est né. Depuis lors, toutes les méthodes concurrentes ont disparu et *UML* est incontesté en tant que langage de modélisation OO standard de l'industrie.

UML a un large spectre d'utilisation. Il peut être utilisé pour la modélisation métier, la modélisation logicielle dans toutes les phases de développement et pour tous les types de systèmes, et la modélisation générale de toute construction ayant à la fois une structure statique et un comportement dynamique. Afin d'atteindre ces capacités étendues, le langage est défini comme étant suffisamment étendu et générique pour permettre la modélisation de systèmes aussi diversifiés, en évitant les systèmes trop spécialisés et trop complexes.

La présentation *d'UML* comprend les différentes parties suivantes:

- ***Views*** : les vues montrent différents aspects du système modélisé. Une vue n'est pas une sou forme graphique, mais une abstraction constituée d'un certain nombre de diagrammes. Ce n'est qu'en définissant un certain nombre de vues, chacune montrant un aspect particulier du système, qu'une image complète du système peut être construite. Les vues lient également le langage de modélisation à la méthode / processus choisi pour le développement.
- ***Diagrams*** : les diagrammes sont les formes graphiques qui décrivent le contenu d'une vue. *UML* a neuf types de diagrammes différents qui sont utilisés en combinaison pour fournir toutes les vues du système.
- ***Model elements*** : les concepts utilisés dans les diagrammes sont des éléments de modèle qui représentent des concepts orientés objet courants tels que les classes, les objets et les messages, ainsi que les relations entre ces concepts, notamment l'association, la dépendance et la généralisation. Un élément de modèle est utilisé dans plusieurs diagrammes différents, mais il a toujours la même signification et le même symbole.
- ***General mechanisms*** : les mécanismes généraux fournissent des commentaires, des informations ou une sémantique supplémentaire sur un élément de modèle; ils fournissent également des mécanismes d'extension pour adapter ou étendre l'UML à une méthode / processus.

2.1. Views

La modélisation d'un système complexe est une tâche extensive. Idéalement, l'ensemble du système est décrit dans un graphique unique qui définit l'ensemble du système sans ambiguïté, et facile à communiquer et à comprendre. Cependant, cela est

généralement impossible. Un seul graphique ne peut pas capturer toutes les informations nécessaires pour décrire un système. Un système est décrit avec un certain nombre d'aspects: fonctionnel (sa structure statique et ses interactions dynamiques), non fonctionnel (exigences de timing, fiabilité, déploiement, etc.) et organisationnel (organisation du travail, mappage vers des modules de code, etc.). Ainsi, un système est décrit dans un certain nombre de vues, où chaque vue représente une projection de la description complète du système, montrant un aspect particulier du système.

Les différentes *views* sont :

- Vue de cas d'utilisation: une vue montrant la fonctionnalité du système telle qu'elle est perçue par les acteurs externes.
- Vue logique: une vue montrant comment la fonctionnalité est conçue à l'intérieur du système, en termes de structure statique et de comportement dynamique du système.
- Vue des composants: une vue montrant l'organisation des composants du code.
- Vue de concurrence: une vue montrant la concurrence dans le système, résolvant les problèmes de communication et de synchronisation qui sont présents dans un système concurrent.
- Vue de déploiement: une vue montrant le déploiement du système dans l'architecture physique avec des ordinateurs et des périphériques appelés nœuds.

Vue de cas d'utilisation

La vue de cas d'utilisation décrit la fonctionnalité que le système doit offrir, telle que perçue par les acteurs externes. Un acteur interagit avec le système; ce peut être un utilisateur ou un autre système. La vue du cas utilisateur est destinée aux clients, concepteurs, développeurs et testeurs; il est décrit dans des diagrammes de cas d'utilisateurs et parfois dans des diagrammes d'activités. L'utilisation souhaitée du système est décrite comme un certain nombre de cas d'utilisation dans la vue des cas d'utilisation, où un cas d'utilisation est une description générale d'une utilisation du système

La vue de cas d'utilisation est centrale, car son contenu pilote le développement des autres vues. Le but final du système est de fournir la fonctionnalité décrite dans cette vue - avec certaines propriétés non fonctionnelles - par conséquent cette vue affecte toutes les autres.

Vue logique

La vue logique décrit comment la fonctionnalité du système est fournie. C'est principalement pour les concepteurs et les développeurs. Contrairement à la vue de cas d'utilisation, la vue logique regarde à l'intérieur du système. Il décrit à la fois les collaborations statiques (classes, objets et relations) et dynamiques qui se produisent lorsque les objets s'envoient des messages pour fournir une fonction donnée. Des propriétés telles que la persistance et la concurrence sont également définies, ainsi que les interfaces et les structures internes des classes.

La structure statique est décrite dans des diagrammes de classes et d'objets. La modélisation dynamique est décrite dans des diagrammes d'état, de séquence, de collaboration et d'activité.

Vue des composants

La vue des composants est une description des modules d'implémentation et de leurs dépendances. Il est principalement destiné aux développeurs et se compose du diagramme des composants. Les composants, qui sont différents types de modules de code, sont affichés avec leur structure et leurs dépendances. Des informations supplémentaires sur les composants, telles que l'allocation des ressources (responsabilité d'un composant), ou d'autres informations administratives, telles que le rapport d'avancement du travail de développement, peuvent également être ajoutées.

Vue de concurrence

La vue de concurrence traite de la division du système en processus et processeurs. Cet aspect, qui est une propriété non fonctionnelle du système, permet une utilisation efficace des ressources, une exécution parallèle et la gestion des événements asynchrones de l'environnement. En plus de diviser le système en threads de contrôle exécutant simultanément, la vue doit également gérer la communication et la synchronisation de ces threads.

La vue d'accès concurrentiel est destinée aux développeurs et intégrateurs du système et se compose de diagrammes dynamiques (diagrammes d'état, de séquence, de collaboration et d'activité) et de diagrammes d'implémentation (diagrammes de composants et de déploiement).

Vue de déploiement

La vue de déploiement montre le déploiement physique du système, tel que les ordinateurs et les périphériques (nœuds) et comment ils se connectent les uns aux autres. La vue de déploiement est destinée aux développeurs, aux intégrateurs et est représentée par le diagramme de déploiement. Cette vue comprend également un mappage qui montre comment les composants sont déployés dans l'architecture physique; par exemple, quel programme ou quels objets s'exécutent sur chaque ordinateur respectif.

2.2. Les diagrammes

Les diagrammes sont les graphiques réels qui montrent les symboles des éléments du modèle arrangés pour illustrer une partie ou un aspect particulier du système. Un modèle de système comporte généralement plusieurs diagrammes de chaque type. Un diagramme fait partie d'une vue spécifique; et quand il est dessiné, il est généralement attribué à une vue. Certains types de diagrammes peuvent faire partie de plusieurs vues, selon le contenu du diagramme.

Diagramme de Use case

Un diagramme de cas d'utilisation montre un certain nombre d'acteurs externes et leur connexion aux cas d'utilisation fournis par le système. Un cas d'utilisation est une description d'une fonctionnalité (une utilisation spécifique du système) fournie par le système. La description du cas d'utilisation réel est normalement faite en texte brut en tant que propriété de documentation du symbole de cas d'utilisation, mais elle peut également être décrite à l'aide d'un diagramme d'activité. Les cas d'utilisation sont décrits uniquement comme vu de l'extérieur par l'acteur (le comportement du système tel que l'utilisateur le perçoit), et ne décrivent pas comment la fonctionnalité est fournie à l'intérieur du système. Les cas d'utilisation définissent les exigences de fonctionnalité du système.

Diagramme de classes

Un diagramme de classes montre la structure statique des classes dans le système. Les classes représentent les «objets» qui sont gérées dans le système. Les classes peuvent être liées les unes aux autres de plusieurs manières: associées (connectées les unes aux autres), dépendantes (une classe dépend / utilise une autre classe), spécialisées (une classe est une spécialisation d'une autre classe) ou empaquetées (regroupées en une unité). Toutes ces relations sont présentées dans un diagramme de classes avec la structure interne des

classes en termes d'attributs et d'opérations. Le diagramme est considéré comme statique dans la mesure où la structure décrite est toujours valide à tout moment du cycle de vie du système. Un système a généralement un certain nombre de diagrammes de classes - toutes les classes ne sont pas insérées dans un seul diagramme de classes - et une classe peut participer à plusieurs diagrammes de classes.

Diagramme d'objet

Un diagramme d'objets est une variante d'un diagramme de classes et utilise une notation presque identique. La différence entre les deux est que le diagramme d'objets montre un certain nombre d'instances d'objets de classes, au lieu de classes réelles. Un diagramme d'objets est donc un exemple de diagramme de classes qui montre une capture instantanée possible de l'exécution du système - à quoi le système pourrait ressembler à un moment donné. La même notation que pour le diagramme de classes est utilisée, à deux exceptions près: les objets sont écrits avec leurs noms soulignés et toutes les instances dans une relation.

Les diagrammes d'objets ne sont pas aussi importants que les diagrammes de classes, mais ils peuvent être utilisés pour illustrer un diagramme de classes complexe en montrant à quoi pourraient ressembler les instances et relations réelles. Les diagrammes d'objets sont également utilisés dans le cadre de diagrammes de collaboration, dans lesquels la collaboration dynamique entre un ensemble d'objets est affichée.

Diagramme d'état transition

Un diagramme d'états est généralement un complément à la description d'une classe. Il montre tous les états possibles que les objets de la classe peuvent avoir, et quels événements provoquent le changement d'état. Un événement peut être un autre objet qui lui envoie un message - par exemple, qu'un temps spécifié s'est écoulé - ou qu'une condition a été remplie. Un changement d'état s'appelle une transition. Une transition peut également être associée à une action spécifiant ce qui doit être fait en relation avec la transition d'état.

Les diagrammes d'état ne sont pas dessinés pour toutes les classes, uniquement pour celles qui ont un certain nombre d'états bien définis et où le comportement de la classe est affecté et modifié par les différents états. Des diagrammes d'état peuvent également être dessinés pour le système dans son ensemble.

Diagramme de séquence

Un diagramme de séquence montre une collaboration dynamique entre un certain nombre d'objets. L'aspect important de ce diagramme est de montrer une séquence de messages envoyés entre les objets. Il montre également une interaction entre les objets, quelque chose qui se produira à un moment précis de l'exécution du système. Le diagramme se compose d'un certain nombre d'objets représentés par des lignes verticales. Le temps passe dans le diagramme et le diagramme montre l'échange de messages entre les objets au fil du temps dans la séquence ou la fonction. Les messages sont affichés sous forme de lignes avec des flèches de message entre les lignes d'objet verticales. Les spécifications de temps et autres commentaires sont ajoutés dans un script dans la marge du diagramme

Diagramme de collaboration

Un diagramme de collaboration montre une collaboration dynamique, tout comme le diagramme de séquence. Il s'agit souvent d'un choix entre montrer la collaboration sous forme de diagramme de séquence ou de diagramme de collaboration. En plus de montrer l'échange de messages (appelé interaction), le diagramme de collaboration montre les objets et leurs relations (parfois appelés le contexte). Utiliser un diagramme de séquence ou un diagramme de collaboration peut souvent être décidé par: Si le temps ou la séquence est l'aspect le plus important à souligner, on choisit le diagramme de séquence; s'il est important de souligner le contexte, on choisit le diagramme de collaboration. L'interaction entre les objets est illustrée dans les deux diagrammes.

Le diagramme de collaboration est dessiné sous la forme d'un diagramme d'objets, où un certain nombre d'objets sont affichés avec leurs relations (en utilisant la notation dans le diagramme de classes / objet). Des flèches de message sont dessinées entre les objets pour montrer le flux de messages entre les objets. Des étiquettes sont placées sur les messages, qui indiquent entre autres l'ordre dans lequel les messages sont envoyés. Il peut également afficher des collaborations, des itérations, des valeurs de retour, etc. Lorsqu'il est familiarisé avec la syntaxe des étiquettes de message, un développeur peut lire la collaboration et suivre le flux d'exécution et l'échange de messages. Un diagramme de collaboration peut également contenir des objets actifs, ceux qui s'exécutent simultanément avec d'autres objets actifs.

Diagramme d'activité

Un diagramme d'activités montre un flux séquentiel d'activités. Le diagramme d'activités est généralement utilisé pour décrire les activités exécutées dans une opération, bien qu'il puisse également être utilisé pour décrire d'autres flux d'activités, tels qu'un cas d'utilisation ou une interaction. Le diagramme d'activités se compose d'états d'action, qui contiennent une spécification d'une activité à effectuer (une action). Un état d'action quittera l'état lorsque l'action a été effectuée (un état dans un diagramme d'états à besoin d'un événement explicite avant de quitter l'état). Ainsi, le contrôle circule entre les états d'action, qui sont connectés les uns aux autres. Les décisions et conditions, ainsi que l'exécution parallèle des états d'action, peuvent également être affichées dans le diagramme. Le diagramme peut également contenir les spécifications des messages envoyés ou reçus dans le cadre des actions effectuées

Diagramme de composant

Un diagramme de composants montre la structure physique du code en termes de composants de code. Un composant peut être un composant de code source, un composant binaire ou un composant exécutable. Un composant contient des informations sur la ou les classes logiques qu'il implémente, créant ainsi un mappage de la vue logique à la vue du composant. Les dépendances entre les composants sont affichées, ce qui facilite l'analyse de la manière dont les autres composants sont affectés par une modification d'un composant. Les composants peuvent également être affichés avec l'une des interfaces qu'ils exposent et ils peuvent être regroupés dans des packages. Le diagramme des composants est utilisé dans les travaux pratiques de programmation.

Diagramme de déploiement

Le diagramme de déploiement montre l'architecture physique du matériel et des logiciels du système. On peut afficher les ordinateurs et les périphériques réels (nœuds), ainsi que les connexions qu'ils ont entre eux; on peut également afficher le type de connexions. À l'intérieur des nœuds, des composants exécutables et des objets sont alloués pour montrer quelles unités logicielles sont exécutées sur quels nœuds. On peut aussi afficher les dépendances entre les composants.

Le diagramme de déploiement décrit l'architecture réelle du système. Ceci est loin de la description fonctionnelle de la vue de cas d'utilisation. Cependant, avec un modèle bien défini, il est possible de naviguer d'un nœud dans l'architecture physique à ses

composants jusqu'à la classe à laquelle il implémente et auquel les objets de la classe participent et enfin à un cas d'utilisation. Différentes vues du système sont utilisées pour donner une description cohérente du système dans son ensemble.

2.3. Les éléments de modèle

Les concepts utilisés dans les diagrammes sont appelés éléments de modèle. Un élément de modèle est défini avec la sémantique, une définition formelle des éléments ou la signification exacte de ce qu'il représente sans ambiguïté. Un élément de modèle a également un élément de vue correspondant, qui est la représentation graphique de l'élément ou le symbole graphique utilisé pour représenter l'élément dans les diagrammes. Un élément peut exister dans plusieurs types de diagramme différents, mais il existe des règles pour lesquelles les éléments peuvent être affichés dans chaque type de diagramme. Certains exemples d'éléments de modèle sont la classe, l'objet, l'état, le nœud, le package et le composant.

Certaines relations différentes sont:

- Association: connecte les éléments et lie les instances.
- Généralisation: également appelé héritage, cela signifie qu'un élément peut être une spécialisation d'un autre élément.
- Dépendance: montre qu'un élément dépend d'une certaine manière d'un autre élément.
- Agrégation: forme d'association dans laquelle un élément contient un autre élément.
- D'autres éléments du modèle en plus de ceux décrits incluent les messages, les actions et les stéréotypes.

2. OCL

Le langage *OCL* est apparu comme un effort pour surmonter les limites *d'UML* lorsqu'il s'agit de spécifier précisément les aspects détaillés de la conception d'un système. *OCL* a été développé pour la première fois en 1995 chez *IBM* comme une évolution d'un langage d'expression dans la méthode Syntropy. Le travail sur *OCL* faisait partie d'une proposition conjointe avec *ObjectTime Limited* présentée en réponse à la demande de propositions pour une analyse standard orientée objet et un langage de conception publié par *l'Object Management Group* (*OMG*). Cette norme est devenue ce que nous appelons maintenant *UML* et *OCL* y a été intégrée en 1997.

Initialement, *OCL* n'était utilisé que comme langage de contrainte pour *UML* mais a rapidement élargi sa portée et maintenant *OCL* est devenu un composant clé de toute technique d'ingénierie pilotée par modèle *(MDE)* en tant que langage par défaut pour exprimer toutes sortes de requêtes de (méta) modèles, manipulation et exigences de spécification. Parmi de nombreuses autres applications, *OCL* est fréquemment utilisé pour exprimer des transformations de modèle (dans le cadre des modèles source et cible des règles de transformation), des règles de bonne formation (dans le cadre de la définition de nouveaux langages spécifiques au domaine, ou des modèles de génération de code (comme un moyen d'exprimer les modèles et les règles de génération). Pour adapter le langage à ces nouvelles applications, plusieurs nouvelles (sous) versions du langage ont été publiées.

2.4. Pourquoi utiliser OCL ?

Les langages de modélisation graphique sont le choix préféré de nombreux concepteurs lorsqu'il s'agit de définir les aspects structurels d'un domaine (c'est-à-dire ses principaux concepts, leurs propriétés et les relations entre eux). L'exemple le plus typique de notation graphique est *UML*, en particulier son diagramme de classes qui est de loin le diagramme *UML* le plus utilisé.

Néanmoins, cette facilité d'utilisation a un prix. Afin de garder le nombre d'éléments de notation gérable, les concepteurs de langage doivent limiter l'expressivité du langage. Cela signifie que les notations graphiques ne peuvent exprimer qu'un sous-ensemble limité de toutes les informations pertinentes d'un domaine. C'est là qu'intervient *OCL* (et en général tout autre langage textuel). Ils sont un complément nécessaire de la notation *UML* (ou d'autres langages graphiques) afin de pouvoir spécifier avec précision tous les aspects détaillés de la conception d'un système.

2.5. Définition des contraintes OCL

Une contrainte *OCL* peut se rapporter à une classe, appelée le contexte de la contrainte. Trois stéréotypes *OCL* sont applicables à une contrainte :

- *inv* pour la spécification d'invariants, d'une condition qui doit être vraie pour chaque instance de la classe en permanence.

- *pre* pour la spécification de préconditions des opérations, d'expressions qui permettent de spécifier des contraintes qui doivent être vérifiées avant l'appel d'une opération.
- *post* pour la spécification de postconditions des opérations, d'expressions qui permettent de spécifier des contraintes qui doivent être vérifiées après l'exécution d'une opération.

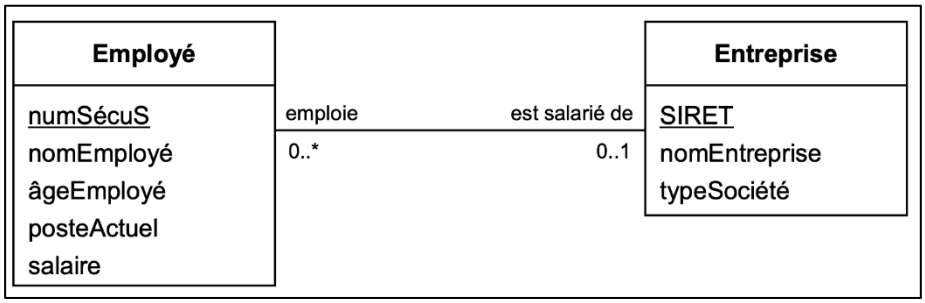


Figure 12: Exemple de diagramme de classes pour appliquer OCL

La contrainte *OCL* suivante exprime le fait que tous les employés dont le poste actuel est *professeur* perçoivent un salaire supérieur ou égal à 12000.

```
1   context Employé inv:
2   self.posteActuel = 'professeur' Implies salaire >= 12000
3
```

Figure 13: Exemple de sélection des employés avec un salaire supérieur ou égal 12000

self référence une instance du contexte, l'élément du modèle sur lequel porte la contrainte (ici une instance de la classe Employé). *implies* indique que la vérification de la condition sur la valeur de l'attribut *posteActuel* implique une contrainte sur la valeur de l'attribut salaire qui devra alors être supérieur ou égal à 12000.

- Navigation entre les classes et opération sur les collections

La navigation permet, à partir d'un objet spécifique, de se déplacer dans le modèle. Par exemple, la contrainte suivante signifie que les petites et moyennes entreprises (PME) n'ont pas plus de 350 salariés.

```
1   context Entreprise inv:
2   self.typeSociété = 'PME' Implies self.Employé -> size() <= 350
3
```

Figure 14: Exemple de sélection des entreprises avec plus de 350 employés

La navigation via une association se fait à l'aide de la notation pointée. Ainsi, *self.Employé* retourne la collection des instances de la classe *Employé* accessibles par l'association depuis l'instance *self* de la classe Entreprise. L'opération *size()* retourne la taille d'une collection.

Dans une spécification de contrainte OCL, les noms de rôle des associations peuvent aussi permettre la navigation. Par exemple, il est aussi possible d'écrire la contrainte précédente de la façon suivante ("Employé" a été remplacé par "emploie").

```
1   context Entreprise inv:
2   self.typeSociété = 'PME' Implies self.emploie -> size() <= 350
3
```

Figure 15: Exemple de sélection des entreprises employés en utilisant le rôle de l'association

D'autres opérations existent sur les collections. Par exemple, *size(Objet)* retourne le nombre d'occurrences d'Objet dans *self*. L'opération *isEmpty()* retourne vrai si *self* est vide.

- Opération sur les éléments d'une collection

La syntaxe d'une opération portant sur les éléments d'une collection est la suivante

```
1   <colléction> -> <opération> ( [ <itérateur> | ] <expression> )
2
```

Figure 16: Définition générique des opérations appliquées aux collections

<*opération*> est l'une des opérations suivantes :

exists représente le quantificateur existentiel qui vérifie si une expression booléenne est vraie pour au moins un élément de <*collection*>. Par exemple, on peut exprimer le fait qu'il existe (un où) plusieurs chercheurs dans un établissement de recherche.

```
1   context Entreprise inv:
2   self.type='Faculté des sciences'
3   implies self.Employé->exists(poste_actuel='professeur')
4
```

Figure 17: Exemple d'utilisation de l'opération exist

forAll représente le quantificateur universel qui vérifie si une expression booléenne est vraie pour toutes les instances de <*collection*>.

Par exemple, on peut spécifier le fait que tous les employés des établissements de recherche ont plus de 18 ans.

```
1   context Entreprise inv:
2   self.type='Faculté des sciences'
3   Implies self.Employé->forAll(age>=18)
4
```

Figure 18: Exemple d'utilisation de l'opération forAll

collect retourne une nouvelle collection de même taille mais dont les éléments sont généralement d'un type diffèrent que ceux de <*collection*>.

Par exemple, on peut l'utiliser pour collecter dans une contrainte, tous les postes des employés d'une société :

```
1    self.Employé -> collect (poste_actuel)
2
```

Figure 19: Exemple d'utilisation de l'opération collect

size retourne le nombre d'éléments d'une collection.

select (resp. reject) retourne un sous ensemble de <*collection*> qui ne contient que les éléments qui vérifient (resp. ne vérifient pas) l'<expression>. Par exemple, on peut l'utiliser pour sélectionner dans une contrainte, les employés majeurs :

```
1    self.Employé -> select (age>=18)
2
```

Figure 20: Exemple d'utilisation de l'opération select

On utilise *allInstances* pour appliquer une opération à toutes les instances d'une classe. *Employé.allInstances* retourne la collection de toutes les instances d'employés.

Des opérations sur les types d'OCL sont aussi prédéfinies, notamment *oclIsTypeOf(t)*.
Par exemple, *o.oclIsTypeOf(t)* retourne vrai si le type de t et le type de l'objet o à laquelle l'opération est appliquée sont les mêmes.

Chapitre 2 : Les Travaux Connexes

I. Aperçu sur les travaux connexes

1. ORDB Vers UML/OCL

Le reverse engineering ou la rétro conception des bases de données est un sujet bien étudié. Une grande partie des recherches menées dans ce domaine s'est concentrée sur le modèle relationnel. Plusieurs approches ont été proposées pour extraire un schéma conceptuel d'une base de données relationnelle. [2] adopte la notation *OMT (Object Modeling Technique)* pour modéliser les données à partir d'une base de données en cours d'exécution, [3] proposent une nouvelle méthodologie pour extraire une *Entiy-Relation étendue (EER)* à partir d'une base de données relationnelle *(RDB),* [4] pour extraire un concept-schéma de *RDB,* l'auteur présente une méthode basée sur l'analyse des déclarations de manipulation de données dans le code d'une application à l'aide d'un schéma de base de données relationnelle, [5] mappe un schéma relationnel dans un schéma orienté objet en prenant en compte différents types de conception optimisés des *RDBs*. [6] présente une méthode pour traduire une base de données relationnelle en un modèle de relation d'objet *(ORM),* [7] montrent comment la notion de vue de base de données relationnelle peut être correctement exprimée en tant que classe dérivée en *UML / OCL,* [8] présente un méthode pour définir *OCL* comme langage de requête pour les modèles de données *UML,* [9] présentent une approche pour extraire automatiquement des règles métiers structurelles à partir d'une base de données et une application sur un système spécifique, [10] présente une approche de reverse engineering de base de données qui prend en charge diagramme d'entité-relation d'une base de données classique basée sur des tables, le travail dans [11] aborde le reverse engineering d'une base de données en récupérant le schéma étendu de relation entité-relation à partir d'un schéma relationnel, [12] présente une approche de reverse engineering basée sur un modèle capable d'extraire un schéma conceptuel exprimé sous la forme d'un diagramme de classes UML étendu avec un ensemble *d'OCL* d'une base de données relationnelle.

Pour le *Forward enginerring* des bases de données objet relationnelle *(ORDB),* [13] présente une méthode qui définit de nouveaux éléments de modèle *UML* pour concevoir la base de données objet relationnelle, [14] décrit une méthode de transformation de

modèles UML, la méthode contient deux phases; la première présente la transformation du diagramme de classes (aspect statique) en schéma de base de données, la deuxième transforme le diagramme d'état transition (aspect dynamique) en déclencheurs de base de données. Pour le reverse engineering, [15] décrit une approche pour récupérer les schémas depuis *ORDB*, l'objectif principal de cette approche est de récupérer des schémas conceptuels, représentés sous forme de diagrammes *UML*, à partir de l'analyse du dictionnaire de données. En comparaison, notre approche présente une certaine similarité dans la phase d'extraction du modèle avec [15].

2. XML & XQuery vers OCL

UML est devenu la partie principale du développement logiciel, y compris les applications Web qui utilisent XML pour échanger des données structurées. C'est pourquoi il est nécessaire de modéliser des éléments *XML* avec *UML*.

La rétro conception ou le reverse engineering nous permet d'obtenir un schéma conceptuel qui aide les développeurs à comprendre les systèmes et à en faciliter la maintenance.

De nombreuses méthodes de mappage de schémas *XML* se concentrent uniquement sur l'obtention de la partie structurelle (éléments, types complexes et attributs ...) sans accorder d'importance aux contraintes et restrictions ainsi qu'aux requêtes XML.

XML schéma présente la partie statique de *XML*, de sorte que le diagramme *UML* qui a les mêmes propriétés, et statique en même temps, est le diagramme de classe, ici nous pouvons mapper et transformer des éléments et des attributs *XML*, en classes et propriétés avec les mêmes caractéristiques (type de données, relation, généralisation,…), au niveau conceptuel, il est nécessaire de générer des modèles conceptuels pour illustrer la structure et les relations des données dans le schéma *XML* [17,18]. Ici, beaucoup de travail a été fait pour représenter différentes approches et méthodes de mappage entre le XML schéma et *UML*.

[19,20] présentent une formalisation des règles de transformation du *XML* schéma au diagramme de classes *UML*, les auteurs de [21] proposent une méthode de rétro conception à partir de *XML* pour générer une spécification d'exigence logicielle *(Software Requirement Specification SRS)* de manière documentaire. Dans [22], l'auteur étend *UML* et ajoute quelques stéréotypes pour décrire le schéma *SOX* utilisé par les éléments de

commerceOne. Dans [23] *Bird, Goodchild et Halpin* ont proposé une approche qui utilise un langage conceptuel de modèle d'objet de rôle pour générer un schéma *XML*. Dans [24] les auteurs ont présenté une solution pour mapper le modèle orienté objet *UML* au *XML* schéma, puis au schéma de table de base de données relationnelle, [25] fournissent un cadre d'évolution par lequel le schéma et les documents *XML* sont mis à jour de manière incrémentielle en fonction des changements le modèle conceptuel (exprimé sous forme de diagramme de classes *UML*), dans [26] les auteurs présentent un ensemble de règles utilisant des feuilles de style *XSLT* qui transforment le diagramme de classes UML en une définition de type de document adéquate *(DTD),* dans [27] les auteurs présentent la correspondance règles de génération de *DTD* à partir de diagrammes *UML*. De l'autre côté, les auteurs dans [28] ont développé un algorithme pour construire un diagramme de classes *UML* à partir de *XML* Schéma, [29] fournit une comparaison de plusieurs approches qui créent automatiquement un modèle spécifique à la plate-forme pour les *XML* schéma, basé sur un ensemble complet de modèles de transformation prenant en charge la création d'un modèle *UML* qui est aussi concis et sémantiquement expressif sans perdre les informations du *XML* schéma, les auteurs de [30] présentent la transformation de méthode du *XML* schéma textuel en schéma *UML* graphique pour faciliter la compréhension du schéma *XML*.

Les documents *XML* étant considérés comme une source d'informations ou une base de données, il est nécessaire d'obtenir des données à partir de cette source. *XQuery* est à *XML* ce que *SQL* est à la base de données. *XQuery* est le langage standard du W3C pour récupérer des données à partir d'un document *XML*.

Tous les travaux précédents se concentrent uniquement sur la partie structurelle sans prendre en considération la transformation des composants de restriction ou ce que nous appelons dans le *XML* schéma les *facets*. La même chose pour la transformation et le mappage des requêtes *XML*, la plupart des recherches négligent cette partie.

3. ORDB vers NoSQL

Pour effectuer une migration complète, plusieurs méthodes ont été appliquées, qui se concentrent sur trois classes : conception et modélisation de bases de données *NoSQL*, conversion de schémas et migration de données.

Avec l'évolution des bases de données *NoSQL*, plusieurs outils ont été produits pour migrer les données de la base de données relationnelle vers *NoSQL* comme *Pentaho Data Integration (Kettle)* [33] et *Apache Sqoop* [32].

Kettle est un moteur d'extraction, de transformation et de chargement (*ETL*) qui utilise une approche axée sur les métadonnées. Ces outils utilisent des composants graphiques qui guident les utilisateurs pour modéliser, préparer et récupérer des ensembles de données non structurés dans une base de données *NoSQL* telle que *MongoDB*, *Cassandra* et *Hadoop*. La tâche principale *d'ETL* consiste uniquement à migrer les données, ce qui oblige l'utilisateur à modéliser les règles de mappage entre les bases de données.

Apache Sqoop est un outil développé pour migrer les données entre les sources de données structurés tels que les systèmes de bases de données relationnelles et *Apache Hadoop*. Il est utilisé pour (importer, exporter) des données (de, vers) des bases de données relationnelles telles que *Mysql*, *Oracle* et *Hadoopfile*.

Wu-Chun Chung et al, ont développé un *framework* appelé *JackHare* utilisant le pilote *JDBC*, et contenant un compilateur de requêtes SQL et une méthode systématique basée sur *HBase* et *Hadoop* pour stocker des données dans une base de données relationnelle à l'aide de *MapReduce* pour gérer les données non structurées. Cette solution a été développée pour traiter une quantité massive de données [34].

Chongxin Li a présenté une approche basée sur l'heuristique pour transformer une base de données en *HBase* en utilisant trois directives pour effectuer la conversion de schéma. Les trois directives couvrent le regroupement des données corrélées dans une famille de colonnes, l'utilisation de références de clé étrangère de l'autre côté s'il a besoin d'accéder aux informations de l'autre côté et enfin la fusion des tables de données concernées pour réduire les clés étrangères [35].

R Lawrence a introduit une interface de requête *SQL* générique pour *NoSQL* et les systèmes relationnels intitulée *Unity*. *Unity* est un système d'intégration et de virtualisation permettant aux utilisateurs d'extraire les données de bases de données relationnelles et *NoSQL* en une seule requête, la virtualisation représente le pont qui mappe les requêtes *SQL* aux *API NoSQL* et exécute automatiquement les opérations non prises en charge par les bases de données *NoSQL* [36].

Tianyu Jia et al, ont présenté une approche de transformation de modèle et de migration de données relationnelle de base de données vers la base de données orientée document *NoSQL MongoDB*. Pour éviter la redondance des données et améliorer les performances, l'algorithme de transformation proposé n'optimise qu'une table spécifique en utilisant des balises d'action pour décrire la limitation du schéma relationnel [37].

G Zhao et coll. ont introduit un modèle relationnel pour *MongoDB* et utilisé l'opération d'extraction de l'algèbre relationnelle, y compris la situation de partitionnement multiple et unique, bien que *MongoDB* soit une base de données de documents *NoSQL* sans schéma préfixé. De plus, basé sur le modèle relationnel, *MongoDB* peut prendre en charge le calcul relationnel comme le système de gestion de base de données relationnelle. Par conséquent, ils ont supposé que théoriquement la migration des données de la base de données relationnelle vers *MongoDB* pouvait être effectuée en toute sécurité et facilement [38]. En utilisant les documents intégrés ou référencés dans *MongoDB,* nous pouvons modéliser les relations entre différents documents, *A Kanade et al,* ils ont présenté la différence de performance ainsi que la modification de la technique de modélisation, et ils ont présenté plusieurs expériences pour trouver le point optimal de normalisation et de liaison entre les documents, pour réduire le temps d'exécution des requêtes dans *MongoDB*.

En résumé, certaines méthodes et approches proposées se concentrent sur la migration des données, tandis que d'autres prennent en compte la partie de la conversion de schéma de la base de données relationnelle vers *NoSQL*, et l'amélioration des performances de cette opération pour atteindre une haute disponibilité. Mais aucune recherche n'a été proposée pour convertir le schéma ou migrer les données de la base de données objet relationnelle vers les bases de données *NoSQL*.

PARTIE 2 :

CONTRIBUTION SCIENTIFIQUE

Chapitre 3 : ORDB Vers UML/OCL

I. Introduction

Le reverse engineering ou la rétro-conception est un ensemble des techniques et des outils liés à la compréhension d'un système logiciel ou matériel existant. Son rôle principal est de récupérer le modèle de conception afin de régénérer les différents modèles d'abstraction à partir de la source d'information (code source d'un programme, schéma physique d'une base de données, une application mobile, documentation, …).

En se basant sur deux étapes essentielles, le processus de la rétro-conception commence par l'identification des composants système, les différentes entités ainsi que leurs relations et contraintes, et par la suite la transformation du résultat obtenu dans un niveau très élevé d'abstraction et plus compréhensible par l'utilisateur.

Dans le cas des sources des données et pour identifier les composants, la rétro-conception effectue une étude et une analyse approfondie de la structure et de contenu afin de générer des composants conceptuels (entités et associations entre les entités) serviront à reproduire le modèle conceptuel qui va générer un schéma compréhensible en langage naturel de la source de données.

La démarche normale du reverse engineering se base sur l'inversion des règles du passage de la couche conceptuelle au couche physique ce qui doit produire comme résultat un modèle conceptuel de données. Dans ce chapitre nous introduisons les règles de transformation qui nous aident à générer un méta-model à partir du schéma physique de la base de données ceci dit, le modèle source représente le schéma de la source de données et le modèle cible est le modèle conceptuel équivalant.

Pour avoir un système d'information solide et fiable, il est indispensable d'assurer la cohérence et l'intégrité de la base de données. Le terme d'intégrité gère plusieurs aspects, sémantique responsable de la qualité des données stockées et technique comme la sécurité des données, le contrôle de la cohérence. Les règles d'intégrités sont des contraintes exprimées au niveau du modèle conceptuel et implémenté au niveau physique par la suite. Ces règles sémantiques représentent les contraintes d'intégrité. Alors on peut dire qu'une base de données est fiable et cohérente lorsque l'ensemble des contraintes d'intégrité est respecté par tous les enregistrements de la base.

La rétro-conception des bases de données est le processus de génération d'une description de contenu des composants de la base, à un haut niveau en utilisant un langage naturel et compréhensible par les utilisateurs. Le processus produit un schéma exprimé en notation de modélisation conceptuelle.

En utilisant un langage humain indépendant de n'importe quelle implémentation physique et en respectant les critères de la clarté et la simplicité, le schéma conceptuel représente une définition abstraite des composants de la base de données (les tables avec leurs relations). Le schéma conceptuel obtenu facilite la compréhension de la structure des données ainsi que les règles de gestion implémentées sous forme des contraintes d'intégrité et des triggers. Le schéma conceptuel peut aussi aider à l'intégration, l'évolution, la migration des données et la réutilisation du système.

Les grands leaders des systèmes de gestion de base de données comme *Oracle*, *Microsoft* et *IBM* ont intégré les différents concepts de l'orienté objet (encapsulation, héritage, utilisation des classes et des méthodes, …) aux leurs *SGBD* Relationnel pour gagner le défi de représentation des données complexes. En conséquence, plusieurs sociétés utilisent les solutions hybrides pour le développement des bases de données qui adoptent les concepts de l'orienté objet.

Avant de faire évoluer ou maintenir un système d'information et pour assurer que ce changement s'est bien déroulé, il est nécessaire d'avoir une idée globale sur la structure de la base de données, et pour atteindre ce but, il est primordial de décrire les tables avec les règles de gestion sous forme d'une représentation facile et homogène à un haut niveau d'abstraction.

Notre but dans ce chapitre est de rétro concevoir le schéma physique d'une base de données objet relationnel, afin de reproduire le modèle conceptuel de données sous format d'un diagramme de classes *UML* enrichi avec des contraintes *OCL* tout en assurant la préservation de la sémantique par le biais de la transformation des différentes entités et associations ainsi que les contraintes que ça soit des contraintes d'intégrité référentiel ou des contraintes gérées par les triggers.

Le reste de ce chapitre est organisé comme suit :

- La section 2 présente un aperçu général sur l'approche proposée de rétro-conception d'une base de données objet relationnel à un schéma conceptuel exprimé en diagramme de classes *UML* et enrichi avec des invariants *OCL*.

- La section 3 présente une description détaillée de la première phase de notre méthode de rétro-conception. Dans cette section on discute les étapes de l'extraction de modèle en commençant par la récupération des informations nécessaires du dictionnaire de la base de données Oracle 12c et par la suite on présente nos règles de transformations.
- La section 4 présente une description détaillée de la deuxième phase de notre méthode de rétro-conception. Après la récupération des différentes entités de la base de données ainsi que les relations et les associations, on récupère aussi les différentes règles de gestions définies au niveau physique de la base sous forme des contraintes d'intégrités référentielles et des déclencheurs et les transformer en invariants *OCL* pour compléter notre diagramme de classes *UML* obtenu dans la première phase.
- La section 5 expose le produit final de notre contribution. En effet, nous présentons une application de reverse engineering d'une base de données objet relationnel développé en java pour valider notre approche proposée.

II. Aperçu général de l'approche proposée

Notre méthode de rétro-conception contient deux phases essentielles :

- Le rôle principal de la première est l'extraction de modèle. Dans cette étape on se focalise sur la partie structurale du schéma objet relationnel, ou on récupère toutes les informations nécessaires à partir du dictionnaire de la base de données comme les types objets crées par les utilisateurs, et on vérifie aussi si le type récupéré s'il est *instantiable,* permet de créer des nouvelles instances, *final,* permet d'avoir des sous types, *super* s'il a un type parent etc…, toujours en prenant considération si les attributs membres de l'objet sont de type primitif (*char*, *varchar*, *date*, …) ou de type complexe. Dans le cas où l'attribut de type complexe on vérifie s'il est un objet ou une référence, ou bien une collection statique *varray* avec un nombre d'élément limité, ou une collection dynamique *nested table* qui peut avoir un nombre illimité des objets.
- Une base de données sans règles de gestion qui contrôlent les données enregistrées risque de perdre la fiabilité et la cohérence des données. Pour maintenir ou faire évaluer une base de données, il est indispensable de

savoir toutes les règles implémentées au niveau de la base que ça soit des contraintes d'intégrité définies au niveau des tables ou bien géré sous forme des *triggers* ou déclencheurs. La deuxième phase de notre méthode de rétro-conception représente l'extraction des contraintes, ici on se concentre sur la partie comportement des objets de la base de données. Cette phase contient deux étapes, dans la première étape on présente les règles de transformation des contraintes d'intégrité comme (*Primary key, Unique, Check, …*) à des expressions *OCL* pour renforcer le diagramme de classes *UML* obtenu auparavant. Les déclencheurs jouent un rôle principal pour définir des complexes règles de gestion et renforcer les contraintes traditionnelles, par définition les déclencheurs regroupent les requêtes *SQL* avec un langage procédural (dans notre cas *PL/SQL*), en conséquence nous proposons des règles pour transformer *(SQL, PL/SQL)* à des expressions *OCL* pour gagner le défi de présenter les complexes règles de gestion en langage naturel et compréhensible.

La figure ci –dessous décrit notre approche de rétro-conception en présentant les deux phases de transformations.

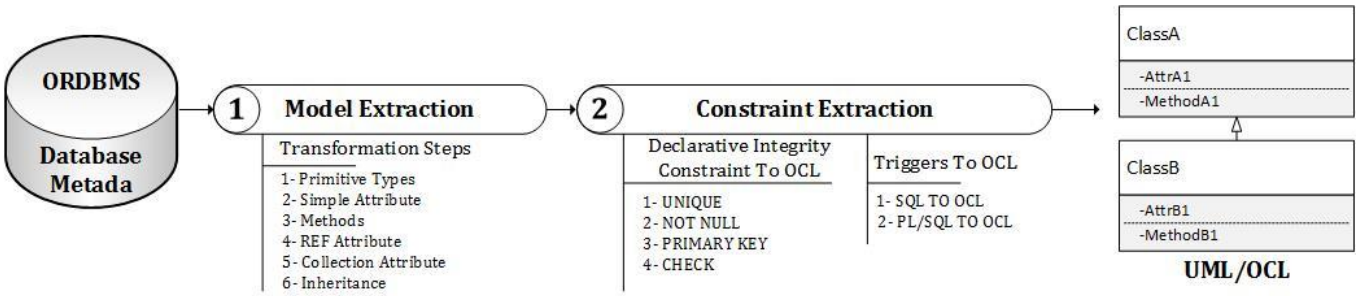


Figure 21: Approche proposée

Avant de présenter en détail la première phase de notre méthode de rétro-conception, on présente une description des objets Oracle qui représente l'élément essentiel d'analyse et de transformation dans notre méthode.

1. Les Objets Oracle

Les types d'objets *Oracle* sont des types définis par l'utilisateur qui permettent de modéliser des entités du monde réel telles que des clients et des commandes en tant qu'objets dans la base de données.

La technologie objet d'Oracle est une couche d'abstraction basée sur la technologie relationnelle *d'Oracle*. Avec cette technologie, nouveaux types d'objet peuvent être crées à partir de n'importe quel type primitif ou d'un objet, référence d'objet ou une collection précédemment crées. Les métadonnées pour les types définis par l'utilisateur sont stockées dans un schéma disponible pour *SQL*, *PL / SQL*, *Java* et d'autres interfaces.

Les types d'objet et les fonctionnalités orientées objet associées, telles que les tableaux de longueur variable et les tables imbriquées, fournissent des méthodes de haut niveau pour organiser et accéder aux données dans la base de données. Sous la couche objet, les données sont toujours stockées dans des colonnes et des tableaux, mais on peut travailler avec les données en termes d'entités réelles, telles que les clients et les commandes, qui rendent les données significatives. Au lieu de penser en termes de colonnes et de tableaux lorsqu'on interroge la base de données, on peut tout simplement sélectionner un client avec ses commandes sans penser à faire des jointures et des requêtes complexes.

En interne, les instructions sur les objets sont toujours essentiellement des instructions sur les tables et les colonnes relationnelles, et on peut continuer à travailler avec les types de données relationnelles et stocker les données dans les tables relationnelles comme auparavant. Mais maintenant, nous avons la possibilité de commencer à utiliser des fonctionnalités orientées objet tout en continuant à travailler avec la plupart des données de manière relationnelle, en passant entièrement à une approche orientée objet. Par exemple, définir certains types de données d'objet et stocker les objets dans des colonnes dans des tables relationnelles, ce qui permet d'étendre les types primitifs du système avec des types définis par l'utilisateur. Avec les aspects de l'orienté objets intégrés dans le *SGBD Oracle* le stockage des données d'objet se fait dans des tables d'objets, où chaque ligne est un objet.

Les avantages des objets

En général, le modèle de type objet est similaire au mécanisme de classe trouvé en *C ++ et Java*. Comme les classes, les objets facilitent la modélisation des entités métier et

de la logique complexe du monde réel, et aussi la réutilisation des objets permet de développer des applications de base de données d'une manière rapide et efficace. En prenant en charge nativement les types d'objets dans la base de données, Oracle permet aux développeurs d'applications d'accéder directement aux structures de données utilisées par leurs applications. Aucune couche de mappage n'est requise entre les objets côté client et les colonnes et tables de la base de données relationnelle qui contiennent les données. L'abstraction d'objets et l'encapsulation de comportements d'objets facilitent également la compréhension et la maintenance des applications.

Ci-dessous on présente plusieurs autres avantages spécifiques que les objets offrent par rapport à une approche purement relationnelle.

Les objets peuvent encapsuler les opérations avec des données

Les tables de base de données contiennent uniquement des données. Les objets peuvent inclure la possibilité d'effectuer des opérations susceptibles d'être nécessaires sur ces données. Ainsi, par exemple un objet de bon de commande peut inclure une méthode pour additionner le coût de tous les articles achetés. Ou un objet client peut avoir des méthodes pour renvoyer l'historique d'achat et le modèle de paiement du client. Une application peut simplement appeler les méthodes pour récupérer les informations.

Les objets sont efficaces

L'utilisation des types d'objets permet une plus grande efficacité :

- Les types d'objets et leurs méthodes sont stockés avec les données dans la base de données, ils sont donc disponibles pour toute application à utiliser. Les développeurs peuvent bénéficier du travail déjà effectué et n'ont pas besoin de recréer des structures similaires dans chaque application.
- On peut extraire et manipuler un ensemble d'objets associés comme une seule unité. Une seule demande pour extraire un objet du serveur peut récupérer d'autres objets qui lui sont connectés. Par exemple, lorsque on sélectionne un objet client et obtenez le nom du client, son téléphone et les multiples parties de son adresse en un seul aller-retour entre le client et le serveur. Lorsqu'on référence une colonne d'un type d'objet *SQL*, on récupère l'intégralité de l'objet.

Les objets peuvent représenter les relations d'agrégation et de composition

Dans un système relationnel, il est difficile de représenter des relations complexes d'agrégation ou de composition. Par exemple un piston et un moteur ont le même statut dans un tableau pour les articles en stock. Pour représenter les pistons en tant que parties de moteurs, on doit créer des schémas complexes de plusieurs tables avec des relations clé primaire et étrangère. Les types d'objets, en revanche, offrent un vocabulaire riche pour décrire les relations partielles. Un objet peut avoir d'autres objets en tant qu'attributs et les objets d'attribut peuvent également avoir leurs propres attributs d'objet. Une hiérarchie complète de listes de pièces peut être créée de cette manière à partir de types d'objets assemblés.

2. Les caractéristiques clés du modèle objet-relationnel

Oracle implémente le système de type objet comme une extension du modèle relationnel. L'interface de type objet continue de prendre en charge les fonctionnalités de base de données relationnelles standard telles que les requêtes *(SELECT… FROM… WHERE),* les validations rapides, la sauvegarde et la récupération, la connectivité évolutive, le verrouillage au niveau des lignes, la cohérence en lecture, les tables partitionnées, les requêtes parallèles, exportation et importation.

Plus *SQL* et diverses interfaces de programmation avec Oracle; y compris *PL / SQL, Java, Oracle Call Interface*; ont été améliorés avec de nouvelles extensions pour prendre en charge les objets. Le résultat est un modèle objet-relationnel, qui offre l'intuitivité et l'économie d'une interface objet tout en préservant la concurrence et le débit élevés d'une base de données relationnelle.

Cette section liste les principales fonctionnalités et concepts du modèle relationnel-objet liés à la base de données. La figure 22 montre un type d'objet et des instances de l'objet.

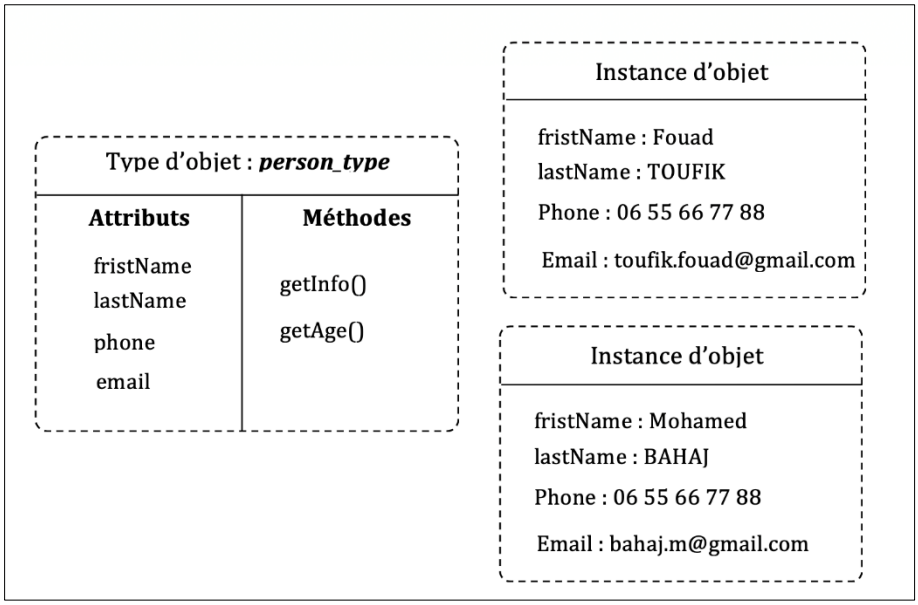


Figure 22: Type d'objet utilisateur Oracle 12c

Un type d'objet est une sorte de type de données. On peut l'utiliser de la même manière qu'on utilise des types de données plus familiers tels que *NUMBER* ou *VARCHAR2*. Par exemple, on peut spécifier un type d'objet comme type de données d'une colonne dans une table relationnelle et aussi déclarer des variables d'un type d'objet. Une variable d'un type d'objet contient une valeur de ce type d'objet. Une valeur d'un type d'objet est une instance de ce type et une instance d'objet est également appelée un objet.

La figure 23 montre comment créer un type d'objet nommé *person_type*. Dans l'exemple, une spécification d'objet et un corps d'objet sont définis :

```
1  CREATE TYPE person_type AS OBJECT (
2    idno           NUMBER,
3    first_name     VARCHAR2(20),
4    last_name      VARCHAR2(25),
5    email          VARCHAR2(25),
6    phone          VARCHAR2(20),
7    MAP MEMBER FUNCTION get_idno RETURN NUMBER,
8    MEMBER PROCEDURE getInfo ( SELF IN OUT NOCOPY person_type ));
9  /
10
11 CREATE TYPE BODY person_type AS
12   MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
13   BEGIN
14     RETURN idno;
15   END;
16   MEMBER PROCEDURE getIndo ( SELF IN OUT NOCOPY person_type ) IS
17   BEGIN
18     DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' || first_name || ' ' || last_name);
19     DBMS_OUTPUT.PUT_LINE(email || ' '  || phone);
20   END;
21 END;
22 /
```

Figure 23: Exemple de création du type d'objet person_type

Un ensemble de types d'objets n'est pas fourni avec la base de données. Au lieu de cela, on définit les types d'objets spécifiques en étendant les types intégrés avec ceux définis par l'utilisateur, comme indiqué dans l'exemple précédent.

Les types d'objets sont composés de deux parties, appelées attributs et méthodes :

- Par exemple, un type d'objet étudiant peut avoir des attributs comme nom, prénom et date d'obtention du diplôme. Un attribut a un type de données déclaré qui peut à son tour être un autre type d'objet. Pris l'ensemble, les attributs d'une instance d'objet contiennent les données de cet objet.
- Les méthodes sont des procédures ou des fonctions fournies pour permettre aux applications d'effectuer des opérations utiles sur les attributs du type d'objet. Les méthodes sont un élément facultatif d'un type d'objet. Ils définissent le comportement des objets de ce type et déterminent ce que (le cas échéant) ce type d'objet peut faire.

Lors de la création d'une variable d'un type objet, le *SGBD* crée une instance du type choisi et le résultat est un objet. Un objet contient des attributs et des méthodes définis pour son type. Parce qu'une instance d'objet est une chose concrète, une fois instanciée on peut affecter des valeurs à ses attributs et appeler ses méthodes.

On utilise l'instruction *CREATE TYPE* pour définir les types d'objet. Dans la figure 23, l'instruction *CREATE TYPE* définit le type d'objet *person_type*.

Les éléments *idno*, *email* et *phone* dans l'instruction *CREATE TYPE* sont des attributs. Chacun a un type de données déclaré.

La définition d'un type d'objet n'alloue aucun stockage. Une fois définis, les types d'objet peuvent être utilisés dans des instructions SQL dans la plupart des mêmes endroits où on utilise des types tels que *NUMBER* ou *VARCHAR2*.

Par exemple, la figure 24 montres la définition d'une table relationnelle pour garder l'historique des différents contacts.

```
1  CREATE TABLE contacts (
2    contact         person_type,
3    contact_date    DATE );
4
5  INSERT INTO contacts VALUES (
6    person_type (10, 'Fouad', 'TOUFIK', 'toufik.fouad@gmail.com', '06  55 66 77 88'),
7   '15 Jan 2020' );
```

Figure 24: Exemple de la table relationnelle contacts avec une colonne objet

La table *contacts* est une table relationnelle avec un type d'objet comme type de données de l'une de ses colonnes. Les objets qui occupent les colonnes des tables relationnelles sont appelés (objets de colonne).

2.6. Héritage de Types

L'héritage de types permet de créer des hiérarchies de types en définissant des niveaux successifs de sous-types de plus en plus spécialisés qui dérivent d'un type d'objet ancêtre commun, qui est appelé un *supertype* des types dérivés. Les sous-types dérivés héritent des fonctionnalités du type d'objet parent et peuvent étendre la définition du type parent. Les types spécialisés peuvent ajouter de nouveaux attributs ou méthodes, ou redéfinir les méthodes héritées du parent. La hiérarchie de types résultante fournit un niveau d'abstraction plus élevé pour gérer la complexité d'un modèle d'application.

La figure 25, montre la création du type *etudiant_type* qui représente un sous type de *person_type*

```
1  CREATE TYPE etudiant_type UNDER person_type (
2    birth_date DATE);
3
```

Figure 25: Exemple d'héritage entre les types d'objet

En tant que sous-type de *person_type*, *etudiant_type* hérite de tous les attributs déclarés dans *person_type* ou hérités par *person_type* et toutes les méthodes héritées par *person_type* ou déclarées dans *person_type*.

2.7. Les tables Objet

Une table objet est un type spécial de table dans lequel chaque ligne représente un objet. Dans l'exemple suivant, l'instruction crée une table d'objets pour les objets *person_type*.

```
1  CREATE TABLE person_obj_tab OF person_type;
2
```

Figure 26: Exemple de création d'une table objet

On peut interpréter cette table en deux manières ou représentation :

- Comme une table à colonne unique dans laquelle chaque ligne est un objet *person_type*, ce qui permet d'effectuer des opérations orientées objet.
- Comme une table à plusieurs colonnes dans laquelle chaque attribut du type d'objet *person_type*; tels que nom, email et téléphone, chaque attribut occupe une colonne, ce qui permet d'effectuer des opérations relationnelles.

La figure suivante montre un exemple d'opérations sur la table *person_obj_tab*

```sql
INSERT INTO person_obj_tab VALUES (
        person_type(12, 'Fouad', 'TOUFIK', 'toufik.fouad@gmail.com', '06  55 66 77 88') );

SELECT VALUE(p) FROM person_obj_tab p
         WHERE p.last_name = 'TOUFIK';

-- PL/SQL Block pour afficher les informations d'une personne
DECLARE
  person person_type;
BEGIN
  SELECT VALUE(p) INTO person FROM person_obj_tab p WHERE p.idno = 12;
  person.getInfo();
END;
```

Figure 27: Insertion et affichage des informations de la table objet person_obj_type

La première instruction *SQL* de l'exemple précédent insère un objet *person_type* dans *person_obj_tab*, traitant *person_obj_tab* comme une table à plusieurs colonnes. La deuxième instruction *SQL* sélectionne dans *person_obj_tab* une table à colonne unique, en utilisant la fonction *VALUE* pour renvoyer des lignes en tant qu'occurrences d'objet.

Le bloc *PL / SQL* sélectionne une personne spécifique et exécute une fonction membre de *person_type getInfo()* pour afficher les informations de la personne sélectionnée.

Par défaut, chaque objet ligne d'une table d'objet possède un identificateur d'objet logique (*OID*) associé qui l'identifie de manière unique dans une table d'objets. Dans un environnement distribué et répliqué, l'identifiant unique généré par le système permet à Oracle d'identifier les objets sans ambiguïté.

2.8. Les Références d'objet

Une *REF* est un pointeur logique vers une ligne d'objet qui est construit à partir de l'identificateur d'objet (*OID*) de l'objet référencé et est un type de données intégré *Oracle*. Les *REFs* et les collections des *REFs* modélise les associations entre les objets, en particulier les associations plusieurs-à-un, ce qui réduit l'utilisation des clés étrangères. Les *REF* fournissent un mécanisme facile pour naviguer entre les objets. On peut utiliser une *REF* pour examiner ou mettre à jour l'objet auquel elle se réfère et aussi l'utiliser pour obtenir l'objet auquel il se réfère. *REF* donne la possibilité de modifier la valeur et pointer vers un autre objet de la même hiérarchie de type d'objet ou lui affecter une valeur nulle.

La figure suivante montre un exemple de liaison entre un *employé* et son *manager* par l'utilisation du mécanisme des références :

```
1  CREATE TYPE emp_type AS OBJECT (
2    name     VARCHAR2(30),
3    manager  REF emp_type );
4  /
5  CREATE TABLE emp_obj_tab OF emp_type;
6
7  INSERT INTO emp_obj_tab VALUES (emp_type ('Bahaj Mohamed', NULL));
8
9  INSERT INTO emp_obj_tab
10   SELECT emp_type ('TOUFIK Fouad', REF(e))
11     FROM emp_obj_tab e
12     WHERE e.name = 'Bahaj Mohamed';
13
```

Figure 28: Exemple de liaison entre deux objets en utilisant les références

2.9. Les Collections Oracle

Pour la modélisation des attributs à valeurs multiples et de relations un à plusieurs ou plusieurs à plusieurs, Oracle prend en charge deux types de données de collection, les *varrays* et les tables imbriquées (*nested tables*). Les types de collection peuvent être utilisés partout où d'autres types de données peuvent être utilisés. On peut avoir des attributs d'objet d'un type de collection en plus des colonnes d'un type de collection. Par exemple, on peut attribuer à un type d'objet *person_type* un attribut de table imbriqué pour contenir une collection d'éléments de *address_type*.

2.5.1. Les Varrays

varray est un ensemble ordonné d'éléments de données. Tous les éléments d'un *varray* donné sont du même type de données ou d'un sous-type de celui déclaré. Chaque élément a un index, qui est un nombre correspondant à la position de l'élément dans le tableau. Le numéro d'index est utilisé pour accéder à un élément spécifique.

Lors d'une définition d'une collection de type *varray*, on spécifie le nombre maximal d'éléments qu'elle peut contenir, avec la possibilité de modifier ce nombre ultérieurement. Le nombre d'éléments dans un tableau est la taille du tableau. Oracle permet aux tableaux d'être de taille variable.

L'exemple suivant montre la création d'une collection *address_list* de type *varray* avec 10 éléments en maximum de type primitif *varchar2(50)*

```
1   CREATE TYPE address_list AS VARRAY(10) OF VARCHAR2(50);
2   /
```

Figure 29: Exemple de création d'une liste varray des type primitif

L'exemple précédent crée une *varray* de type primitif *varchar2(50)*. Oracle nous donne la possibilité de créer une collection de type objet.

```
1   CREATE TYPE address_type AS OBJECT (
2       country   VARCHAR2(15),
3       street      VARCHAR2(20));
4   /
5   CREATE TYPE address_varray_type AS VARRAY(5) OF address_type;
6   /
7   CREATE TABLE person_table (
8     name VARCHAR2(30),
9     address_list address_varray_type);
10
11  INSERT INTO person_table VALUES (
12      'TOUFIK FOUAD',
13      address_varray_type( address_type ('Morocco', 'Med 5'),
14                           address_type ('Morocco', 'Mly Youssef'));
15
```

Figure 30: Exemple de création d'une liste varray des objets

Dans la figure 30 nous avons créé un *varray* qui représente un tableau de type objet. Le type *varray address_varray_type* est utilisé comme type de données pour une colonne de la table *person_table*. L'instruction *INSERT* montre comment insérer des valeurs dans *address_varray_type* dans la table.

La création d'un type de *varray*, comme un type d'objet SQL, n'alloue aucun espace. Il définit un type de données qui peut être utilisé comme :

- Type de données d'une colonne d'une table relationnelle.
- Un attribut de type objet.
- Type d'une variable PL/SQL, un paramètre ou une valeur de retour d'une fonction.

Un *varray* est normalement stocké en ligne, c'est-à-dire dans le même espace de table que les autres données de sa ligne. S'il est suffisamment volumineux, Oracle le stocke en tant que *BLOB.*

On peut créer un type *varray* de *XML Type* ou d'un type *LOB* à des fins procédurales, comme dans *PL/SQL* ou dans les requêtes de vue. Cependant, le stockage de base de données d'un *varray* de ces types n'est pas pris en charge.

2.5.2. Les tables imbriquées (Nested tables)

Une table imbriquée est un ensemble non ordonné d'éléments de données de même type. Aucune limite n'est spécifiée dans la définition de la table et l'ordre des éléments n'est pas conservé. On sélectionne, insère, supprime et met à jour les données dans une table imbriquée comme on le fait avec des tables ordinaires à l'aide de l'expression *TABLE*.

Les éléments d'une table imbriquée sont stockés dans une table de stockage distincte qui contient une colonne qui identifie la ligne de table parent ou l'objet auquel appartient chaque élément. Une table imbriquée a une seule colonne et le type de cette colonne est un type primitif ou un type d'objet. Si la colonne d'une table imbriquée est un type d'objet, la table peut également être affichée comme une table à plusieurs colonnes, avec une colonne pour chaque attribut du type d'objet.

Pour créer une table imbriquée on utilise l'instruction (*CREATE TYPE ... IS TABLE OF)*. La définition d'une table imbriquée n'alloue aucun espace. Il définit un type de données qui peut être utilisé comme :

- Type de données d'une colonne d'une table relationnelle.
- Un attribut de type objet.
- Type d'une variable *PL/SQL*, un paramètre ou une valeur de retour d'une fonction.

Lorsqu'une colonne d'une table relationnelle est de type table imbriquée, Oracle stocke les données de la table imbriquée pour toutes les lignes de la table relationnelle dans la même table de stockage. De même, avec une table d'objets d'un type qui a un attribut de table imbriquée, Oracle stocke les données de table imbriquée pour toutes les instances d'objet dans une seule table de stockage associée à la table d'objet.

```
CREATE TYPE address_type AS OBJECT (
    country   VARCHAR2(15),
    street      VARCHAR2(20));
/
CREATE TYPE address_nt_type AS TABLE OF address_type;
/
CREATE TYPE manager_type AS OBJECT (
  id NUMBER;
  name VARCHAR2(30),
  address_list address_nt_type);

CREATE TABLE manager_table OF manager_type(
  CONSTRAINT primary_key primary key(id)
)
NESTED TABLE address_list STORE AS address_nt_table;
/

```

Figure 31: Exemple de création d'une liste nested tables des objets

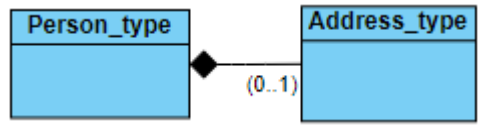
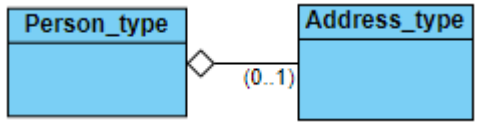
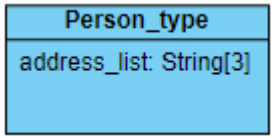
L'exemple précédent crée une collection dynamique de type table imbriquée *NESTED TABLE address_nt_type* de type *address_type.* La table imbriquée crée est définit comme attribut dans le type *manager_type*.

III. Extraction de modèle

La phase d'extraction de modèle transforme chaque *UDT* à un équivalant ensemble de class et des associations dans un diagramme de classes *UML*. Cette phase regroupe une liste des étapes et des règles de transformation.

En effectuant des requêtes d'interrogation auprès du dictionnaire de la base de données, on récupère les informations relatives aux différent *UDT* présents dans le schéma étudié.

Règle	Schéma Source Oracle DB	Résultat UML DC
Integer, Float et *Date* sont transformés respectivement aux types UML *Integer*, *Real* et *Date*.	*Integer, Float, Date*	*Integer, Real, Date*
Number(Pr, Sc) est transformé en *Integer* lorsque la précision est *0* et en *Real* dans les autres cas.	*Number*(*precision,Scale*)	(*Integer, Real*)
Les types de données caractères *char* et *varchar* sont transformé en *String* Dans le cas où le type est *char(n)* la longueur de l'attribut doit être égale à *n*. Si le type de données est *varchar*, la longueur de l'attribut ne peut pas dépasser *n*. Pour vérifier ces conditions, on utilise les instructions *OCL*.	*Char(n), Varchar2*(*n)*	*String*
Les *UDTs* dans la base de données objet relationnelle Oracle peuvent avoir quatre différentes représentations (Un Type d'objet, une référence vers un type d'objet, une collection d'un type d'objet, une collection des références d'un type d'objet)		

La transformation génère une nouvelle relation de composition, une forte association entre les deux types (*address_type, person_type)* avec *(0..1)* comme multiplicité à côté de l'entité *address_type*.	*create type address_type as object (…)* *create type person_type as object(address address_type …)* **(Ex1)**	
En utilisant les références, une autre représentation de l'UDT est possible dans *l'ORDB*. Les références permettent de créer des complexes objets et récupérer des données facilement sans utiliser les jointures entre les tables. Les références permettent également la création d'une relation faible entre deux types. L'exemple **Ex1** présente une forte relation entre *address_type* et *person_type*, en déclarant l'attribut *address* dans *person_type* comme référence de *address_type*, la transformation génère un relation d'agrégation avec les mêmes propriétés mentionné précédemment dans **Ex1**.	*Create type address_type as object(…)* *Create type person_type as object(address REF address_type …)*	
Les attributs définis comme collection *(table imbriquée, varray)* de types primitifs *(entier, float, varchar* …), Sont transformées en tableau de type primitif dans le schéma conceptuel. La longueur minimale est égale à 0 et la taille est	*Cas 1 :* *create type address_type as varray(3) of varchar2;*	

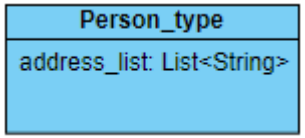
<table>
<tr>
<td rowspan="2">égale à la longueur du varray, et illimité dans le cas d'une table imbriquée.

L'attribut address_list défini dans person_type est une collection de varchar2, le processus de mapping de l'attribut cible génère une collection String à l'intérieur de la classe person_type, limitée dans le cas de varray (3) (cas 1) où 3 est la longueur maximale de la collection et illimité en cas d'une table imbriquées (cas 2).</td>
<td>create type person_type as object(address_list

address_type,…);</td>
<td></td>
</tr>
<tr>
<td>Cas 2:

create type address_type as table of varchar2

create type person_type as object(address_list

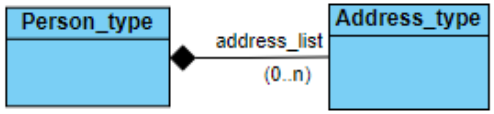
address_type,…)</td>
<td> 
</td>
</tr>
<tr>
<td>Lorsque l'attribut est une collection d'UDT, la transformation donne naissance à une nouvelle composition,

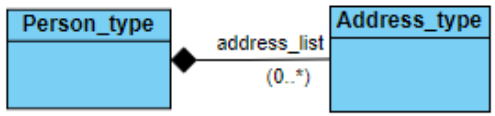
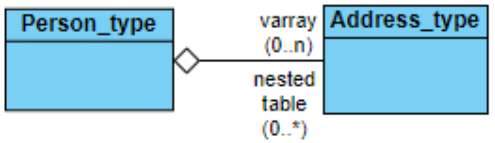
L'attribut address_list défini dans person_type est une collection de type_adresse, la transformation génère une relation de composition entre person_type et address_type avec (0 .. *) comme multiplicité côté adresse_type et liste_adresses comme nom du rôle en cas de table imbriquée, en cas de</td>
<td>Cas 1:

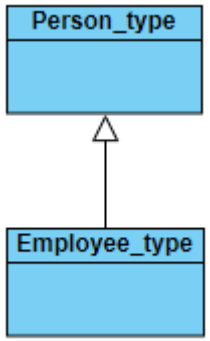
create type address_type as object(city varchar,..)

create type address_type_varray as varray(n) of address_type

create type person_type as object(address_list

address_type_varray,…)</td>
<td> 
</td>
</tr>
</table>

varray nous changeons la multiplicité en *(0..n)* où *n* est la longueur maximale de la collection. **(Ex2)**		
	Cas 2: *create type address_type as object(city varchar,..)* *create type address_type_nested as table of address_type* *create type person_type as object(address_list* *address_type_nested,…)*	
Lorsque l'attribut est une collection des *REF*s d'un UDT, on conserve toutes les propriétés et les résultats de la précédente transformation **(Ex2)** et on change l'association entre les deux types en une relation faible et change l'association de composition en agrégation.	*Create type address_ref as object(addr_r REF* *address_type)* *Create address_varray as varray(n) of address_ref;*	

	Create address_nestes as table of of address_ref; *create type person_type as object(address_list1 address_varray, address_list2 REF address_nested,…)*	
Oracle offre le mécanisme d'héritage qui connecte les sous-types d'une hiérarchie à leurs supertypes. Les sous-types héritent automatiquement les attributs et les méthodes de leur type parent. Tous les attributs où les méthodes mises à jour dans un supertype sont mises à jour dans les sous-types. *Employee_type* est un sous-type de *person_type*, la transformation génère une relation de généralisation entre ces classes dans le schéma conceptuel.	*create type person_type as object(…) not final;* *create type employee_type under person_type*	
Les tables de base de données ne contiennent que des données, les objets ont la possibilité d'effectuer des opérations sur ces données. Les méthodes d'objets sont des fonctions ou des procédures déclarées dans un type de définition d'objet pour implémenter le comportement voulu auprès des objets d'un type donné. Il existe trois types de méthodes qui peuvent être déclarés dans une définition de type (membre, méthodes statiques et méthodes constructeurs).		

Après avoir récupéré les informations sur la méthode (paramètres et résultat renvoyé), le processus de mapping transforme toutes sortes de méthodes de l'UDT et ajoute la même signature à la classe ciblée dans le schéma conceptuel

Table 1: Régles de transformation du modèle

IV. Extraction des contraintes

1. Transformation des contraintes d'intégrité déclaratives

Les contraintes sont les règles appliquées aux données enregistrées dans la table. En cas de violation entre la contrainte et l'action *DML (Data Manipulation Language)* de données, l'action *(insert,update, delete)* est abandonnée par la contrainte. Les contraintes sont utilisées pour garantir la précision et cohérence des données stockées dans une table typée dans une base de données objet relationnelle.

Pour enrichir le schéma conceptuel et le rendre complet et compréhensible, toutes ces contraintes doivent être inclus dans le diagramme de classes, afin de transformer et présenter ces contraintes, nous utilisons *OCL*. *OCL* est un langage de spécification textuelle, spécialement conçu pour être utilisé dans le cadre de langages de spécification schématiques tels qu'UML.

OCL a toujours été utilisé pour ajouter des règles et des restrictions au niveau modèle et méta modèle dans *UML*. *OCL* est fortement connecté aux diagrammes *UML*, car il est utilisé comme un texte naturel et compréhensible dans les différents diagrammes. *OCL* utilise les éléments définis dans les diagrammes *UML*, tels que les classes, méthodes et attributs

Le langage est basé sur les types. Chaque expression *OCL* est évaluée en un type soit prédéfini par le langage ou défini par le modèle sur lequel l'expression est construite.

Pour présenter toutes les transformations possibles de contraintes d'intégrité déclarative appliquées à une table objet relationnelle, on se focalise sur l'exemple dans la figure 32.

```
1   CREATE TYPE person_type AS OBJECT(person_id int, passport_id int,
2                                     name VARCHAR(20), age int, city VARCHAR(20));
3   CREATE TABLE person_table OF person_type(person_id primary key,
4                                     passport_id unique, name not null,
5                                     city default 'casa', check (age<100)) ;
6
```

Figure 32: Exemple des contraintes déclaratives appliquées sur une table objet relationnelle

La contrainte *DEFAULT* est utilisée pour insérer une valeur par défaut dans une colonne. La valeur par défaut sera ajoutée à tous les nouveaux enregistrements, si aucune

autre valeur n'est spécifiée. Après avoir transformé *l'UDT* ciblé en classe, nous avons maintenu la possibilité d'utiliser les concepts d'objet comme les constructeurs. Le constructeur est une fonction membre non statique d'une classe qui est utilisé pour initialiser des objets de son type de classe. Les constructeurs sont invoqués lors de l'initialisation et sont sélectionnés selon les règles d'initialisation et peuvent attribuer des valeurs à tout champs ou propriété d'un objet au moment de la création.

L'exemple ci-dessous présente un constructeur d'initialisation qui peut affecter des valeurs par défaut à plusieurs attributs.

```
1   public Person_type (String city) {
2   this.city=city;
3   }
4   Person_type person=new person('casa');
5
```

Figure 33 : Exemple de constructeur d'initialisation

La contrainte *NOT NULL* s'applique à un champ pour qu'il contienne toujours une valeur. Cela signifie que l'insertion ou la mise à jour est non autorisée sans ajouter une valeur à ce champ, pour transformer et présenter cette contrainte comme une condition *OCL*, on crée un invariant qui utilise la méthode *ocllsUndefined()* qui retourne *true* si la valeur invalide ou nulle. Le *context* de l'invariant est la class transformée d'un *UDT* dans la première phase de l'extraction du modèle.

L'exemple suivant présente les instructions *OCL* qui empêche l'attribut *name* de la class *person_type* de contenir une valeur nulle

```
1   context person_type inv:
2           Not self.name.ocllsUndefined()
```

Figure 34: Transformation de la contrainte déclarative NULL

Les contraintes *UNIQUE* et *PRIMARY Key* assurent et fournissent la garantie de l'unicité d'une colonne ou d'un ensemble de colonne. La contrainte *Unique* est automatiquement définie au sein de la contrainte *Primary Key*. Une colonne avec la contrainte *Primary Key* ne peut pas contenir une valeur *Null*.

L'exemple suivant présente la transformation de la contrainte *Unique* appliqué sur une colonne à des instructions OCL :

```
context Person_type inv:
        Person_type.allInstance()
        ->forAll(person1,person2 | person1 <> person2
            Implies person1.passport_id <> person2.passport_id)
```

Figure 35: Transformation de la contrainte décalarative UNIQUE

allInstances() est une fonctionnalité associée à tout type qui renvoie l'ensemble de toutes les instances du type donné.

La méthode *forAll()* dans *OCL* permet de spécifier une expression booléenne, qui doit être valable pour tous les objets d'une collection.

La contrainte *PRIMARY KEY* a une contrainte *UNIQUE*, alors on prend le même résultat de la transformation de la contrainte UNIQUE et on ajoute la méthode *ocllsUndefined()* pour chaque objet retournée et récupérée de la collection à l'aide de la méthode *allInstances()* pour empêcher le stockage des valeurs nulles

```
context Person_type inv:
        Person_type.allInstance()
        ->forAll(person1,person2 | person1 <> person2
            Implies person1.person_id <> person2.person_id
            And Not person1.person_id.ocllsUndefined())
```

Figure 36: Transformation de la contrainte déclarative Primary Key

La contrainte *CHECK* permet de spécifier une condition à chaque ligne dans une table, chaque contrainte *CHECK* génère un invariant *OCL* avec la même expression présentée dans le corps de la contrainte, l'exemple ci-dessous montre la transformation de la contrainte *CHECK*

```
context class_name inv:
    self.attribute_name.<condition SQL-TO-OCL Mappnig>

context person_type inv:
    self.age<100
```

Figure 37: transformation de la contrainte déclarative CHECK

Dans les bases de données relationnelles classiques, on utilise les clés étrangères pour gérer les relations entre les tables et pour extraire les données, on utilise plusieurs jointures dans les requêtes pour obtenir le résultat adéquat. Dans les bases de données objet relationnelle, de nouveaux concepts ont été mis en œuvre pour gérer les relations entre les tables telles que les objets imbriqués et les références aux lignes objets à l'aide de *REF* et *OID*.

(OID) Object Identifier identifie de manière unique les lignes objets dans les tables objets relationnelles. Une *REF* est un pointeur logique ou une référence à une ligne objet que nous pouvons construire à partir d'un identifiant d'objet.

On peut utiliser *REF* pour obtenir, examiner ou mettre à jour l'objet et également changer une *REF* afin qu'elle pointe vers un objet différent de la même hiérarchie de type d'objet ou lui affecter une valeur nulle. Les *REFs* sont des types intégrés *(built-in)* dans la base de données Oracle, *REF* et les collections des *REFs* modélise les associations entre les différents objets, en particulier les relations un à plusieurs qui réduisent le besoin des clés étrangères, les *REFs* fournissent un mécanisme facile pour la navigation entre les objets, l'exemple suivant présente une relation *command_type* et *client_type* en utilisant *REF* :

```
1   create type client_type as object(client_id int,name varchar(20),…);
2   create type command_type as object(command_id int,command_date,
3                                      client_ref REF client_type);
```

Figure 38: Exemple de relation entre deux types en utilisant REF

On peut mettre une contrainte sur une colonne, élément d'une collection ou un attribut d'un objet pour référencer un objet dans une table spécifiée en utilisant la contrainte sous ensemble *SCOPE IS* lors de la déclaration de *REF*. Les *SCOPED REFs* nécessitent moins d'espace de stockage et permettent un accès plus efficace que les simples *REFs*, une *REF* peut être liée à une table d'objets ou à n'importe quel sous-type du type déclaré. Si une REF est liée à une table d'objets d'un sous-type, la colonne contenant la REF est effectivement sous une contrainte pour ne contenir que les références aux instances du sous-type et de ses sous-types s'ils existaient dans la table (mécanisme d'héritage dans *ORDB*).

L'exemple suivant montre la transformation des *SCOPED REFs* à des instructions *OCL*

```
create table client_table of client_type;
create table command_table(client_ref REF
                client_type SCOPE IS client_table);

context Command_type inv:
    self.allInstances()->forAll(Command_type cmd |
        Client_type.allInstances()->collect(clt.getOID)->exist(cmd.getClientRef))
```

Figure 39: Exemple de transformation des Scoped Ref

Comme indiqué dans le résultat ci-dessus, le contexte principal est la classe *command_type*. On prend chaque élément de la collection *allInstances()* en utilisant la méthode *forAll()*, pour vérifier chaque instance de *command_type*. L'étape suivante consiste à vérifier l'existence de client *REF* dans l'ensemble de données client_type.

2. Transformation de SQL à OCL

Cette section présente les étapes de transformation des instructions *SQL SELECT* en expressions *OCL*. En particulier, on décrit le mappage des projections *SQL*, des sélections, des méthodes, des *clauses order by, group by et having*. Cette transformation est nécessaire pour extraire les contraintes implémentées dans le cadre des déclencheurs. Dans ce qui suit, on présente des exemples de transformation d'instructions *SQL*.

```
SELECT {DISTINCT | UNIQUE} colName,
methodName()
FROM tableName
WHERE ({coleName, methodName} condition)
ORDER BY colName
```

Figure 40: Exemple générique de la requête Select

Les expressions *OCL* équivalant :

```
context className inv
    self.allInstances() ->
        select({colName | mehodName} condition)->
        collect(colName,methodName) ->
        asSet()->asSequence()
        OR
        asOrderedSet()
```

Figure 41: Resultat de transformation de la requête générique Select

La transformation commence par la clause *FROM*, *tableName* est la classe ciblée qui s'est transformée en phase d'extraction de modèle en *className*, le contexte de l'expression *OCL* est *className* qui spécifie l'entité du modèle UML pour laquelle l'expression est définie, la méthode *allInstances()* renvoie l'ensemble de toutes les instances de la classe donnée.

La clause *WHERE* est transformée en un *iterator* de sélection *OCL* qui renvoie une collection de tous les éléments qui valident la condition, les *colNames* de la clause *WHERE* sont mappés aux noms d'attribut et d'association correspondants. Les fonctions *SQL* sont traduites à leurs homologues *OCL* (si des opérations *OCL* existantes sinon nouvelles méthodes doivent être définies au préalable).

La clause *SELECT* est transformée en *collect iterator OCL* qui crée une collection d'objets selon la structure définie dans la définition de *tuple*, chaque attribut présent dans l'expression *OCL* correspond à une colonne de l'instruction *SELECT*, pour récupérer des données différentes et uniques d'une table donnée on utilise *DISTINCT*, cette clause est mappée en ajoutant la méthode *asSet()* après avoir transformé la clause *SELECT*.

ORDER BY est utilisé pour trier l'ensemble de résultat par une ou plusieurs colonnes. Pour trier les données en OCL, on utilise *asSequence()* qui retourne une collection ordonnée, cette collection peut contenir des éléments en double, si les clauses *DISTINCT* et *ORDER BY* sont toutes les deux présentes, on peut utiliser la méthode *asOrederedSet()* pour obtenir une collection ordonnée avec des éléments uniques.

L'héritage est le mécanisme qui relie les sous-types d'une hiérarchie à leurs super types, les sous-types héritent automatiquement les attributs et les méthodes de leur type

parent, un sous-type peut être dérivé d'un super type soit directement soit indirectement via des niveaux intermédiaires. Un super type peut avoir plusieurs sous-types frères de même niveau, mais un sous-type ne peut avoir qu'un seul super type parent (héritage unique et non support de l'héritage multiple).

Avec les types d'objet dans une hiérarchie de types, on peut modéliser une entité telle que *person_type* et définir différents sous-types spécialisés comme *professor_type* et *student_type*. L'exemple ci-dessous montre les instructions de création des types mentionnés précédemment:

```
1   create type person_type as object(person_id,
2                                person_name varchar(20),…) not final;
3   Create type professor_type under person_type(…);
4   Create type student_type under person_type(…);
5   Create table person_table of person_type;
```

Figure 42: Exemple d'héritage sous Oracle 12c

person_type est déclaré *NOT FINAL* pour créer des sous-types. La table objet relationnelle *person_table* est créée pour contenir des données des trois types.

L'exemple ci-dessous montre l'utilisation de la fonction *value()* qui aide à sélectionner les lignes *professor_type* dans *person_table*, *value()* prend comme argument une variable de corrélation (alias de table) associée à une ligne d'une table objet et renvoie les instances d'objet stockées dans la table objet. Le type des instances d'objet (avec les sous-types) est du même type que la table d'objets.

```
1   select value(p) from person_table p
2       Where value(p) IS OF (professor_type);
```

Figure 43: Exemple de selection des objets d'un sous-type en utilisant valueOf

La transformation de la requête d'héritage génère l'invariant *OCL* comme décrit ci-dessous :

```
1  context person_type inv:
2      self.allInstances()->select(p : Person_type | p.oclIsTypeOf(professor_type))
3
```

Figure 44: Resultat de transformation de la selection des sous-types

La fonction *OCL ocllsTypeOf()* vérifie le type de chaque instance et renvoie l'objet souhaité en fonction du paramètre passé à la fonction. Dans cet exemple, on sélectionne toutes les instances de type *professor_type*.

3. Transformation des déclencheurs

Comme les contraintes d'intégrité déclarative, les déclencheurs peuvent mettre des contraintes de saisie de données et appliquer tout type de règle d'intégrité, un déclencheur s'applique uniquement aux nouvelles données. Par exemple, un déclencheur peut empêcher une instruction *DML (Data Manipulation Language)* d'insérer une valeur *NULL* dans une colonne de base de données, mais la colonne peut contenir des valeurs *NULL* qui ont été insérées dans la table avant la définition du déclencheur ou pendant la désactivation du déclencheur.

Les contraintes sont plus faciles à écrire et moins sensibles aux erreurs que les déclencheurs qui appliquent les mêmes règles. Cependant, les déclencheurs peuvent appliquer des règles métiers complexes ou d'intégrité référentielle que nous ne pouvons pas définir avec des simples contraintes.

Les Triggers sont similaires aux procédures stockées. Un trigger stocké dans la base de données peut inclure des instructions *SQL* et *PL/SQL* à exécuter en tant qu'unité et peut être invoqué à plusieurs reprises, contrairement à une procédure stockée, nous pouvons activer et désactiver un trigger, mais nous ne pouvons pas l'invoquer explicitement. Il se compose en déclenchant un événement ou une instruction, en déclenchant une restriction et enfin en déclenchant une action. L'événement déclencheur est une instruction *SQL*, un événement de base de données ou un événement utilisateur qui provoque le déclenchement du trigger, la restriction de déclenchement est une expression booléenne qui doit être vraie pour que le trigger se déclenche, l'action de déclenchement est un bloc *PL/SQL* qui contient du code *SQL* et procédural à exécuter lorsque l'événement déclencheur se produit et que la condition de restriction est vraie.

Pour identifier les triggers appliquant une règle métier complexe définie par l'utilisateur, on utilise la proposition définie dans [28]: tous les triggers contenant dans leur section des instructions PL/SQL levant une exception sont classés comme des triggers imposant des contraintes.

La transformation de tels déclencheurs commence par le contexte de l'invariant *OCL*, qui est la classe *UML* correspondant à la table où le déclencheur est défini (la classe cible est déjà récupérée dans la première phase de l'extraction de modèle), le corps de l'invariant *OCL* est composé de la condition de restriction du trigger, si elle est définie, et des instructions *OCL* générées à partir du bloc *PL/SQL* défini dans le corps du trigger. La figure 45 présente la transformation générique et basique d'un trigger simple vers *OCL*.

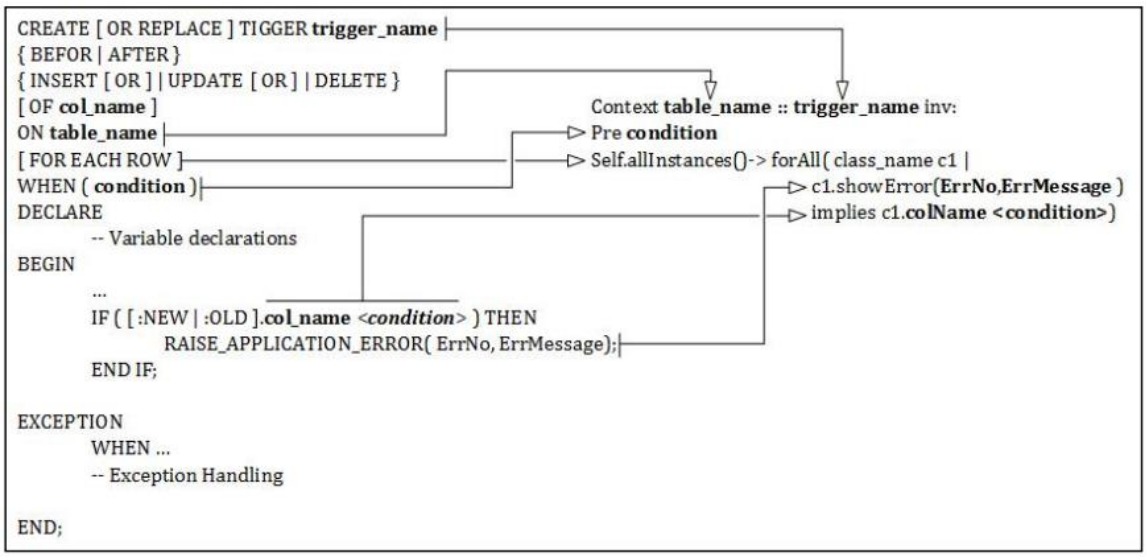


Figure 45: Transformation d'un trigger génériqe sous Oracle 12c

Le trigger est déclenché lorsque la condition de la clause *WHEN* est vraie. Pour transformer un trigger conditionnel et exécuter les instructions *OCL* après avoir vérifié la condition, on utilise une pré-condition. Dans *OCL*, le pré et post-conditions sont des contraintes qui définissent un contrat qu'une implémentation de l'opération doit remplir, une pré-condition doit être remplie lorsqu'une opération est appelée, une post-condition doit être vraie lorsque l'opération s'exécute. Comme mentionné précédemment, les déclencheurs sont similaires aux procédures stockées dans certaines caractéristiques, pour cela, les déclencheurs jouent le rôle des méthodes dans un monde orienté objet. Le contexte de l'invariant *OCL* est l'opération *(trigger_name)* de la classe déjà générée à partir de la table *(table_name)* sur laquelle le trigger est créé, la clause *FOR EACH ROW* est transformée en opération *OCL allInstance()*, la condition qui gère l'exception est transformée en fonction *forAll()*, cette fonction affiche le message d'erreur pour chaque instance si l'expression booléenne est vraie.

Dans la figure 46, on présente un exemple de transformation de la contrainte *transaction_check*, le trigger est exécuté lorsque le nouveau montant de la valeur insérée est supérieur ou égal à *100*. Ce trigger génère une exception lorsque le montant retiré de la nouvelle transaction est supérieur au solde du compte, le solde du compte est stocké dans la table des comptes, cette variable est récupérée à l'aide de la variable *account_balance* en utilisant la clause *SELECT INTO*, la transformation de la variable déclarée dans le trigger *transaction_check*, donne naissance à un objet de type *Account*, défini par l'expression *let* et l'objet *Account* en question est récupéré en recherchant le *account_id* à l'aide la fonction *select()*.

La méthode *select()* renvoie une collection avec tous les éléments de classe qui valident la condition *OCL*. Dans *PL/SQL*, l'instruction *SELECT INTO* ne peut renvoyer qu'une seule ligne, par contre la fonction *select()* renvoie une collection d'objets, on utilise la méthode *first()* pour obtenir le premier élément de la liste, le solde du compte est comparé selon une expression booléenne qui mappe la négation de la condition PL/SQL. Enfin, on utilise la post-condition *OCL* (*balancePost*) pour vérifier la nouvelle valeur du solde du compte, la nouvelle valeur doit être égale à la valeur précédente du solde moins le montant, on utilise le modificateur *@pre* pour obtenir la valeur du solde du compte dans la pré-condition *OCL*.

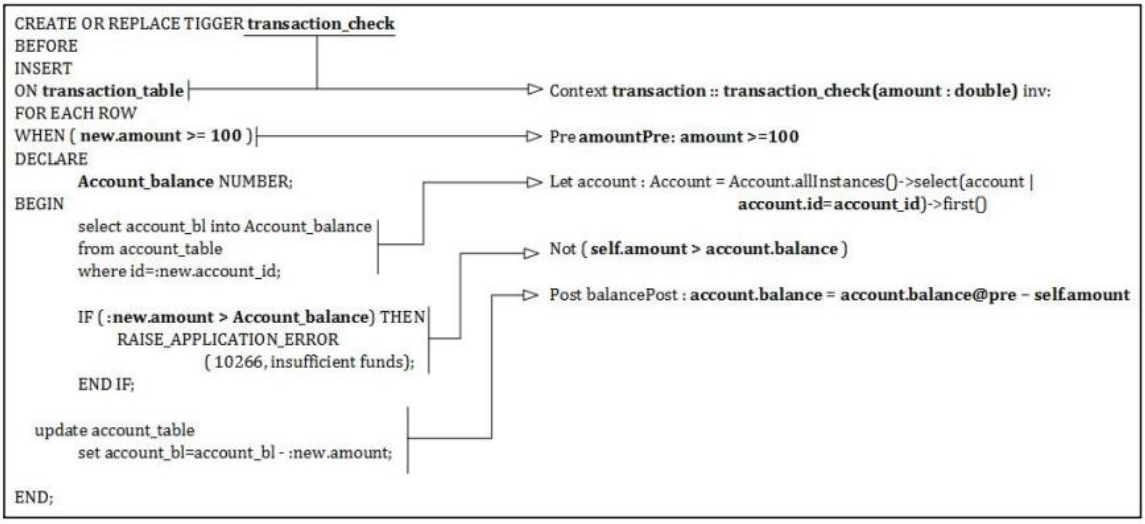


Figure 46: Exemple de transformation du trigger transaction_check

V. Implémentation & Validation

Pour démontrer la validité de notre approche, un outil a été développé (Figure 48) pour présenter la méthode de rétro conception proposée. Pour développer notre prototype, nous avons utilisé java comme langage de programmation et pour créer un diagramme de classes *UML*, nous avons utilisé *graphviz*. *Graphviz* est un logiciel de visualisation des graphes open source lancée par *AT&T Labs Research* pour dessiner des graphes spécifiés dans *DOT* (*DOT* est un langage de description de graphiques en texte brut). Cet outil prend un ensemble de paramètres en entrée pour établir la connexion avec l'instance Oracle, les paramètres d'entrée sont: l'adresse IP du serveur hébergeant la base de données Oracle, le port, le schéma, le nom d'utilisateur et le mot de passe. Après avoir établi la connexion, un ensemble de requêtes *SQL* sont exécutées sur le dictionnaire de données pour obtenir des informations sur les types, les attributs, les méthodes, les associations et d'autres composants.

Le schéma conceptuel obtenu est exprimé sous forme de diagramme de classes *UML* comme illustré à la figure 49. L'exemple ci-dessous présente un ensemble d'instructions SQL de création de types avec des attributs et des méthodes. Cette étude de cas présente un scénario simple pour illustrer le processus de transformation d'une base de données relationnelle objet *ORDB* en diagramme de classes *UML*.

```
1   CREATE TYPE address AS OBJECT(int address_id, city varchar(20), country varchar(20));
2   CREATE TYPE job AS OBJECT (int job_id, title varchar(20), double minSal, double maxSal);
3   CREATE TYPE job_ref AS OBJECT(job_r REF JOB);
4   CREATE TYPE job_list AS VARRAY(3) of job_ref;
5   CREATE TYPE transaction AS OBJECT(transaction_id int, transaction_date date,
6                               transaction_amount double, MEMBER FUNCTION get_last_trans_id
7                               RETURN VARCHAR2)
8   CREATE TYPE transaction_ref AS OBJECT(transaction_r REF transaction);
9   CREATE TYPE transaction_list AS TABLE OF transaction_ref;
10  CREATE TYPE account AS OBJECT(account_id int, account_balance double, transaction_lst transaction_list);
11  CREATE TYPE account_list AS VARRAY(5) of account;
12  CREATE TYPE person AS OBJECT(person_id int, first_name varchar(20), last_name varchar(20),
13                               birth_date date,addr address,
14                               MEMBER FUNCTION get_age RETURN INT,
15                               MEMBER PROCEDURE
16                               show_information )
17                               NOT INSTANTIABLE NOT FINAL;
18  CREATE TYPE client UNDER person(category varchar(20), inscription_date date,
19                               Account_lst account_list,
20                               MEMBER PROCEDURE get_client_info);
21  CREATE TYPE employee UNDER person(salary double, hire_date date, job_lst job_list,
22                               MEMBER FUNCTION
23                               calcul_salary(worked_houres int,
24                               price_per_hour double)
25                               RETURN double);
26
```

Figure 47: Exemple de requête de création Etude de cas

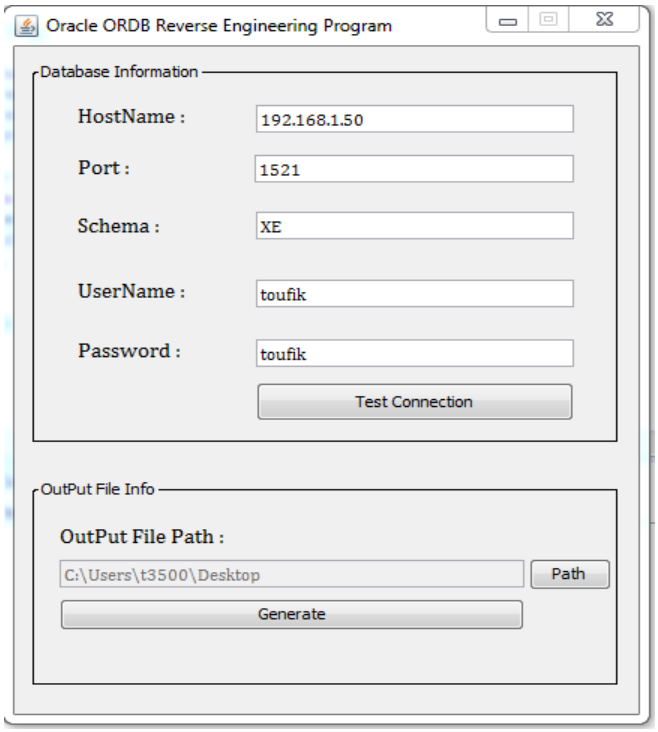


Figure 48: Programme de reverse engineering d'une base de données objet relationnel Oracle

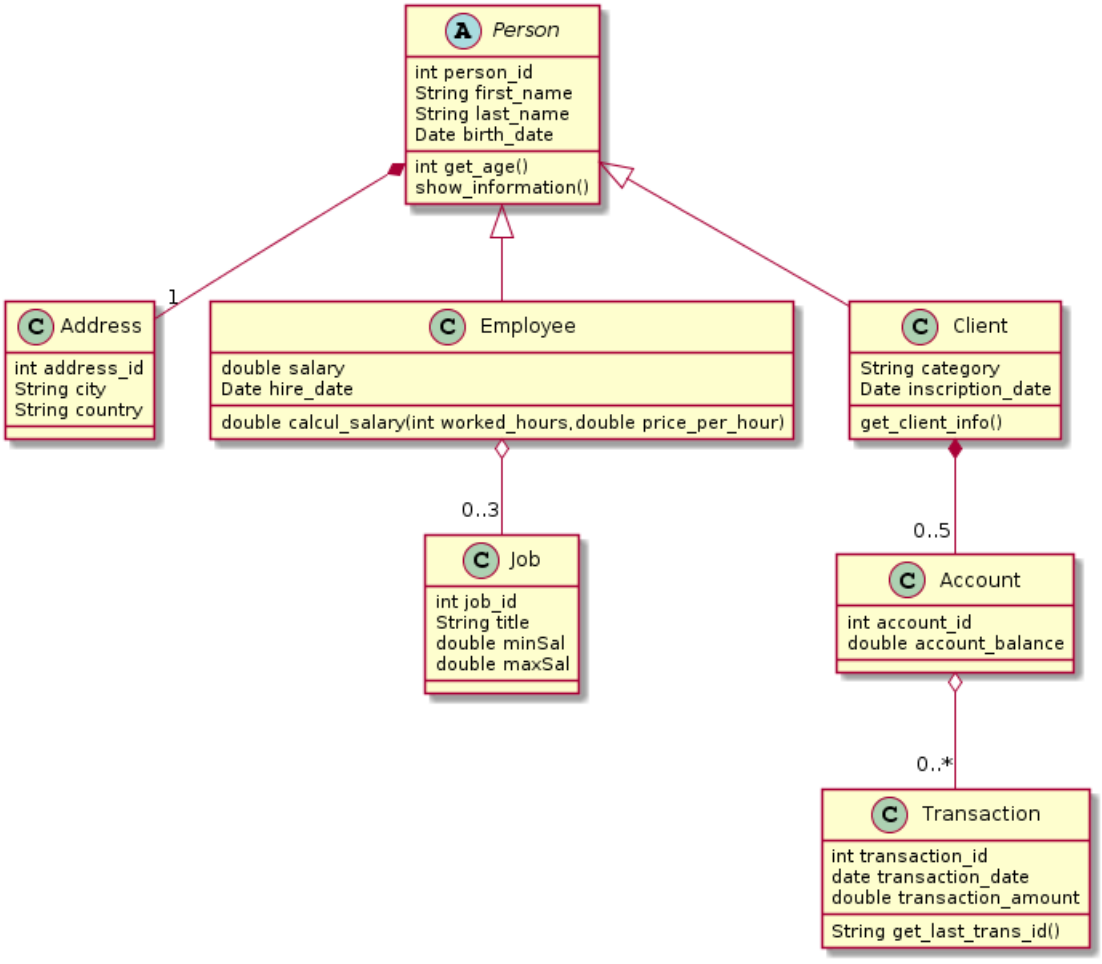


Figure 49: Résultat de la transformation exprimé en diagramme de classes UML

VI. Conclusion et pistes d'évolution

A travers cet axe, nous avons contribué au renforcement des approches de rétro conception ou reverse engineering des bases de données, plus précisément, nous avons présenté une nouvelle méthode de rétro conception pour obtenir un schéma conceptuel enrichi avec un ensemble de contraintes *OCL*, en exécutant un ensemble de requêtes sur le dictionnaire de base de données. Ce schéma conceptuel facilite la compréhension des contraintes d'intégrité et peut aider à la migration, la maintenance et l'évolution de la base de données pour les systèmes utilisant les *ORDBs*. Notre méthode est basée sur deux phases principales, l'extraction du modèle et l'extraction des contraintes; chaque phase contient un ensemble de règles et d'étapes de transformation. Et pour profiter davantage de cette solution, nous envisageons d'étendre notre méthode de rétro-conception pour traiter les méthodes d'objet (les fonctions et les procédures membres) et ajouter la transformation des déclencheurs à notre outil prototype.

Chapitre 4: XML & XQuery Vers OCL

I. Introduction

UML est devenu le langage standard pour la conception et la représentation d'un système orienté objet, il fournit une notation graphique et des éléments pour construire un modèle d'application qui décrit les différents composants du système, leur structure et leur comportement. De plus, *UML* contient différents diagrammes pour modéliser l'aspect statique du système (diagramme de cas d'utilisation, diagrammes de classes,) ainsi que l'aspect dynamique (diagramme de séquence, diagrammes d'état,).

Parallèlement avec l'évolution de l'utilisation des applications Web, *XML* joue un rôle important dans l'échange de données structurées et le transport d'informations sur le Web. *XML* est un format basé sur du texte qui fournit un mécanisme pour décrire les structures de document en utilisant le balisage. *XML* a la définition de type de document (*DTD*) qui définit la structure du document en utilisant un ensemble de règles pour décrire les éléments et autres composants de balisage, mais la *DTD* a certaines limites lorsque nous décrivons des données hautement structurées, dans la *DTD* aucune contrainte n'est imposée sur le type de données de caractère autorisé, donc la saisie de données n'est pas possible, il n'y a pas de support pour les espaces de noms, c'est pourquoi le *World Wide Web Consortium (W3C)* propose d'autres solutions. La meilleure alternative est le *XML* schéma, les XML schémas sont eux-mêmes des documents *XML*, Ils incluent la plupart des types de programmation de base tels que *Integer*, *String* et nombre à virgule flottante, et fournissent également une approche orientée objet pour définir le format d'un document *XML*. Le *XML* schéma présente la partie statique de *XML*, de sorte que le diagramme *UML* qui a les mêmes propriétés, et statique en même temps, est le diagramme de classes, ici nous pouvons mapper et transformer des éléments et des attributs *XML*, en classes et propriétés avec les mêmes caractéristiques (type de données, relation, généralisation,…), au niveau conceptuel, il est nécessaire de générer des modèles conceptuels pour illustrer la structure et les relations des données dans le schéma *XML*. Ici, beaucoup de travail a été fait pour représenter différentes approches et méthodes de mappage entre le XML schéma et UML.

Les documents *XML* étant considérés comme une source d'informations ou une base de données, il est nécessaire d'obtenir des données de cette source. *XQuery* est au

XML ce que *SQL* est à la base de données, *XQuery* est le langage standard du W3C pour récupérer des données à partir d'un document *XML*, la plupart des travaux se concentrent uniquement sur la partie structurelle sans prendre en considération la transformation des restrictions ou ce que nous appelons dans le *XML* schéma les *Facets*, La même chose pour la transformation et le mappage des requêtes *XML*, la plupart des recherches négligent cette partie.

Dans ce chapitre, nous nous concentrons sur les restrictions et les composants *Facets*, pour transformer ces composants, on utilise le langage de contrainte d'objet *OCL*, après on va transformer la généralisation/spécialisation en particulier les contraintes de partition et d'exclusion mutuelle dans *XML* à des expressions *OCL* et enfin, on se focalise sur la partie de la transformation des requêtes *XML*, à l'aide des collections *OCL*.

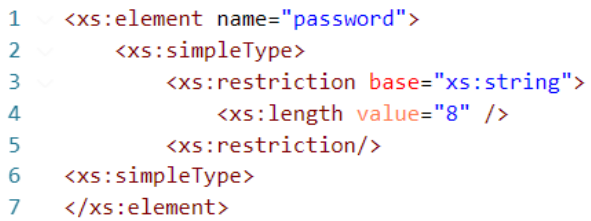
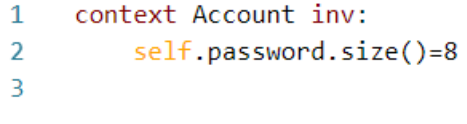
Le reste de ce chapitre est organisé comme suit :

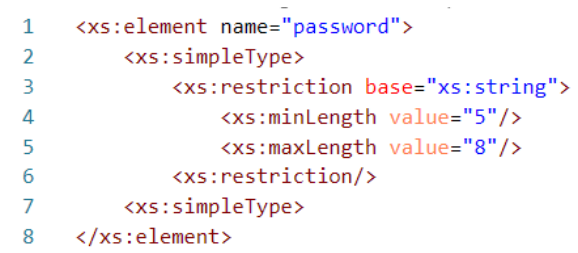
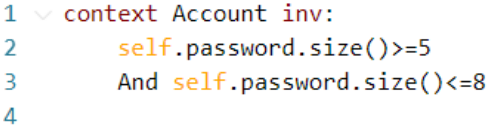
- La section 2 présente une description détaillée de la première étape de notre travail, ici on se focalise sur la partie mapping et transformation des restrictions de *l'XML* schéma vers des expressions *OCL*.
- La section 3 présente comment implémenter la généralisation/spécialisation dans *XML* schéma et comment transformer ces deux importants concepts à des expressions *OCL* par la suite.
- La section 4 présente en détail la transformation du langage de requête pour *XML XQuery* à des expressions *OCL*. Dans cette section on présente l'un des aspects et des points forts *d'OCL*, c'est le principe de navigation entre les différents objets qui nous aide à manipuler plusieurs objets à la fois et créer des invariants *OCL* qui jouent le rôle des requêtes objets.
- La section 5 présente l'implémentation et la validation de notre approche, nous avons développé un outil qui prend en paramètre un fichier *XML* schéma, par la suite le programme il le parse et récupère les restrictions appliquées et enfin il transforme les restrictions obtenues à des expressions *OCL*.

II. Mapping entre constraining facet et OCL

De nombreux travaux ont été réalisés pour représenter le schéma *XML* en *UML*. En détail, les chercheurs se focalise sur la partie conversion des différents éléments et attributs en classes, propriétés et relations (Généralisation, Composition, Agrégation…), mais la représentation des restrictions et des composant *Facets* est très rare, les diagrammes *UML* ne fournissent pas tous les aspects pertinents d'une spécification, c'est pourquoi il est nécessaire de décrire des contraintes supplémentaires sur l'objet dans le modèle. Les contraintes spécifient des conditions invariantes qui doivent tenir pour le système modélisé, les contraintes sont souvent décrites en langage naturel et cela entraîne toujours des ambiguïtés, alors on doit utiliser un langage formel pour exprimer les différentes contraintes, faciles à lire et à écrire, ce langage est *OCL*, *XML* schéma contient 12 restrictions; nous présentons ici quelques contraintes avec les clauses *OCL* correspondantes.

	Bloc XML Schéma	Expressions OCL

Length
 	
On utilise cette restriction pour limiter la longueur d'une valeur dans un élément. L'exemple définit un élément appelé *password* avec une restriction de taille, la valeur doit être exactement huit caractères.
 	
on suppose que l'élément A*ccount* soit le nœud parent
 	
de l'élément de mot de passe, notre contexte ici est la classe compte.
	
	`<xs:element name="password">`
	`  <xs:simpleType>`
	`  <xs:restriction base="xs:string">`
	`  <xs:length value="8" />`
	`  <xs:restriction/>`
	`  <xs:simpleType>`
	`  </xs:element>`
		
		`context Account inv:`
		`  self.password.size()=8`

minLength and maxLength
 	
on utilise cette restriction pour définir la longueur minimale et maximale. L'exemple définit un élément *password* avec une restriction (la valeur doit être au minimum de cinq caractères et au maximum huit caractères).
	
	`<xs:element name="password">`
	`  <xs:simpleType>`
	`  <xs:restriction base="xs:string">`
	`  <xs:minLength value="5"/>`
	`  <xs:maxLength value="8"/>`
	`  <xs:restriction/>`
	`  <xs:simpleType>`
	`  </xs:element>`
		
		`context Account inv:`
		`  self.password.size()>=5`
		`  And self.password.size()<=8`

<table>
<tr>
<td>La classe Account est le contexte, ici on utilise l’opérateur booléen AND pour combiner les deux conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern

On utilise cette restriction pour limiter le contenu d'un élément XML afin de définir une série de chiffres ou de lettres pouvant être utilisés. L'exemple définit un élément name avec une restriction. La valeur acceptable est zéro ou plusieurs occurrences de lettres minuscules de (a à z).

On suppose que l'élément Person est le nœud parent de l'élément name, on suggère d'ajouter une méthode «pattern (…)» au type OCL String, la valeur retournée de cette méthode est un type booléen, égale true si l'attribut respecte le modèle passé en paramètre.</td>
<td>

```
<xs:element name="name">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="([a-z])*" />
        <xs:restriction/>
    <xs:simpleType>
</xs:element>
```

</td>
<td>

```
context Person inv:
    self.name.pattern("[a-z]*") == true

```

</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</table>

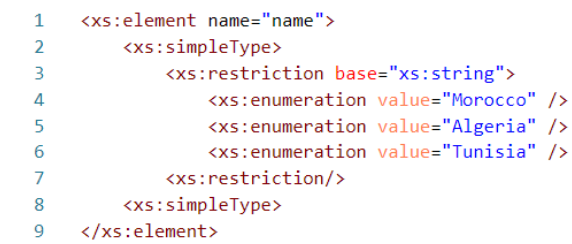
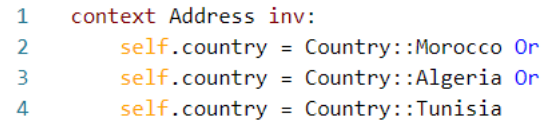
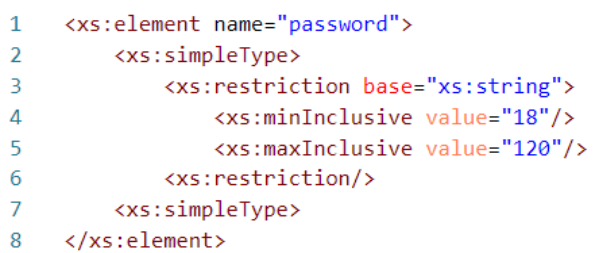
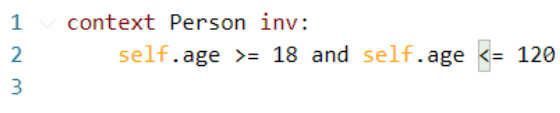
Enumeration On utilise cette restriction pour limiter le contenu d'un élément *XML* à un ensemble de valeurs acceptables. L'exemple définit un élément appelé *country* avec une restriction. Les seules valeurs acceptables sont: *Morocco, Algeria et Tunisia*. On suppose que le nœud parent de l'élément *country* est l'élément *address*	 ``` <xs:element name="name"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="Morocco" /> <xs:enumeration value="Algeria" /> <xs:enumeration value="Tunisia" /> <xs:restriction/> <xs:simpleType> </xs:element> ```	 ``` context Address inv: self.country = Country::Morocco Or self.country = Country::Algeria Or self.country = Country::Tunisia ```
minInclusive and maxInclusive on utilise cette restriction pour contrôler la valeur d'un élément, l'exemple définit un élément appelé *age*. La valeur de *age* ne peut pas être inférieure à *0* ni supérieure à *120*. On suppose que le nœud parent de l'élément *age* est l'élément *person*, alors le contexte de l'invariant OCL est la classe *Person*	 ``` <xs:element name="password"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:minInclusive value="18"/> <xs:maxInclusive value="120"/> <xs:restriction/> <xs:simpleType> </xs:element> ```	 ``` context Person inv: self.age >= 18 and self.age <= 120 ```

Table 2: Régles de transformation des constraining facets

III. La généralisation/spécialisation de l'XML schéma

1. Représentation du généralisation/spécialisation dans XML schéma

Il existe plusieurs mécanismes dans le schéma *XML* qui prennent en charge la généralisation/ spécialisation. La brique principale et fondamentale dans *l'XML* est *element* qui compose avec *attribut* et les éléments imbriqués une infrastructure solide et suffisante pour représenter les différentes structures des données. Alors le point de départ pour n'importe quelle transformation d'un modèle conceptuel à *l'XML* est le *mapping* des *concepts* à des *elements*, les relations généralement sont transformées à des attributs ou à des relations entre des éléments imbriqués. Il y a des complications importantes lorsqu'on considère certains points comme l'identité d'objet, mais le processus global de transformation de structure est assez clair et généralement intuitif.

Cependant, une fois que nous avons une structure de base encodée en *XML*, comment on peut capturer les relations de généralisation/spécialisation et leurs contraintes? On trouve trois composants dans *l'XML* schéma qui prennent en charge les divers aspects de la généralisation/spécialisation: les types dérivés, les groupes de substitution et les éléments abstraits.

2.1. Les types dérivés

Dans *l'XML* schéma, chaque élément à un type bien défini qui décrit un contenu valide de l'élément, les types se divisent en deux grandes catégories: simples et complexes, un type simple peut être dérivé d'un autre en utilisant les restrictions. Par exemple, *String* est un type simple, alors on peut spécifier un type personnalisé de *String*, *domainName*, qui représente l'ensemble des *String* qui correspondent aux domaines génériques d'internet en appliquant une restriction sur le type *String*:

```xml
<xs:simpleType name="domainName">
    <xs:restriction base="xs:string">
        <xs:enumeration value="com" />
        <xs:enumeration value="gov" />
        <xs:enumeration value="net" />
        <xs:enumeration value="org" />
    </xs:restriction>
</xs:simpleType>
```

Figure 50: Création d'un nouveau element basé sur String dans XML schéma

De même, les types complexes peuvent être dérivés par restriction d'un type de base, les restrictions valides incluent celles qui augmentent les contraintes sur les attributs ou les éléments du type complexe d'une manière compatible avec le type de base. Par exemple, un élément facultatif dans le type de base peut être requis dans le type dérivé. Ainsi, la dérivation de types simples et complexes par restriction entraîne un ensemble de valeurs autorisées pour le type dérivé qui est un sous-ensemble des valeurs autorisées pour le type de base.

L'extension de types complexes implique la création d'un type complexe dérivé dont le contenu de modèle est un sur-ensemble du contenu de modèle de son type de base, lorsque nous étendons un type complexe, nous pouvons ajouter au type dérivé des attributs ou des éléments supplémentaires en plus de ceux trouvés dans le contenu de modèle du type de base comme l'exemple suivant :

```xml
<xs:complexType name="A">
    <xs:sequence>
    <xs:element name="A1" type="xs:string" />
    <xs:element name="A2" type="xs:string" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name="B">
    <xs:complexContent>
        <xs:extension base="A">
            <xs:sequence>
                <xs:element name="B1" type="xs:string" />
                <xs:element name="B2" type="xs:string" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Figure 51: Création d'un nouvel élément par extension dans XML Schéma

Dans cet exemple, le type *B* est dérivé par extension du type *A*, en plus d'inclure les éléments *A1* et *A2*, le type *B* inclue également les éléments *B1* et *B2*.

Le concept d'extension de type complexe est un mécanisme de « réutilisation » similaire au mécanisme d'héritage dans l'orienté objet, bien que l'extension forme une hiérarchie de types qui prend en charge la substitution d'un type plus spécialisé par un type plus général, elle n'implique pas nécessairement une contrainte de sous-ensemble/sur-ensemble sur l'ensemble des objets représentés par l'extension des types complexes. A côté de son utilisation potentielle pour la généralisation/spécialisation, certains considèrent l'extension de type comme un mécanisme pratique pour écrire du code efficace, et non comme un mécanisme conceptuel pour définir des hiérarchies de généralisation/spécialisation.

2.2. Les groupes de substitution

Dans *l'XML* schéma, les éléments globaux peuvent être organisés en un groupe de substitution, dans lequel un ensemble particulier d'éléments peut être substitué à un élément nommé, appelé l'élément *head*. Par exemple, si les éléments *B* et *C* ont chacun été déclarés substituables à *A* en incluant l'attribut *substitutionGroup=«A»* dans les déclarations des éléments *B* et *C*, alors la signification est que *B* ou *C* peut apparaître partout où *A* est requis. La présence d'un groupe de substitution ne nécessite pas l'utilisation des éléments substituables, et n'empêche pas l'utilisation de l'élément principal, il établit simplement un moyen d'utiliser un ensemble d'éléments de manière interchangeable.

Le concept de groupe de substitution constitue une forme de généralisation / spécialisation, bien qu'elle ne soit pas identique à la notion de sous-ensemble naturel de généralisation / spécialisation. Un groupe de substitution définit une classe d'équivalence d'éléments qui peuvent être utilisés de manière interchangeable. Cependant, les groupes de substitution peuvent former des multiples hiérarchies. En effet, nous disons que l'utilisation d'un groupe de substitution implique une généralisation / spécialisation conceptuelle dans le sens où un concept (un élément substituable) est un type particulier d'un autre concept (l'élément principal).

2.3. Les types et les éléments abstraits

Il est possible d'exiger l'utilisation de la substitution pour un élément ou un type particulier en le déclarant abstrait. Un élément déclaré abstrait ne peut pas être utilisé dans une instance document - un élément substituable non abstrait doit être utilisé à sa place. Ainsi, déclarer un élément comme abstrait nécessite la spécification d'un groupe de substitution. De même, déclarer un type abstrait nécessite l'utilisation de types concrets qui étendent le type abstrait. Dans les deux cas, les éléments abstraits sont associés à des hiérarchies de concepts liées à une relation conceptuelle.

2. Transformation du généralisation/spécialisation de l'XML schéma

Nous nous concentrons sur deux cas de contraintes de généralisation / spécialisation, nous commençons par la contrainte de partition qui peut être représentée dans la figure ci-dessus en utilisant *c-xml* [39]

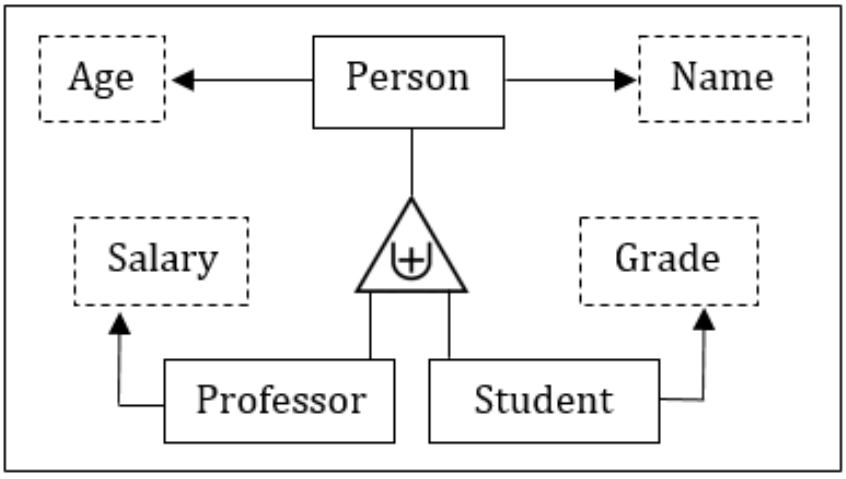


Figure 52: généralisation spécialisation dans C-XML, contrainte de partition

Dans cette section, nous illustrons nos exemples en utilisant *Conceptual XML (CXML)* qui est un modèle conceptuel composé d'un ensemble d'objets, ensembles de relations et de contraintes sur ces ensembles d'objets et de relations. En *C-XML*, nous représentons des ensembles d'objets ou des concepts en écrivant des noms à l'intérieur des rectangles, avec une bordure pleine indiquant un concept non lexical et une bordure en pointillés indiquant un concept lexical.

Dans la figure 52 l'ensemble des objets chez l'élément *professor* et *student* est un sous-ensemble de l'ensemble des objets *person*, *c-xml* nous donne la possibilité d'ajouter une contrainte à la généralisation en écrivant un symbole à l'intérieur du triangle de généralisation / spécialisation, dans notre cas on représente la contrainte de partition en ajoutant le symbole $(\uplus)$ dans le triangle, nous transformons notre exemple *c-xml* en *xml* schéma (figure 53) en suivant le mécanisme de [40].

```
<xs:element name="Person" type="PersonType" abstract="true" />
<xs:complexType name="PersonType">
    <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="age" type="xs:integer" />
    </xs:sequence>
    <xs:attribute name="OID" type="xs:ID" use="required" />
</xs:complexType>
<xs:element name="Student" type="StudentType"
    substitutionGroup="Person" />
<xs:complexType name="StudentType">
    <xs:complexContent>
        <xs:extension base="PersonType">
            <xs:sequence>
                <xs:element name="grade" type="xs:string" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="Professor" type="ProfessorType"
    substitutionGroup="Person" />
<xs:complexType name="ProfessorType">
    <xs:complexContent>
        <xs:extension base="PersonType">
            <xs:sequence>
                <xs:element name="salary" type="xs:decimal" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Figure 53: Transformation de c-xml présenté dans la figure 52

Dans la terminologie des ensembles on dit que :

Student U **Professor** = **Person** and **Student** ∩ **Professor** = {}

student et *professor* sont en état de partition, pour s'assurer que les ensembles sont disjoints, on utilise un identifiant unique *OID (Object Identifier)* pour chaque élément, qui est défini dans l'élément abstrait parent *person*, les *OIDs* jouent le même rôle de clé

```
1  context Student inv
2      Student.allInstances()->forAll(s1,s2 | s1 <> s2
3                                  Implies s1.oid <> s2.oid)
4      Let professorColl : Set = Professor.allInstances()
5      Student.allInstances()->forAll(Student s |
                                                        d))
```

Figure 54: Transformation de la généralisation/spécialisation, contrainte de partition

primaire dans les bases de données relationnelles. Pour transformer l'unicité de l'*OID* et s'assurer que les ensembles de *student* et *professor* sont disjoints, on introduit la clause OCL équivalente :

Le second cas est la contrainte d'exclusion mutuelle, en *c-xml* on remplace le symbole $(\uplus)$ par le symbole **+** (Figure 55) en suivant le mécanisme de transformation du schéma *c-xml* en XML schéma dans, nous conservons le même code de *l'xml* schéma précédent et nous changeons la valeur abstraite en false de l'élément *person.* Nous avons toujours **Student** ∩ **Professor** = {}, mais non plus **Student U Professor = Person,** et nous avons **Student** U **Professor** ⊆ **Person.** Nous avons maintenant la possibilité de créer une instance de *Person* et d'ajouter des invariants pour présenter notre clause *OCL*

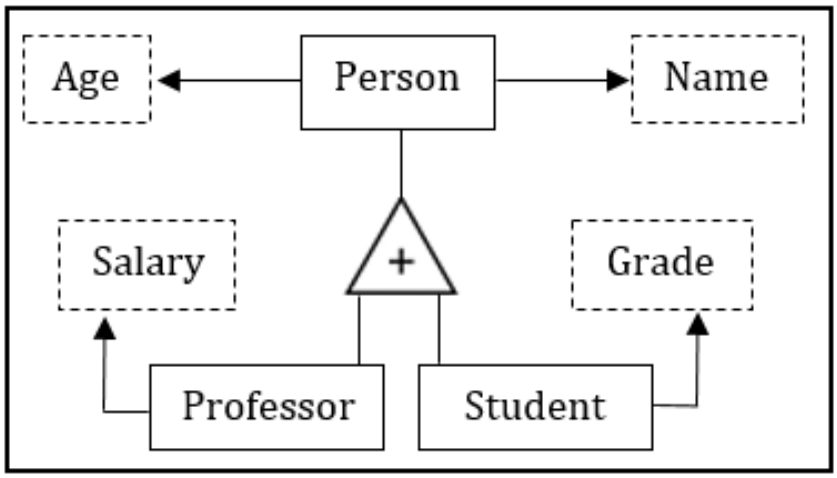


Figure 55: Représentation de la généralisation spécialisation dans C-XML, Exclusion mutuelle

```
1   context Person inv
2   Person.allInstances()->forAll(Person p1,Person p2 |
3           p1 <> p2 Implies p1.oid <> p2.oid)
4
```

Figure 56: Transformation de la généralisation/spécialisation, Exclusion mutuelle

person.allInstances() nous permet d'obtenir toutes les instances de *person*, *student* et *professor* et vérifier le l'unicité de l'*OID* rapidement, contrairement à la contrainte de partition lorsqu'on utilise les collections imbriquées (*student*, *professor*) pour vérifier les différences entre les sous-ensembles.

Dans cette partie, nous avons présenté la transformation de deux cas simples de contraintes de généralisation / spécialisation (partition et exclusion mutuelle) en utilisant la notion de groupe de substitution.

Le processus de transformation n'est pas entièrement satisfaisant dans certains cas, comme la généralisation / spécialisation sans contrainte et la généralisation / spécialisation avec une seule contrainte d'union. Ces deux cas sont plus difficiles à gérer.

Dans les exemples précédents, nous avons présenté une généralisation / spécialisation simple en utilisant les groupes de substitution. Nous ne pouvons pas transformer plusieurs généralisation / spécialisations car nous n'avons aucun moyen de spécifier dans le schéma XML qu'un élément est membre de deux groupes de substitution.

IV. Mapping entre XQuery et OCL

XPath est un langage de requête utilisé pour parcourir un document XML. Il est couramment utilisé pour rechercher des éléments ou des attributs particuliers avec des modèles correspondants.

XPath est une recommandation officielle du *World Wide Web Consortium (W3C).* Il définit une langue pour rechercher des informations dans un fichier *XML*. Il est utilisé pour parcourir les éléments et les attributs d'un document *XML*. *XPath* fournit différents types d'expressions qui peuvent être utilisées pour rechercher des informations pertinentes dans le document *XML*.

Les principales caractéristiques de *XPath* sont :

- Définitions de structure: *XPath* définit les parties d'un document XML telles que les nœuds d'élément, d'attribut, de texte, d'espace de nom, d'instruction de traitement, de commentaire et de document.
- Expressions de chemin : *XPath* fournit des expressions de chemin puissantes pour sélectionner des nœuds ou une liste de nœuds dans des documents XML.
- Fonctions standard : *XPath* fournit une bibliothèque de fonction standard pour la manipulation des valeurs de chaîne de caractère, numériques, booléennes et de comparaison de date/heure, de manipulation des nœuds et des séquences, etc.

- Recommandation W3C : *XPath* est une recommandation officielle du World Wide Web Consortium (W3C).

XQuery est un langage fonctionnel utilisé pour récupérer des informations stockées au format *XML*. *XQuery* peut être utilisé sur des documents *XML*, des bases de données relationnelles contenant des données au format *XML* ou des bases de données *XML*. *XQuery* 3.0 est une recommandation du W3C dès 2014.

Les principales caractéristiques de *XQuery* sont :

- Langage fonctionnel : *XQuery* est un langage pour récupérer / interroger des données *XML*.
- Analogue à *SQL* : *XQuery* est à XML ce que SQL est aux bases de données.
- Basé sur *XPath* : *XQuery* utilise des expressions *XPath* pour naviguer dans les documents *XML*.
- Universellement accepté : *XQuery* est pris en charge par toutes les principales bases de données.
- Norme W3C : *XQuery* est une norme W3C.

Les principaux avantages de *XQuery* :

- En utilisant *XQuery*, les données hiérarchiques et tabulaires peuvent être récupérées.
- peut être utilisé pour interroger des structures arborescentes et graphiques.
- peut être directement utilisé pour interroger et créer des pages Web.
- peut être utilisé pour transformer des documents *XML*.
- *XQuery* est idéal pour les bases de données *XML* et les bases de données d'objets. Les bases de données d'objets sont beaucoup plus flexibles et puissantes que les bases de données purement tabulaires.

Dans *XQuery*, les expressions de chemin sont utilisées pour localiser des nœuds dans les données *XML*, tels que les éléments, les attributs et les nœuds de texte, le résultat de l'expression de chemin est une liste ordonnée de nœuds uniques. Une expression de chemin se compose d'une série d'étapes, chaque étape représente un mouvement à travers un document dans une direction particulière, et peut appliquer un ou plusieurs prédicats pour éliminer les nœuds qui ne satisfont pas à une condition donnée, le résultat de chaque étape est une liste de nœuds qui sert de point de départ pour l'étape suivante [31].

OCL fournit un mécanisme de mouvement afin de naviguer dans une association sur le diagramme de classes pour faire référence à d'autres objets et à leurs propriétés. Pour réaliser ça, nous naviguons dans l'association en utilisant rôle de l'association : *object.roleName*.

Les deux langages, *OCL* et *XQuery* ont le mécanisme de mouvement qui nous aide à récupérer les données, à partir de ce point, nous suggérons de transformer les requêtes *XML* en clauses *OCL*.

Pour comprendre notre *mapping* entre les requêtes *XML* et *OCL*, nous représentons dans la figure 57 un exemple de définition d'un fichier *XML* schéma et le diagramme de classes *UML* correspondant dans la figure 58.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:complexType name="gender">
        <xs:sequence>
            <xs:element name="id" type="xs:string" />
            <xs:element name="name" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
    <xs:element name="catalog">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="catalog_id" type="xs:string" />
                <xs:element name="catalog_Name" type="xs:string" />

                <xs:element name="author" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="first_name" type="xs:string" />
                            <xs:element name="last_name" type="xs:string" />
                            <xs:element name="birht_date" type="xs:date" />

                            <xs:element name="book" minOccurs="1" maxOccurs="unbounded">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="isbn" type="xs:string" />
                                        <xs:element name="title" type="xs:string" />
                                        <xs:element name="description" type="xs:string" />
                                        <xs:element name="price" type="xs:double" />
                                        <xs:element name="price" type="xs:date" />
                                        <xs:element name="gender" type="gender" />
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

Figure 58: Exemple d'un document XML Schema

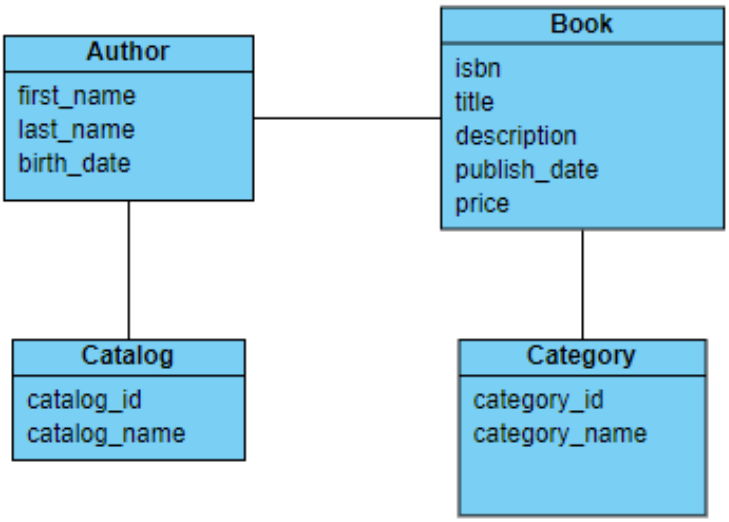


Figure 57: Diagramme de classes UML relatif au document XML figure 57

1. Les expressions Path

Les expressions *Xquery Path* sont utilisées pour localiser les nœuds, tels que les nœuds d'élément, d'attribut et de texte, dans un fichier *XML*. Une expression *Path* se compose d'une série d'une ou plusieurs étapes, séparées par *slash* «/» ou un double *slash* «//». Chaque étape correspond à une séquence de nœuds, nous prenons l'exemple de la liste de tous les titres de livres de tous les auteurs :

```
1    ("books.xml")/catalog/author/book/title
2
```

Figure 59: Exemple d'une expression xpath dans XQuery

En utilisant le mécanisme de navigation entre les objets dans *OCL*, nous pouvons récupérer tous les titres des livres de tous les auteurs. Dans notre diagramme de classes *UML*, nous avons une association un à plusieurs entre la classe *Author* et la classe *Book*, ainsi qu'une association un à plusieurs entre la classe *Catalog* et la classe *Author*, de sorte que la classe *Catalog* contient une collection d'auteurs et la classe *Author* contient une collection de *Book*. L'expression *OCL* est la suivante :

```
1    Catalog.getAuthors().getBooks().getTitle()
2
```

Figure 60: Représentation de la requête xpath dans OCL

2. Les Prédicats

Les prédicats sont utilisés dans les expressions *Path* pour filtrer le résultat en appliquant un test spécifié, ils conservent certains éléments et en ignorent d'autres. *XQuery* utilise des prédicats pour limiter les données extraites du document *XML*. Le prédicat suivant est utilisé pour sélectionner tous les éléments de *Book* qui ont un élément de *price* avec une valeur inférieure à *50* :

```
("books.xml")/catalog/authors/book[price<50]

-- L''expression OCL equivalente

context Book inv :
Book.allInstances()->select(b Book | b.price < 50)

```

Figure 61: Exemple et représentation du XQuery prédicat en OCL

book.allInstances() est de type *Set(Book)* et représente l'ensemble de tous les *Books* qui existent dans le système au moment où l'expression est évaluée. On utilise la méthode *select()* pour spécifier un sous-ensemble d'une collection, le résultat est une liste qui contient tous les éléments de la collection pour lesquels la condition sur le *price* est vraie.

3. Les expressions de séquences & Let

L'expression *Let* nous permet de définir une variable et de l'utiliser plusieurs fois. *XQuery* fournit des séquences pour utiliser une liste d'éléments qui peuvent être très similaires les uns aux autres ou de type différent. *XQuery* prend en charge les opérateurs pour construire, filtrer et combiner des séquences d'éléments, il est noté que les séquences ne sont jamais imbriquées.

L'exemple ci-dessus récupère tous les *Book* dont le *price* est inférieur à 100 du dernier *Author* de notre *Catalog* ainsi que l'expression OCL équivalente.

```
Let $maxPrice :=100
Let $firstAuthor= ("books.xml")/catalog/author[fn:last()]
Let $books=$firstAuthor/book[price<$maxPrice]

-- L''expression OCL equivalente

context Catalog Inv:
Let maxPrice : Integer =100
Let firstAuthor : Author = self.getAuthors()->first()
Let books : Sequence : firstAuthor.getBooks()
->select(b : Book | b.price < maxPrice).asSequence()
```

Figure 62: Exemple et représentation du XQuery let et expressions de séquences en OCL

4. Les expressions arithmétiques

Les expressions arithmétiques effectuent des opérations qui impliquent l'addition, la soustraction, la multiplication, la division et le module *(+, -, *, /)*, le résultat d'une expression arithmétique est une valeur numérique, une séquence vide ou une erreur.

L'exemple ci-dessus montre une opération arithmétique simple dans *XQuery* et l'expression *OCL* équivalente.

```
1   Let $somme=5+9
2
3   -- L''expression OCL equivalente
4
5   Let somme: Integer=5+9
6
```

Figure 63: Exemple et représentation du XQuery arithmetic expressions de séquences en OCL

5. Les expressions de comparaison

Les expressions de comparaison sont utilisées pour comparer des valeurs, il existe trois types d'expressions de comparaison: général, valeur et nœud.

2.1. Comparaison générale

Les opérateurs de comparaison générale sont utilisés pour comparer des valeurs atomiques ou des nœuds contenant une valeur atomique, et peuvent également fonctionner sur des séquences de plus d'un élément, ainsi que des séquences vides. Les opérateurs de comparaison généraux sont *(=,! =, <, <=,>,> =)*, le résultat d'une comparaison générale qui ne génère pas d'erreur est toujours vrai ou faux.

L'expression de comparaison ci-dessus renvoie *true* si au moins le *price* d'un *Book* est inférieur à *150*

```
("books.xml")/catalog/authors/book/price < 50

-- L''expression OCL equivalente

context Book inv:
    Book.allInstances()->exists(b Book | b.price <50)

```

Figure 64: Exemple et représentation du XQuery general comparison en OCL

La méthode *exist()* vérifie s'il y en a au moins un élément dans une collection pour laquelle une contrainte est valable.

2.2. Comparaison de valeurs

La comparaison de valeurs diffère fondamentalement de la comparaison générale en ce qu'elle ne peut fonctionner que sur une seule valeur atomique. Ils utilisent *eq* (égal à), *ne* (différent de), *lt* (inférieur à), le (inférieur ou égal à), *gt* (supérieur à) et *ge* (supérieur ou égal à). L'exemple ci-dessus renvoie *true* s'il n'y a qu'un seul élément *author* dont la valeur *last_name* est égale à *Hugo*

```
("books.xml")/catalog/authors/last_name eq "Hugo"

-- L''expression OCL equivalente

context Author inv:
    Author.allInstances()->exists(a Author | a.getLastName()->size()=1
                                    and a.lastName = "Hugo")

```

Figure 65: Exemple et représentation du XQuery value comparison en OCL

6. Les expressions logiques

Une expression logique est soit une expression *AND*, soit une expression *OR*. Si une expression logique ne génère pas d'erreur, sa valeur est toujours l'une des valeurs booléennes *true* ou *false*. L'exemple ci-dessus obtient tous les *Book* de *Category technology* ou *Science-Fiction*

```
("books.xml")/catalog/authors/book[category_name eq "Technology"
                                   Or
                                    category_name eq "Science Fiction" ]

-- L''expression OCL equivalente

context Book inv:
    Book.allInstances()->select(b Book | b.getCategoryName().equals("Technology")
                               Or
                               b.getCategoryName().equals("Science Fiction"))
```

Figure 66: Exemple et représentation du XQuery logique expression en OCL

7. Les expressions FLWOR

XQuery fournit une fonctionnalité appelée expression *FLWOR* qui prend en charge l'itération et la liaison des variables aux résultats intermédiaires. Ce type d'expression est souvent utile pour calculer les jointures entre deux ou plusieurs documents et pour restructurer des données. *FLWOR* signifie *(for, let, where, order by, return).*

Les clauses *for* et *let* dans une expression *FLWOR* génèrent une séquence ordonnée de *tuples* de variables liées, appelée flux de *tuple*. La clause optionnelle *where* sert à filtrer le flux de tuple, en conservant certains *tuples* et en rejetant d'autres. La clause facultative *order by* peut être utilisée pour réorganiser le flux de *tuple*. La clause *return* construit le résultat de l'expression *FLWOR*. L'exemple ci-dessus obtient le *title* de tous les *Book* dont le *price* est supérieur à 200:

```
For $b in ("books.xml")/catalog/authors/book
Where $b/price>200
order by price
return $b/title

-- L''expression OCL equivalente

context Book inv
    Book.allInstances()->select(b Book | b.price>200)->collect(title)->asSequence()

```

Figure 67: Exemple et représentation du XQuery FLOWR expression en OCL

Lorsqu'on veut spécifier une collection qui est dérivée d'une autre collection, mais qui contient des objets différents de la collection d'origine, on peut utiliser une l'opération *collecte.*

8. Les constructeurs

XQuery fournit des constructeurs qui peuvent créer dynamiquement un nouveau nœud XML (éléments, attributs, texte ...) dans une requête. L'exemple ci-dessus ajoute un nouvel élément *additionalInfo*, qui contient des éléments (*rating* et *priceCategory*).

```
For $b in ("books.xml")/catalog/authors/book
Where $b/price>200
return  <additionalInfo>
            <rating>1</rating>
            <priceCategory>Expensive</priceCategory>
        </additionalInfo>

-- L''expression OCL equivalente

context Book
Def additionalInfo: Set(TupleType(rating: Integer, priceCategory: String))= Book.allInstances()
                    ->select(b Book | b.price>200)->Tuple{b.rating=1, b.priceCategory="Expensive"}

```

Figure 68: Exemple et représentation du XQuery constructors en OCL

L'expression *Def* permet la définition et la réutilisation de variables / opérations sur plusieurs expressions OCL.

TupleType combine différents types en un seul type d'agrégat; les parties d'un *TupleType* sont décrites par ses attributs, chacun ayant un nom et un type.

9. Les expressions conditionnelles

XQuery prend en charge une expression conditionnelle basée sur les mots-clés *if*, *then* et *else*. La valeur d'une expression conditionnelle est définie comme suit :

Si la valeur booléenne effective de l'expression de test est *true*, la valeur de l'expression *then* est renvoyée, si la valeur booléenne effective de l'expression de test est *false*, la valeur de l'expression *else* est renvoyée.

```
Let $technologyBook= count(("books.xml")/catalog/authors/book[
category_name='Technology' ])
if $ technologyBook >0 then
return $technologyBook
else
return “Technology Book List Empty”

-- L''expression OCL equivalente

Let result : String
Let technologyBook: Integer=  Book.allIsntance() ->count("Technolgy")
if technologyBook then
Result = technologyBook
else
Result="Technology Book List Empty"
end if
```

Figure 69: Exemple et représentation du XQuery condition expression en OCL

10. Les expressions quantifiées

Une expression quantifiée détermine si certains ou tous les éléments d'une séquence satisfont à une condition particulière. La valeur d'une expression quantifiée est toujours vraie ou fausse. Une expression quantifiée commence par un quantificateur, qui est le mot-clé *some* ou *every*, suivi d'une ou de plusieurs clauses utilisées pour lier les variables, suivi du mot-clé *satisfy* et d'une expression de test.

Dans l’exemple suivant, l’expression renvoie *true* s'il existe au moins un *Book* de la *Category Math*.

```
Some $book in ("books.xml")/catalog/authors/book Satisfy $book/category_name = "Math"

-- L''expression OCL equivalente

context Book inv
    Book.allInstances()->exists(b Book | b.getCategory().getCategory_name = "Math")

```

Figure 70: Exemple et représentation du XQuery quantified expression en OCL

L'opération *exist()* dans *OCL* permet de spécifier une expression booléenne qui doit tenir au moins un objet dans une collection.

Le deuxième exemple renvoie *true* si toutes les catégories de livres sont *"Math"*

```
Every $book in ("books.xml")/catalog/authors/book Satisfy $book/category_name = "Math"

-- L''expression OCL equivalente

context Book inv
    Book.allInstances()->forAll(b Book | b.getCategory().getCategory_name()="Math")

```

Figure 71: Exemple et représentation du XQuery quantified expression en OCL

L'opération *forAll()* dans OCL permet de spécifier une expression booléenne, qui doit être valable pour tous les objets d'une collection.

11. Les expressions *instance of* et *cast*

Pour déterminer si une séquence d'un ou plusieurs éléments correspond à un type de séquence particulier, on utilise l'expression *instance of*. *Instance of* ne convertit pas une valeur en type de séquence spécifié. Il renvoie simplement vrai ou faux, indiquant si la valeur correspond à ce type de séquence.

```
Let $book =("books.xml")/catalog/authors/book[fn:last()]
$book instance of xs:Integer

-- L''expression OCL equivalente

context Book inv
    Let book : Book = Book.allInstances() ->asSequence()->last()
    Book.ocllsTypeOf(Integer)

```

Figure 72: Exemple et représentation du XQuery instanceOf en OCL

Le *casting* est le processus de changement d'une valeur d'un type à un autre. L'expression de conversion peut être utilisée pour convertir une valeur en un autre type. *XQuery* fournit l'expression *cast* qui crée une nouvelle valeur d'un type spécifique basé sur une valeur existante. L'expression *cast* prend deux opérandes : une expression d'entrée et un type cible.

```
1  Let $price = ("books.xml")/catalog/authors/book[fn:last()]/price
2  $price cast as xs:String
3
4  -- L''expression OCL equivalente
5
6  context Book inv
7      Let price : Integer= Book.allIsntances()->asSequence()
8          ->last().getPrice().oclAsType(String)
9
```

Figure 73: Exemple et représentation du XQuery cast en OCL

V. Conclusion et pistes d'évolution

Dans ce chapitre, nous avons présenté dans la première étape un ensemble de règles de transformation, des facettes contraignantes du schéma *XML* aux expressions *OCL*. Pour valider notre approche, nous fournissons un outil développé en *java* qui prend un fichier *XML Schema* en entrée, après avoir extrait différentes restrictions à l'aide de *Java* et *DOM XML Parser*, on les transforme en des expressions OCL, après nous avons présenté la transformation de la généralisation contraignante (partition et les contraintes d'exclusion mutuelle) à des clauses *OCL*.

Dans la deuxième étape, on capture les expressions *XQuery*, le langage de requête pour les documents *XML*, nous avons présenté les clauses *OCL* équivalentes pour les différentes expressions *XQuery*. Notre prochain objectif sera axé sur la mise en œuvre d'un outil de rétro-ingénierie complet, qui prend un fichier *XML* schéma et un fichier *XQuery* en entrée pour générer après un diagramme conceptuel *UML* enrichi avec des clauses *OCL*.

Chapitre 5 : ORDB Vers NoSQL

I. Introduction

De nombreuses entreprises adoptent le modèle objet relationnel, qui est venu pour corriger la limitation du modèle relationnel. Cette fusion combine entre la maturité du modèle relationnel et la simplicité du modèle objet pour représenter les données du monde réel. En corrigeant la limitation du modèle relationnel, le modèle objet donne aux concepteurs de système et aux administrateurs de bases de données plus de variété pour la modélisation, car ils peuvent se concentrer et penser à un niveau d'abstraction élevé pour fournir une modélisation simple des données.

Au cours des dernières années, les données des utilisateurs se sont rapidement développées, de nouvelles sources de données sont apparues, telles que les données spatiales, le système global de positionnement, le système de surveillance et les réseaux sociaux, et toutes ces sources produisent une grande quantité de données, ce qui donne lieu à des nouveaux défis pour le stockage, la gestion et l'analyse des données.

De plus, les données évoluent vers un format semi-structuré et non structuré, par conséquent les bases de données relationnelles ou objet relationnelles classiques ne peuvent pas stocker ces nouvelles données car elles utilisent un schéma préfixé (tables et colonnes). Pour gérer cette situation, un groupe de bases de données a été créé, appelé bases de données *NoSQL*. Il existe quatre types de base de données *NoSQL*: base de données orienté documents, base de données clé-valeur, base de données orienté colonnes et les bases de données graphiques. Chaque type résout un problème qui ne peut pas être traité par une base de données objet relationnelle.

NoSQL est un nouveau groupe de bases de données apparu ces dernières années comme solution de remplacement des bases de données classiques. Les bases de données *NoSQL* offrent une nouvelle approche pour le stockage et l'extraction de données non structurées autre que la relation tabulaire ou objet adoptée par la base de données relationnelle et objet relationnelle [41]. Les bases de données *NoSQL* sont le choix idéal pour les applications avec une quantité massive de données, en offrant une évolutivité horizontale, une haute disponibilité et des performances puissantes.

Avec l'évolution rapide des données (données complexes, données de réseaux sociaux), de nombreuses entreprises commencent à utiliser les bases de données *NoSQL* pour stocker, gérer et analyser les données. Néanmoins, les majorités gèrent toujours les données dans des bases de données relationnelles ou objet relationnelles. Par conséquent, cette situation donne lieu à des nouvelles techniques de migration des bases de données relationnelles ou objet relationnelles vers les bases de données *NoSQL*. Le modèle de données entre ces deux types de bases de données est totalement différent, les bases de données *NoSQL* sont des bases de données distribuées sans schéma et sans aucune opération de jointure. Pour cette raison, la transformation des modèles et la migration des données sont indispensables pour aider les développeurs d'applications à déplacer leurs données.

Le reste de ce chapitre est organisé comme suit :

- La section 2 présente un aperçu général de l'approche proposée, ici on présente une brève introduction sur l'approche de la transformation et migration qui contient deux principales phases : extraction de modèle et des données et la migration des données.
- la section 3 présente le concept de rétro-conception ou le *reverse engineering* des bases de données ainsi que le rôle important de cette opération dans la compréhension de la base de données avec les différentes tables et relations à travers la récupération d'un modèle conceptuel à partir d'un modèle physique de données.
- La section 4 décrit en détail notre approche proposée de la transformation du modèle et de la migration des données, ici on présente les différences entre la base de données objet relationnelle Oracle et les bases de données *NoSQL* et on introduit par la suite le data modèle qui joue le rôle du pont et de l'intermédiaire entre ces deux types hétérogènes de base de données, par la suite on propose un ensemble de règles de transformation de Oracle vers *NoSQL*
- la section 5 présente une mise en œuvre et l'implémentation de notre approche, ici on présente l'environnement de notre solution ainsi que l'outil développé en Java qui prend en paramètre les informations de connexion aux bases de données (*Oracle 12c* et *MongoDB*) avant d'effectuer la transformation et la migration des données.

II. Aperçu général de l'approche proposée

Notre méthode de transformation comporte deux phases principales. L'objectif principal de la première est l'extraction de modèles; dans cette étape, on se concentre sur la partie structurelle du schéma objet relationnel, on récupère toutes les informations nécessaires du dictionnaire de la base de données, telles que le type de définition utilisateur *UDT (NOM, SUPER TYPE, FINAL ou non, INSTANTIABLE ou ABSTRACT, ...),* tables basées sur les types, dépendances entre les tables, on identifie également tous les types de collections *(NESTED TABLE, VARRAY).*

La deuxième phase se concentre sur la transformation du modèle et la migration des données en utilisant un modèle de données prédéfini, qui joue le rôle de pont entre les deux bases de données *(Object Relational Database et NoSQL).* Dans notre approche, le modèle de données est utilisé comme une couche d'intégration, qui assure la connexion entre les deux bases de données hétérogènes et gère la communication du modèle relationnel objet au modèle orienté document ou orienté colonne de *NoSQL*. Dans les sections suivantes, on présente en détail le modèle de données proposé et les règles de transformation des objets *ORACLE* vers *MongoDB*.

La figure ci –dessous décrit notre approche proposée de migration en présentant les deux phases de transformations.

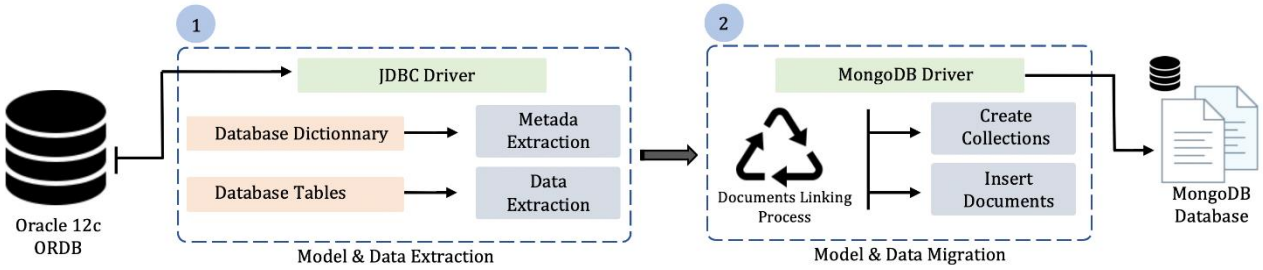


Figure 74: Aperçu général de l'approche proposée

III. Le Reverse Engineering

Le *reverse engineering* représente l'étude et l'analyse d'un système pour en déduire son fonctionnement interne, et se retrouve dans de nombreux domaines de l'ingénierie. Dans le domaine des bases données, faire du *reverse engineering* revient à élaborer un processus de transformation où tout composant doit se transformer. La transformation est équivalente au processus de migration d'une base de données source à une base de données cible, respectant la transformation du schéma physique, le *mapping* des données, les requêtes, l'intégrité des données, la sécurité des données, et l'indépendance des données.

Avec la croissance des données des 4V *(Volume, Variété, Vélocité, Value),* la gestion des données évolue de la scalabilité verticale vers la scalabilité horizontale pour une augmentation en puissance. Des centaines de milliers de petits serveurs créent l'informatique répartie au lieu d'une seule machine puissante. Les données doivent être transformées en un modèle différent pour soutenir l'informatique distribuée.

Les données devraient être plus variées, donc la flexibilité est nécessaire pour correspondre au format natif de l'information, elles peuvent être stockées dans des documents, graphiques, ou de bases de données clé/valeur. La structure des données change avec les entreprises agiles. Mais un schéma prédéfini fort est limité par ce genre de scénario d'entreprise. Modification d'une colonne existante ou ajouter des nouvelles colonnes nécessite de recréer la table dans les bases de données relationnelles, mais il est flexible pour ajouter un nouvel attribut ou un objet composite dans la base de données *NoSQL*.

La migration se base sur une structure en couche, chaque couche est construite autour de la couche sous-jacente pour fournir une abstraction détaillée. Il faut commencer avec l'étude du modèle physique de la base de données source, concevoir un modèle pour la migration, et au final procéder à la migration du schéma physique et le data mapping.

Le modèle de données : est une forme aplatie de la base de données source, qui décrit abstraitement comment sont représentées les données. Enrichi avec les caractéristiques sémantiques nécessaires et souhaitées, qui sont conformes aux exigences, extraites de la base de données source, afin d'établir la migration vers le modèle ou la technologie souhaitée.

Schéma et migration des données : la migration du schéma est une description de l'ensemble de données, qui s'appuie sur un modèle de données spécifiques. La migration

est réalisée grâce à un système d'automatisation, doté d'une capacité d'abstraction totalement intuitive, et un accès total à la base de données s'appuyant sur la collecte et l'enrichissement sémantique.

La migration se base sur l'extraction des concepts objet du modèle de données, pour exploiter d'une façon concrète les objets pour parvenir à établir des références entre les différentes collections, également de stocker de manière idéale toute forme de structure de données, qui nécessitait un ensemble de jointures en logique relationnelle. Les tables dans le modèle objet relationnel sont traduites sous forme de collections et pour chaque enregistrement un document est créé respectant la structure de données champ-valeur. Les documents correspondent aux objets JSON, et les valeurs d'un champ peuvent inclure d'autres documents, des tableaux, ou même des tableaux de documents. Le NoSQL a un schéma flexible, ce qui permet de réaliser la création du schéma physique et le mapping des données au fur et à mesure respectant une syntaxe précise.

IV. Transformation et migration des données d'Oracle vers MongoDB

Avant de présenter en détail notre processus de transformation et de migration de données, dans cette section nous allons décrire les bases de données traitées dans notre travail afin de présenter le concept de chacune via ses propres objets et composants: la célèbre base de données orienté document *NoSQL* pour stocker et gérer les données semi-structurées et non structurées (*MongoDB*) et la base de données objet relationnelle pour le stockage et la gestion des données classiques et d'objets (*Oracle 12C*).

1. Base de données Objet Relationnelle (Oracle 12c)

Oracle 12c est un système de gestion de base de données objet relationnelle développé et produit par *Oracle Corporation* entre autres solutions, qui tirent parti de la maturité du modèle relationnel et des concepts orientés objet. Les types d'objets Oracle sont des types définis par l'utilisateur qui permettent de représenter des entités du monde réel, telles que des personnes et des adresses, en tant qu'objets dans la base de données.

Pour organiser et accéder aux données de la base de données, *Oracle* propose des méthodes et des mécanismes de niveau supérieur tels que les types d'objet et les fonctionnalités orientées objet techniques, telles que *varray* et les tables imbriquées. Sous

la couche d'objet, les données sont comme d'habitude stockées dans des colonnes et des tables, mais Oracle nous donne la possibilité de travailler avec les données en termes d'objet du monde réel qui rendent les données compréhensibles et faciles à gérer. Au lieu d'analyser en termes de modèle relationnel (tables et colonnes). Par conséquent, à titre d'exemple, lorsque nous avons besoin de récupérer des données sur les clients, on sélectionne simplement le type d'objet client créé précédemment. En général, avec la base de données objet relationnelle, nous pouvons utiliser les fonctionnalités orientées objet tout en travaillant avec la plupart des données de manière relationnelle (solution hybride), ou utiliser entièrement une approche orientée objet.

Basé sur les types de base de données intégrés, les types d'objet crées par les utilisateurs, les références d'objet ou les collections, on peut créer de nouveaux types d'objet. Les types d'objets peuvent traiter des données complexes, telles que des données spatiales et multimédias. En général, le modèle de type objet est équivalent au mécanisme de classe trouvé sur les langages de programmation orienté objet. Avec les classes, on peut réutiliser les objets, il est donc possible de créer et de gérer l'application de base de données rapidement et plus dynamiquement. En prenant en charge les types d'objets, Oracle permet aux développeurs d'applications de gérer directement les structures de données créées par leurs applications.

La figure 75 montre les principaux composants objets de la base de données Oracle.

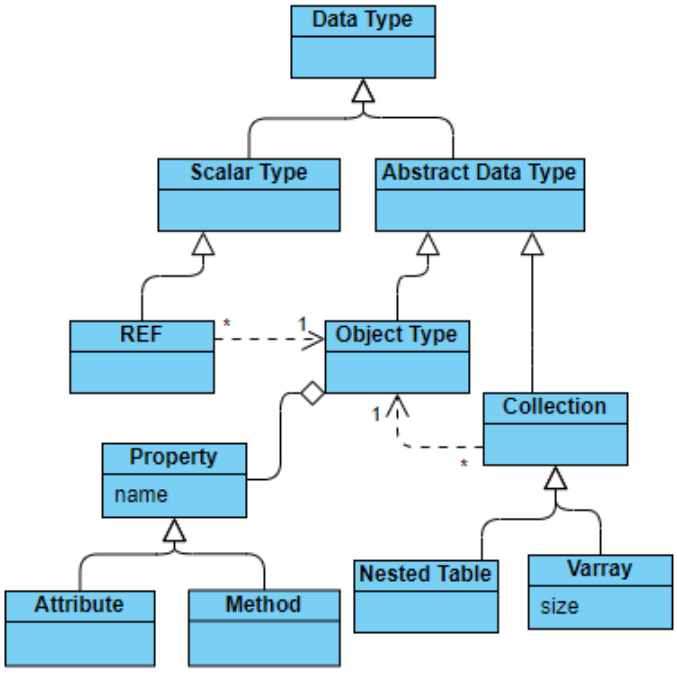


Figure 75: le meta-model des objets principaux dans Oracle 12c

2. Base de données orientée document (MongoDB)

MongoDB est une base de données *NoSQL* open source basée sur une structure orienté document. Pour stocker et gérer des données complexes, comme les fichiers binaires volumineux tels que les vidéos et les images dans un environnement distribué à haute vitesse, *MongoDB* prend en charge les structures de données BSON ainsi qu'un langage de requête puissant. Au lieu d'utiliser des procédures et des méthodes stockées, les développeurs peuvent stocker et utiliser des fonctions et des valeurs *Javascript* côté serveur [42].

MongoDB contient un ensemble de collections. Une collection n'a pas de conception prédéfinie comme les tableaux classiques et stocke les données sous forme de documents. Un document est un ensemble de champs et d'attributs et peut être représenté sous forme de ligne dans une collection. Il peut contenir des structures de données complexes telles que des listes ou un ensemble de documents. Pour identifier les documents, *MongoDB* ajoute pour chaque document un champ ID, qui est utilisé comme clé primaire. Chaque collection peut contenir différents types de documents, mais les requêtes et les index ne peuvent être effectués que sur une seule collection. La figure 76 montre la structure de *MongoDB*

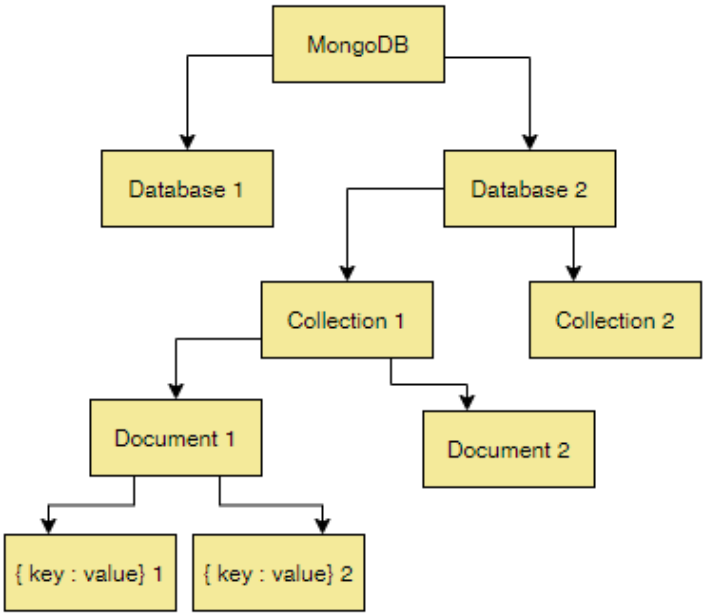


Figure 76: Structure la base de données MongoDB

2.1. Modélisation des données avec MongoDB

La modélisation des données est un processus très important lors de la conception d'une application, cette étape permettra de définir les conditions nécessaires à la construction de la base de données. Cette définition est précisément le résultat de la compréhension des données acquises au cours du processus de modélisation des données.

Un gros avantage de *NoSQL* par rapport aux bases de données relationnelles ou objet relationnelle est que les bases de données *NoSQL* sont plus souples, en raison de la possibilité d'avoir un modèle sans schéma, qui en théorie, peut causer moins d'impact sur la vue de l'utilisateur si une modification dans le modèle de données est nécessaire.

Les bases de données *NoSQL* mettent en pratique la nécessité d'un rapprochement entre les professionnels de la base de données et les applications, ainsi que la nécessité d'un rapprochement entre les développeurs et les bases de données. Ce chapitre présentera le modèle de données *MongoDB* ainsi que les principaux concepts et les structures disponibles pour le développement de ce modèle

2.2. Document et collection

MongoDB a le document en tant qu'unité de base de données. Les documents dans *MongoDB* sont représentés en *JavaScript Object Notation (JSON).* Les collections sont des groupes de documents, une collection est similaire à une table dans un modèle relationnel et un document est un enregistrement dans ce tableau. Et enfin les collections appartiennent à une base de données dans *MongoDB.* Les documents sont sérialisés sur le disque dans un format connu sous le nom *Binary JSON (BSON),* une représentation binaire d'un document *JSON*.

Contrairement au modèle relationnel, où on déclare la structure de la table, une collection ne fait pas respecter une certaine structure d'un document. Il est possible qu'une collection contienne des documents avec des structures complètement différentes.

En plus de cela, une autre caractéristique intéressante de *MongoDB* est que non seulement les données sont représentées par des documents. Fondamentalement, toutes les interactions de l'utilisateur avec *MongoDB* sont faites à travers des documents. Outre l'enregistrement de données, les documents sont un moyen pour:

- Définir quelles données peut être lues, écrites, et / ou mis à jour dans les requêtes.

- Définir les champs qui seront mis à jour.
- Créer des index.
- Interroger les informations de la base de données.

2.3. JSON

JSON (JavaScript Object Notation – Notation Objet issue de JavaScript) est un format léger d'échange de données. Il est facile à lire ou à écrire pour des humains. Il est aisément analysable par des machines. Il est basé sur un sous-ensemble du langage de programmation *JavaScript (JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999). JSON* est un format texte complètement indépendant de tout langage, mais les conventions qu'il utilise seront familières à tout programmeur habitué aux langages descendant du *C*, comme par exemple : *C* lui-même, *C++, C#, Java, JavaScript, Perl, Python* et bien d'autres. Ces propriétés font de *JSON* un langage d'échange de données idéal [43].

Le format *JSON* est également considéré comme très convivial et lisible par l'homme. *JSON* ne dépend pas de la plate-forme choisie et ses caractéristiques sont basées sur deux structures de données:

- Un ensemble ou un groupe de paires clé / valeur.
- Une liste de valeurs ordonnée.

Ainsi, afin de clarifier tous les doutes, nous allons parler des objets. Les objets sont une collection non ordonnée de paires clé / valeur qui sont représentées par le schéma suivant:

{

"key" : "value"

}

En ce qui concerne la liste des valeurs ordonnée, une collection est représentée comme suit:

["value1", "value2", "value3"]

Dans la spécification de *JSON*, une valeur peut être:

- Une chaîne délimitée par "".

- Les valeurs booléennes (vrai ou faux).
- Une valeur nulle.
- Un autre objet.
- Un autre tableau de valeurs ordonné.

2.4. BSON

BSON [44], abréviation de *Binary JSON*, est une sérialisation binaire codée de documents *JSON*. Comme *JSON*, *BSON* soutient l'intégration des documents et des tableaux dans d'autres documents et des tableaux. *BSON* contient également des extensions qui permettent la représentation des types de données qui ne font pas partie de la spécification JSON. Par exemple, BSON a un type de date et un type *BinData*.

BSON peut être comparé à des formats d'échange binaires. BSON est *"schéma moins"*, ce qui peut lui donner un avantage en termes de flexibilité, mais aussi un léger désavantage en matière d'efficacité d'espace. En outre, l'une de ses caractéristiques c'est qu'il est léger, une fonctionnalité qui est très importante pour le transport de données sur le Web.

Le format *BSON* a été conçu pour être facilement navigable et à la fois encodé et décodé d'une manière très efficace pour la plupart des langages de programmation qui sont basés sur *C*. Ceci est la raison pour laquelle *BSON* a été choisi comme format de données de persistance pour *MongoDB*.

Les types de représentation des données en *JSON* sont:

- *String UTF-8 (string)*
- *Integer 32-bit (int32)*
- *Integer 64-bit (int64)*
- *Floating point (double)*
- *Document (document)*
- *Array (document)*
- *Binary data (binary)*
- *Boolean false (\x00 or byte 0000 0000)*
- *Boolean true (\x01 or byte 0000 0001)*
- *UTC datetime (int64)*
- *Timestamp (int64)*

- *Null value ()*
- *Regular expression (cstring)*
- *JavaScript code (string)*
- *JavaScript code w/scope (code_w_s)*
- *Min key()*
- *Max key()*
- *ObjectId (byte*12)*

2.5. Caractéristiques des Documents

Le nom d'un champ dans un document est une chaîne. Nous avons certaines restrictions sur les noms des champs qui doivent être respectés. Elles sont:

- Le champ _id est réservé pour une clé primaire.
- On ne peut pas commencer le nom en utilisant le caractère *$*.
- Le nom ne peut pas avoir un caractère *nul, ou (.)*

En outre, les documents qui ont indexé les champs doivent respecter la limite de taille pour un champ indexé. Les valeurs ne peuvent pas dépasser la taille maximale de *1024 octets*.

2.5.1. La clé primaire du document

Le champ *_id* est réservé à la clé primaire. Par défaut, ce champ doit être le premier dans le document, même si, au cours d'une insertion, il n'est pas le premier champ à insérer. Dans ce cas, *MongoDB* le déplace à la première position.

Le champ *_id* peut avoir une valeur qui est un type *BSON*, à l'exception de la matrice. En outre, si un document est créé sans indication du champ *_id*, *MongoDB* va créer automatiquement un champ *_id* de type *ObjectId*. Cependant, ce n'est pas la seule option. On peut toujours utiliser toute valeur identifiant notre document aussi longtemps qu'elle est unique.

2.5.2. Concevoir un document

Comme mentionné précédemment, les collections ne forcent pas à définir la structure du document au préalable. Mais cela est certainement une décision à faire dans un moment donné. Cette décision aura un impact sur des aspects importants, notamment dans le domaine concernant la performance d'une requête.

Maintenant, on cherche comment les applications représentent une relation entre les documents. Jusqu'à maintenant, la plupart sont habitués à concevoir le monde relationnel, comme se demander quelle est la relation entre un étudiant et leurs classes ou entre un produit et de ses commandes.

MongoDB a aussi sa propre façon de représenter ce genre de relation. En fait, il existe deux façons:

- Les documents intégrés ou les documents embarqués
- Les références

Les documents embarqués

Grâce à l'utilisation des sous-documents, on peut construire des structures de données plus complexes et optimisés. Ainsi, lors de la modélisation d'un document, on peut choisir d'intégrer les données liées dans un seul document.

La décision d'intégrer les données dans un seul document est souvent liée à l'intention d'obtenir une meilleure performance de lecture, car avec une seule requête, nous pouvons complètement récupérer les informations dont nous avons besoin.

Voir l'exemple suivant :

```
{
id: "1",
fname: "Fouad",
lname:  "TOUFIK",
        address: [
                {
                country: "Morocco",
                street : "21 Fake Street",
                }
          ]
}
```

L'avantage de ce type de document est que, avec une seule requête, nous avons toutes les données que nous devons présenter à l'utilisateur. Cela vaut également pour les mises à jour : avec une seule requête, nous pouvons modifier le contenu de ce document. Néanmoins, lorsque nous décidons d'intégrer des données, nous devons nous assurer que le document ne dépasse pas la limite de taille de *BSON* de *16 MB*.

Les références

La normalisation est un processus fondamental pour aider à construire des modèles de données relationnelles. Afin de minimiser la redondance, dans ce processus, nous divisons les grandes tables en plus petites et de définir les relations entre eux. Nous pouvons dire que la création d'une référence dans *MongoDB* est la façon dont nous devons «normaliser» notre modèle. Cette référence décrit la relation entre les documents.

Vous pouvez être confus au sujet de la raison pour laquelle nous envisageons des relations dans un univers non-relationnel, même si cela ne signifie pas que les relations ne sont pas présentes dans les bases de données *NoSQL*. Nous allons très souvent utiliser les concepts de modélisation relationnelle pour résoudre des problèmes communs. Comme indiqué précédemment, afin d'éliminer la redondance, les documents peuvent se référer les uns aux autres.

MongoDB possède un langage de requête simple et puissante nommée *Mongo Query Language*, la figure 77 montre les requêtes de base pour la gestion des documents, après avoir créé une base de données et une collection pour contenir les nouvelles données insérées.

```
>>> use person_database
switched to db person_database
>>> db.createCollection("person_collection")
{ "ok" : 1 }
>>> db.person_collection.insert({lname:"TOUFIK",fname:"Fouad"})
WriteResult({ "nInserted" : 1 })
>>> db.person_collection.find({})
{ "_id" : ObjectId("5f57a04db40484fc796a4df8"), "lname" : "TOUFIK", "fname" : "Fouad" }
>>> db.person_collection.remove({lname:"TOUFIK"})
WriteResult({ "nRemoved" : 1 })
>>> db.person_collection.drop()
true
>>>
```

Figure 77: Exemple de requête basique dans Online MongoDb Web Shell

MongoDB offre l'API Atlas en respectant les principes du style architectural *REST (Representational State Transfer)* pour exposer différentes ressources internes, ce qui permet un accès par programme aux fonctionnalités d'Atlas. Avec cette puissante *API*, *MongoDB* expose toutes les entités au format JSON et prend en charge un sous-ensemble de méthodes *HTTP* courantes telles que *(GET, POST, PUT, PATCH, DELETE et HEAD)* pour échanger des données. *MongoDB* offre d'autres fonctionnalités telles que l'accès basé sur la clé, *Digest authentication* et *HTTPS – Only* pour s'assurer que la clé *API* utilisée n'est jamais envoyée sur le réseau et s'assurer que toutes les données envoyées sur le réseau sont entièrement cryptées à l'aide de TLS et de nombreuses autres fonctionnalités.

3. Comparaison de MongoDB et Oracle 12c

Les bases de données SQL sont appelées bases de données relationnelles ou bases de données objets relationnels et utilisent une structure de données basée sur une table ou un objet, avec un schéma prédéfini. Les bases de données *NoSQL* ou bases de données non relationnelles, peuvent être des bases de données orienté colonnes, des bases de données de paires clé-valeur, des bases de données basées sur des documents ou des bases de données graphiques. Sans utiliser de schéma prédéfini, les bases de données *NoSQL* permettent de travailler librement avec des données non structurées. Les bases de données relationnelles ou les bases de données objets relationnels sont évolutives verticalement, mais généralement plus coûteuses, tandis que la nature de mise à l'échelle horizontale des bases de données *NoSQL* est plus rentable.

Le tableau 3 montre les principales différences basées sur les termes et les concepts entre Oracle 12c en tant que base de données objet relationnelle et *MongoDB* en tant que base de données de documents *NoSQL*.

Oracle SQL Termes / Concepts	MongoDB Termes/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	$lookup, embedded documents $lookup Performs a left outer join to an unsharded collection in the ***same*** database
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline
SELECT INTO NEW_TABLE	$out can take a document to specify the output database as well as the output collection

Table 3: les composants d'Oracle 12c database vs Mongo database

Le tableau 4 montre les principales différences basées sur les instructions de schéma entre *Oracle 12c* et *MongoDB*

Oracle SQL Schema statements	MongoDB Schema Statements
create type person as object(..)	db.createCollection('person')
create type employe as object(age number, name varchar2)	db.employe.insert({ age:'30', name: 'TOUFIK'})
insert into address(street) values ('2 Fake St')	db.address.insert({ street: '2 Fake St})
select * from employe where name like 'TOUFIK'	db.employe.find({name : 'TOUFIK'})
delete from employe where name like 'TOUFIK'	db.employe.remove({name : 'TOUFIK'})
drop table employe	db.employe.drop()

Table 4: les instructions d'Oracle 12c database vs Mongo database

Dans la section suivante, nous présentons le modèle de données *(Data Model)* proposé qui joue un rôle très important en tant que pont entre la base de données objet relationnelle Oracle 12c et la base de données de documents *MongoDB.*

4. Définition du Data Model

La base de données orientée documents *MongoDB* et la base de données objet relationnelle sont des bases de données hétérogènes, avec des objectifs et une orientation différente. Une couche d'intégration est donc indispensable pour établir la connexion et assurer la communication entre les deux bases de données. Cette couche est basée sur un *Data Model* bien défini, qui joue le rôle de pont entre la base de données objet relationnelle et *MongoDB* pour échanger les données et les métadonnées. La définition des éléments de la base de données objet relationnelle est une étape cruciale, dans le but de les transformer à une base de données orientée documents *MongoDB.* Les éléments sont caractérisés par l'expression dans la figure suivante :

Type_Schema = { T | T:=Tn, A, S_type, Ms, Rel } where:

- **Tn** = Name of the User Definition Type
- **A** = denotes a set of class attributes

  A = { An, T, Dc } each attribute has a name **An**, a type **T** (int, char, date …), and Declarative Constraint **Dc** (Primary key, Unique)
- **S_type** = represents the name of the super type if existed
- **Ms** = defines a set of methods declared inside User Definition Type (**UDT**)
- **Rel** = A relationship where a Type T is participating, where each Type has a set of relationships with other Type

  REL = { rel | rel:= RelType, DirT, Cn } where

  RelType represent a type of relationship which supports four types: inheritance, association, aggregation and composition.

  DirT is the name of the type **T'** interacts with Type.

  Cn defines a cardinality to identify if the attribute can hold one value or multi-values.

Figure 78: Définition du Data Model

Association: est la relation entre deux ou plusieurs types. Il est utile de spécifier la navigabilité entre les objets. La relation d'association est définie par l'expression ci-dessous où *RelType = «association »:*

T = {Tn, At, T'n, Cn, Ms, "association"} où :

Tn est le nom du type; ***At*** est un ensemble d'attributs du type ***T***; ***T'n*** est le nom du type ***T'*** interagissant avec le type ***T***; ***Ms*** est un ensemble de méthodes définies dans le type ***T***, et enfin ***Cn*** qui définit la cardinalité du type ***T***.

Aggregation: est une association binaire représentant une relation *tout / partie*. Où la *partie* est partageable et indépendante du type. La relation d'agrégation est définie par l'expression ci-dessous où *RelType = "agrégation"*

T = {Wn, At, Pn, Cn, Ms, «aggregation»} où:

Wn définit le nom de tout le Type ***T*** composé de zéro ou de plusieurs objets de la *partie* ***P***, ***Pn*** est le nom des objets de la *partie* compose l'ensemble du Type ***T***.

Composition: est une forme forte d'agrégation, où l'élément *partie* est physiquement inclus dans le type. Si un composite (tout) est supprimé, toutes ses parties composites sont supprimées avec lui. La relation de composition est définie comme suit où *RelType = «composition»:*

T= { Wn, At, Pn, Cn, Ms, "composition" }

Inheritance: est une relation de généralisation / spécialisation entre le super-type général (supérieur dans la hiérarchie) et les sous-types (inférieur dans la hiérarchie). La relation d'héritage est présentée comme suit:

Inh_Rel = {STn, SBn, "inheritance"} où:

STn est le nom du type super ou général et ***SBn*** est le nom du sous-type.

Dans l'approche que nous proposons, pour créer ce modèle, la couche d'intégration et de transformation est responsable de l'extraction des métadonnées de la base de données et de la récupération des tables basées sur les types et des informations des UDT (User Definition Type). Nous utilisons un groupe de classes proposé par *JDBC (Java Database*

Connectivity) pour interroger le dictionnaire de base de données Oracle 12c. Le modèle obtenu représente une définition claire du schéma objet relationnel, qui nous aide à définir un ensemble de règles et d'étapes de transformation.

5. Règles de transformation de l'ORDB vers MongoDB

Oracle 12c est un *ORDBMS (Object Relational Database Management System)*, qui permet aux utilisateurs de créer leurs propres objets de données prédéfinis, en spécifiant la structure et le comportement des données à l'aide des méthodes et des attributs des objets, les objets créés appelés *UDT (User Definition Type).* Les *UDTs* permettent aux utilisateurs de créer des données complexes du monde réel, un UDT stocke les données métier dans leur représentation naturelle.

La base de données *MongoDB* contient un ensemble de collections, chaque collection contient des données sous forme de documents et n'a pas de schéma prédéfini. Le document est un ensemble de champs et peut être représenté sous forme de ligne dans la base de données objet relationnelle, les documents peuvent contenir des données complexes telles que des listes ou même une liste des documents.

L'exemple ci-dessous montre la structure et l'instruction de création de *l'UDT Address* dans Oracle :

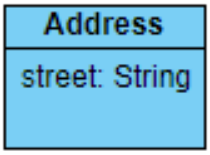


Figure 79: Représentation d'un Simple UDT

CREATE TYPE address_type AS OBJECT(street VARCHAR(20), ...);

CREATE TABLE address_table OF address_type;

En exécutant l'instruction *create*, nous avons la possibilité de créer une table objet relationnelle basée sur le type défini par l'utilisateur. Chaque *UDT* est mappé à une collection *MongoDB*, dans cet exemple, le nom de la collection est *address_type* avec le

même nom que le type créé. La table typée *address_table* contient des lignes de données de l'objet *address_type*; la collection *address_type* contiendra des objets *address_type* sous forme de document au format *BSON (Binary encoded JSON).* Le code ci-dessus montre comment créer la collection *address_type* dans *MongoDB.*

db.createCollection("address_type")

Chaque cardinalité d'un type d'objet défini dans notre modèle de données proposé, joue un rôle important dans la gestion des règles de transformation dans cette section. La cardinalité d'un type d'objet participant à une relation indique le nombre d'occurrences qu'une instance de ce type peut être associée dans la relation. Chaque type d'objet *ObjType1* participe à une relation (association, agrégation ou composition) avec un autre type d'objet *ObjType2*, donne lieu à une relation entre les documents de la collection *ObjType1* et les documents de la collection *ObjType2*. Cependant, quel est le type de lien entre ces documents, est ce que le *referencing* ou *l'embedding*.

Dans une base de données objet relationnelle, un type d'objet peut contenir un autre type d'objet, et nous avons la possibilité de les référencer. Oracle a quatre différentes représentations de la déclaration de l'UDT au sein d'un autre UDT :

- Simple *UDT*.
- *REF* d'un simple *UDT*.
- Collection des *UDT*.
- Collection de REF des *UDTs*.

Nous prenons l'exemple de *UDT address_type* créé auparavant, et le plaçons comme attribut dans un nouvel *UDT* appelé *employee_type*. Cette définition de *employee_type* avec *address_type* à l'intérieur, elle représente une relation *one to one*.

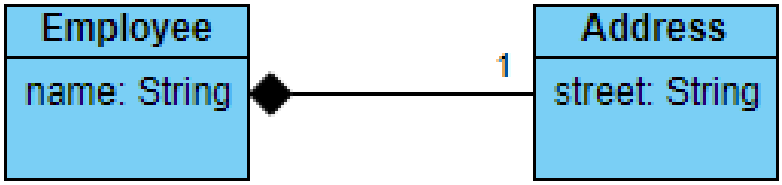


Figure 80: Représentation de la relation un à un (composition)

CREATE TYPE employee_type AS OBJECT(address address_type , name VARCHAR(20), ...);

Les relations dans *MongoDB* représentent la façon dont différents documents sont logiquement connectés les uns aux autres. Les relations peuvent être conçues via des approches embarquées *(embedded)* ou référencées *(referenced)*. Les documents embarqués utilisent les relations entre les données en stockant les données associées dans un seul document. Ces modèles de données dénormalisés permettent aux applications d'extraire et de gérer les données associées en une seule requête. Si les documents référencés gèrent les relations entre les données en incluant des liens ou des références d'un document à un autre, nous parlons ici de modèles de données normalisés [42].

La transformation génère une nouvelle relation de composition entre les deux *UDTs employee_type* et *address_type* avec une multiplicité *(0..1)* du côté *address_type*. La composition est une représentation d'une forte association entre deux entités. Par conséquent, un document *address_type* sera inclus dans le document *employee_type*. L'exemple ci-dessus montre le résultat de la transformation et la structure de *employee_type*.

```
1   {
2   name: "TOUFIK",
3   address :
4       {
5       street: "21 Fake St"
6       }
7    }
8
```

Figure 81: Transformation de la relation un à un (embedding)

En utilisant le mécanisme de références entre différents *UDT*, une autre représentation *d'UDT* est possible dans *l'ORDB*. Les références permettent de créer des objets complexes et de récupérer des données facilement sans utiliser les opérations *JOIN* comme dans les bases de données classiques. Les références créent une relation faible entre les types d'objets et évitent définitivement les problèmes de la redondance.

L'exemple ci-dessus présente une relation de composition entre *employee_type* et *address_type*, en ajoutant le mot clé ***REF*** à l'attribut *address_type* dans la définition de *employee_type*, la transformation génère une relation d'agrégation avec la même multiplicité mentionnée précédemment.

Figure 82: Représentation de la relation un à un (aggregation)

CREATE TYPE employee_type AS OBJECT(address REF address_type, name VARCHAR(20), ...);

Après cette mise à jour, nous devons changer la relation entre le document *employee_type* et *address_type* de l'intégration *(embedding)* au référencement. Par conséquent, nous utilisons *address_type* comme document référencé et conservons les informations d'adresse dans une collection distincte de la collection *employee_type*. Le type *employee_type* contient un champ qui référence le champ d'identification du document *address_type*.

```
1   {
2       _id : 102030,
3       street: "21 Fake Street"
4   }
5   {
6       name: "TOUFIK",
7       address_type: 102030
8   }
9
```

Figure 83: Transformation de la relation un à un (referencing)

Oracle 12c propose des types de données comme les tables imbriquées *(Nested Table)* et les collections *varray*. *Nested table* est un ensemble ordonné d'éléments de données avec le même type de données, *nested table* est une collection illimitée. *Varray* est un ensemble ordonné d'éléments avec le même type de données, *varray* est une collection limitée.

Nous pouvons changer la relation entre *employee_type* et *address_type* de *(one to one) à (one to many)* en définissant l'attribut adresse comme *varray* ou *nested table* dans *employee_type* UDT, l'exemple ci-dessus montre les différents cas de déclaration d'attribut d'adresse:

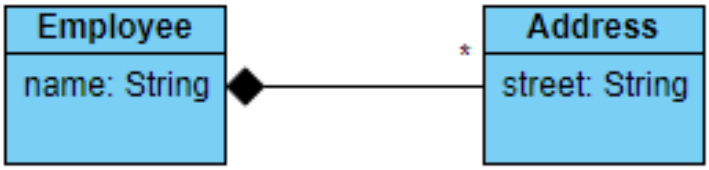


Figure 84: Représentation de la relation un à plusieurs (composition)

Cas 1:

CREATE TYPE address_type_varray AS VARRAY(n) OF address_type;

CREATE TYPE employee_type AS OBJECT(address_list address_type_varray);

Cas 2:

CREATE TYPE address_type_nested ASTABLE OF address_type;

CREATE TYPE employee_type AS OBJECT(address_list address_type_nested);

Cas 3:

CREATE TYPE address_ref AS OBJECT(addr_r REF address_type)

CREATE address_varray AS VARRAY(n) OF address_ref; CREATE address_nested AS TABLE OF address_ref;

CREATE TYPE employee_type AS OBJECT(address_list1 address_varray,...)

CREATE TYPE employee _type AS OBJECT(address_list2 REF address_nested,...)

L'attribut *address_list* défini dans *employee_type* est une collection de *address_type*, la transformation génère une relation de composition entre *employee_type* et *address_type* avec *(0 .. *)* multiplicité côté *adresse_type*, en cas de *nested table (cas 1)*, dans le cas de *varray* on change la multiplicité à *(0..n)* où ***n*** est la taille de la collection *(cas 1).*

Après cette modification, la relation entre le document *employee_type* et *address_type* passe à la relation *(one to many)*. Ainsi, le document *employee_type* contient un tableau intégré de document *address_type*, le résultat de la relation (one to many):

```
1   {
2       name: “ TOUFIK ”,
3       address :[
4               { street: “ 21 Fake Street”},
5               { street: “ 215 street 20”},
6               ]
7   }
8
9
```

Figure 85: Transformation de la relation un à plusieurs (embedding)

Lorsque l'attribut *address_type* est une collection de références UDT *(REF) (cas 3)*, nous conservons toutes les propriétés obtenues de l'exemple précédent et nous modifions la relation entre *employee_type* et *address_type* de la composition à l'agrégation. Par conséquent, nous changeons la relation entre le document *employee_type* et *address_type* de *l'embedding* au *referencing*, l'exemple ci-dessus présente la dernière transformation :

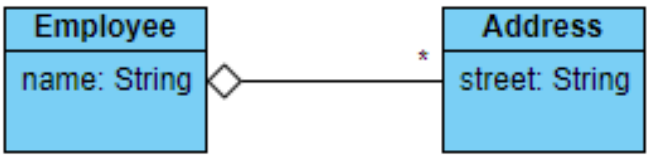


Figure 86: Représentation de la relation un à plusieurs (aggregation)

```
{
    _id : 1234,
    street: "24 Fake Street"
}
{
    _id : 5678,
    street: "10 Street 9"
}

{
    name: "TOUFIK",
    address:[1234,5678]
}

```

Figure 87: Transformation de la relation un à plusieurs (referencing)

V. Implémentation et Validation

Dans cette section, nous présentons l'environnement expérimental que nous avons mis en place pour valider notre approche.

MongoDB utilise trois composants importants, à savoir les serveurs de configuration, les routeurs de requêtes et les nœuds de partition.

- *Sharding* est le processus de stockage de données sur différentes machines et c'est l'un des puissants mécanismes de *MongoDB* répondant à la nécessité de stocker une grande quantité de données. Avec l'énorme évolution croissante des données, une seule machine peut ne pas être suffisante pour stocker des données ni offrir un temps d'exécution de lecture et d'écriture tolérable. Avec *Sharding*, *MongoDB* permet d'ajouter plus de machines pour prendre en charge le stockage massif de données et fournir une haute disponibilité et une cohérence des données, cette action appelée mise à l'échelle horizontale.
- Les serveurs de configuration représentent un groupe de serveurs utilisés pour stocker les métadonnées et les informations de routage du cluster *MongoDB* en spécifiant quelles données sont disponibles dans quelle

partition. Dans un environnement de production, les clusters fragmentés ont exactement trois serveurs de configuration.

- Les routeurs de requête sont des instances *MongoDB* qui communiquent avec l'application cliente et envoient des opérations au fragment approprié. Le routeur de requête traite et dirige les opérations vers les fragments et renvoie les résultats aux clients.

Avec l'augmentation rapide des données, l'utilisation d'un stockage distribué est fortement recommandée. Pour tels systèmes de stockage, un cluster est nécessaire, mais sa configuration et le déploiement des services sont difficiles. L'installation et la configuration du système d'exploitation et de l'environnement d'exécution du logiciel adéquats et l'utilisation de toutes les dépendances requises sont une tâche difficile. De plus, nous avons un ensemble diversifié de matériel serveur à côté de la différence entre les paramètres de test et de production.

Pour résoudre le problème de la préparation de l'environnement expérimental, nous utilisons Docker. Docker est un projet open source qui automatise le déploiement d'applications à l'intérieur de conteneurs logiciels [45], la version initiale de Docker date de mars 2013. Docker est un service de gestion de conteneurs, qui aide les développeurs à développer facilement des applications, à les expédier dans différents conteneurs, qui peut être déployé n'importe où, il a également la capacité de réduire la taille du développement en utilisant une version plus petite des systèmes d'exploitation via des conteneurs, ce qui les rend facilement évolutifs. La figure 88 montre l'architecture des couches de Docker.

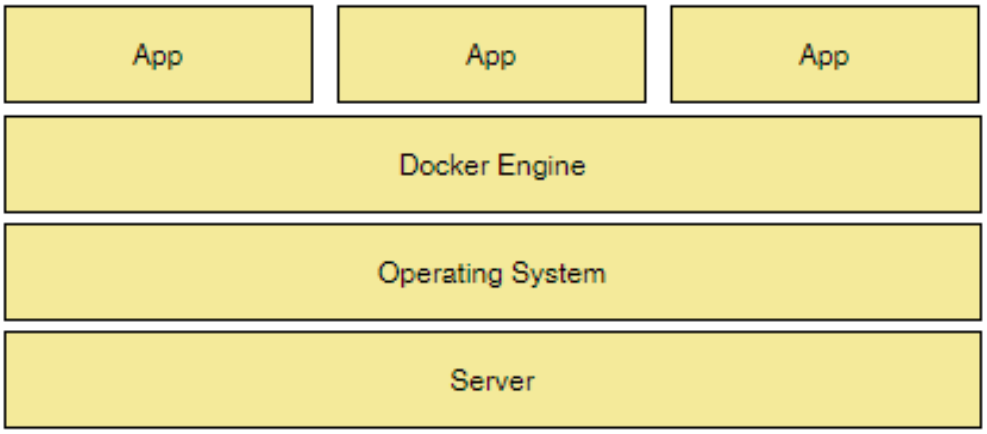


Figure 88: Architecture de Docker

Comme mentionné précédemment, nous utilisons *Docker* pour configurer notre environnement, la figure 89 montre l'architecture de notre solution.

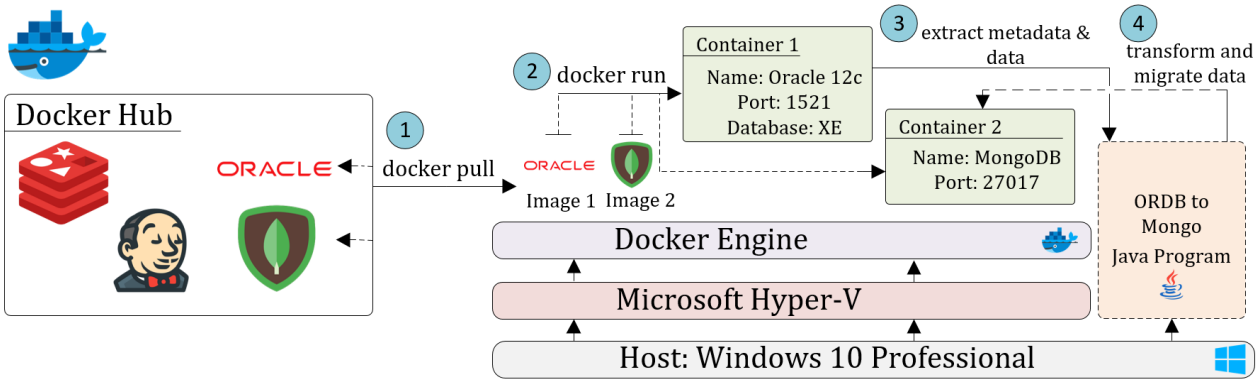


Figure 89: Architecture de la solution proposée

Docker Hub est un référentiel basé sur le cloud fourni par *Docker* pour télécharger et partager des images de conteneurs créées par d'autres membre de communauté. Grâce à *Docker Hub*, un utilisateur peut accéder aux référentiels d'images sources publiques, extraire et pousser des conteneurs d'images. Dans notre cas, on télécharge du *docker hub* une image préconfigurée *d'Oracle Standard Edition 12c Release 2* sur *CentOS*, et on télécharge également une image officielle de *MongoDB* en exécutant ces deux commandes docker pour chaque image.

docker pull mongo
docker pull truevoly/oracle-12c

Après avoir télécharger les deux images, nous pouvons exécuter chaque image pour créer un conteneur en exécutant la commande *docker run*.

docker run mongo
docker run -d -p 8080:8080 -p 1521:1521 truevoly /oracle-12c

Le port par défaut pour MongoDb est 27017

Pour le conteneur *Oracle,* le port *8080* est utilisé pour se connecter à la console de gestion Web *Oracle Application Express* et le port 1521 est le port par défaut pour se connecter à la base de données Oracle *XE*. L'utilisateur *truevoley* explique en détail comment configurer et exécuter le conteneur oracle 12 [46].

Pour démontrer la validité de notre approche, un outil a été développé pour présenter la transformation du modèle et la migration des données en se basant sur la méthode proposée. Pour développer notre prototype, nous utilisons *java* comme langage de programmation, pour avoir une application multiplateforme et portable. Pour établir la connexion avec l'instance Oracle et plus précisément le schéma utilisateur, cet outil prend un ensemble de paramètres tels que : adresse *IP* du serveur hébergeant la base de données Oracle, port, schéma, nom d'utilisateur et le mot de passe. Ces paramètres concernent *l'Oracle 12c* comme base de données source, l'outil prend également les informations de connexion de *MongoDB* comme la base de données cible (adresse IP, nom de base de données et port). Les expériences ont été réalisées sur *PC* avec *3,70 GHz AMD Ryzen 5 3400G* avec *16 G de RAM et windows 10 professionnel*. L'outil utilise Oracle *JDBC (Java Database Connectivity)* pour établir une connexion avec la base de données Oracle et exécuter un ensemble de requêtes sur la table de dictionnaire Oracle pour obtenir des métadonnées du type défini par l'utilisateur et de la table objet relationnelle, en utilisant *mongodb-driver*, l'outil établit également une connexion avec *MongoDB* et créer des collections et des documents contenant des données selon les modèles et les données extraits lors de la phase précédente.

Nous présentons ci-dessous un exemple basique et simple de transformation *d'Oracle 12c UDT* et de migration des données stockées dans des tables typées vers des collections *MongoDB* avec des documents contenant des données.

La figure 90 montre l'interface du programme développé *(Oracle 12c To MongoDB Transformation Program)*

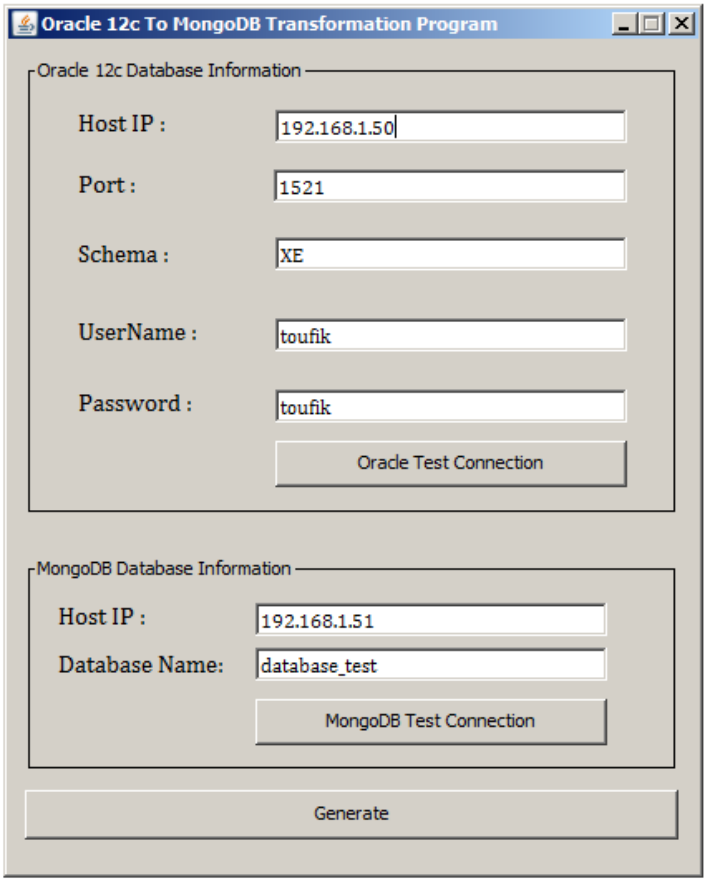


Figure 90: Oracle 12c to MongoDb Transformation Program

Les requêtes dans la figure 91 créent une table typée appelée *employee_table* basée sur l'objet oracle *employee_type*, qui contient *address_type* comme attribut, et insèrent des lignes de données dans la table *employee_table.*

```
1  CREATE TYPE address_type AS OBJECT
2      (city varchar(20),street varchar(250));
3
4  CREATE TYPE employee_type AS OBJECT
5      (lname varchar(20),fname varchar(20)
6          ,address address_type);
7
8  CREATE TABLE employee_table OF employee_type;
9
10 insert into employee_table('TOUFIK','Fouad',
11     address_type('Marrakech','24 Fake Street'));
12 insert into employee_table('BAHAJ','Mohammed',
13     address_type('Settat','’125 Street number 21'));
```

Figure 91: Requêtes de création et d'insertion Oracle 12c

La figure 92 montre le résultat de la transformation et les données obtenues d'Oracle 12c vers MongoBD.

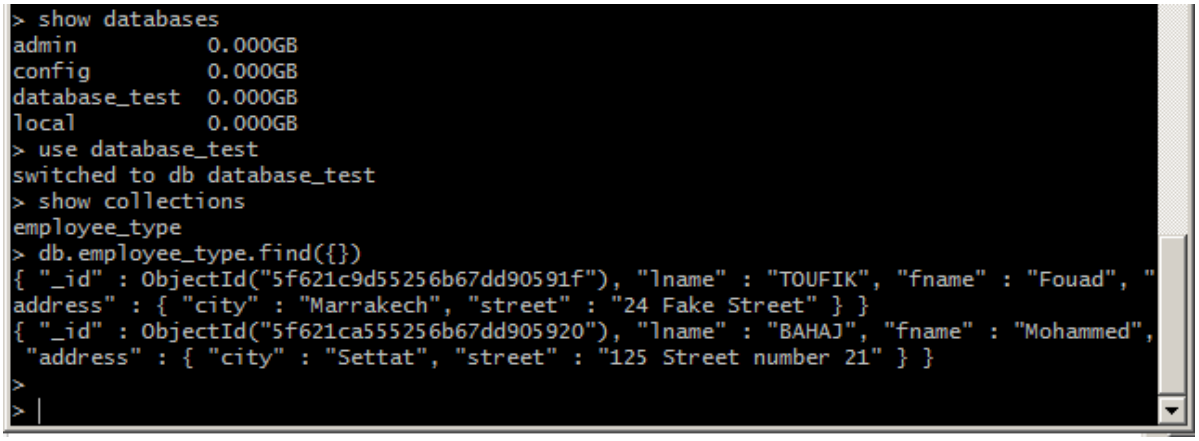


Figure 92: Résultat de transformation de l'UDT employee_type

VI. Conclusion et pistes d'évolution

Dans ce chapitre, nous avons proposé une approche de conversion de schéma et de migration de données de la base de données objet relationnelle Oracle 12c vers la base de données orientée document *MongoDB*, notre approche comporte deux étapes principales : extraction de modèle de données et migration de données. En définissant un modèle de données, qui joue le rôle de pont entre les deux bases de données, et en suivant un ensemble de règles de transformation, nous avons développé un programme pour valider notre approche de conversion de schéma et migration des données *d'Oracle 12c* vers *MongoDB*.

Une extension évidente de notre recherche est d'inclure les mécanismes d'optimisation de notre outil de transformation et de migration de données. Une autre promesse concernant nos futurs travaux est de couvrir le reste des bases de données *NoSQL* telle que les bases de données orienté Graph, et de fournir aux utilisateurs un outil complet de transformation d'Oracle 12c en bases de données *NoSQL*.

Conclusion générale

Avec l'évolution rapide des applications informatique, une masse importante de données est produite sous forme structurée, semi structurée et non structurée par plusieurs sources de données telles que les réseaux sociaux *(facebook, twitter, linkedin, ...)*, ce qui rend la compréhension et le stockage de ces données de plus en plus un défi.

Durant ces dernières années de nombreuses méthodes et approches ont été proposées pour comprendre les bases de données à l'aide du *reverse engineering* ainsi que la migration du schéma et des données à partir des bases de données relationnelles classiques vers les bases de données *NoSQL*. Cependant, et face à l'ampleur du nombre de sources de données hétérogènes, on trouve les bases de données objet relationnelle et *XML* qui nécessite aussi la compréhension et la migration.

Le travail présenté dans ce manuscrit est une contribution à la résolution de ce problème. Il se base sur trois axes principaux, le reverse engineering d'une base de données objet relationnelle, la transformation des *constraining facets* définit dans un document *XML schéma* avec les requêtes *XQuery* à des expressions *OCL* et finalement la transformation et la migration des données d'une base de données objet relationnelle à une base de données *NoSQL*.

Notre première contribution vise à élaborer une solution de *reverse engineering* qui prend en entrée une base de données objet relationnelle et génère comme résultat dans un niveau élevé d'abstraction un schéma conceptuel exprimé en diagramme de classes *UML* et enrichi avec des expressions *OCL*, les expressions *OCL* obtenues présente les contraintes déclaratives définit au niveau des tables de la base de données ainsi que les contraintes définit sous forme de *triggers*.

Dans le même but de compréhension des données et de structure pour aider les concepteurs et les développeurs à maintenir leur application, la deuxième contribution consiste à proposer une méthode et développer un outil qui permet de transformer les *constrainng facets* et les requêtes *XQuery* à des expressions *OCL* équivalentes.

Notre troisième contribution se focalise sur la transformation du schéma et la migration des données d'une base de données objet relationnelle à une base de données *NoSQL* en respectant certaines règles et étapes à travers un modèle de données qui joue le pont et l'intermédiaire entre les deux bases de données qui sont hétérogènes.

En perspectives générales, nous envisageons d'optimiser les différentes parties de nos approches proposées et les rendre plus flexible afin d'effectuer le *reverse engineering* et la migration des données dans un délai optimal.

Pour enrichir davantage l'outil développé en *java* de rétro conception d'une base de données objet relationnelle *oracle 12c*, nous envisageons d'ajouter les expressions *OCL* qui représentent les contraintes et les triggers définies au diagramme de classes *UML* obtenu.

Pour la partie de la transformation du schéma et la migration des de données d'une base de données objet relationnelle à une base de données *NoSQL*, nous envisageons d'ajouter la migration vers une base de données *clé/valeur NoSQL*

BIBLIOGRAPHIE

[1] Hainaut, J-L, Database Reverse Engineering, Models, Techniques and Strategies, in Preproc. of the 10th Conf. on Entity-Relationship Approach, San Mate0 (CA), 1991 Premerlani, W.J., Blaha, M.R: An approach for reverse engineering of relational databases. In Working Conference on Reverse Engineering. (May 1993) pages 151-160.

[2] Roger H. L. Chiang, Terence M.Barron, Veda C. Storey: Reverse Engineering of relational databases: extraction of an EER model from relational database. In Data & Knowledge Engineering (March 1994) pages 107-142.

[3] Anderson, M: Extracting an entity relationship schema from a relational database through reverse engineering. In Entity Relationship approach (1994) pages 403-419.

[4] Ramanathan, S. Hodges, J: Extraction of object-oriented structures from existing relational databases. In ACM Sigmod Record (March 1997) pages 59-64.

[5] David W. Embely: Relational database reverse engineering: A model-centric, transformational, interactive approach formalized in model theory. In DEXA'97 Database and Expert Systems Applications (1997) pages 372-377

[6] H. Balsters, Modelling Database Views with Derived Classes in the UML/OCL-framework in P. Stevens, ≪UML≫2003 - The Unified Modeling Language. Modeling Languages and Applications, Volume 2863 of Lecture Notes in Computer Science; Springer, 2003, pp 295- 309.

[7] D. H. Akehurst, B. Bordbar, On Querying UML Data Models with OCL in M. gogolla, ≪UML≫ 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Volume 2185 of Lecture Notes in Computer Science ; Springer, 2001, pp 91-103.

[8] Chaparro. O, Aponte. J, Ortega. F, Towards the Automatic Extraction of Structural Business Rules from Legacy Databases in IEEE Working conference on reverse engineering, 2012, pp 479 – 488.

[9] D. Yeh, Y. Li, and W. Chu, "Extracting entityrelationship diagram from a table-based legacy database," Journal of Systems and Software, vol. 81, no. 5, pp. 764 – 771, 2008.

[10] Alhajj, R., "Extracting the extended entityrelationship model from a legacy relational database", Information Systems, Elsevier Science Ltd., 2002, pp.597-618.

[11] Cosentino and S. Martinez: Extracting UMLOCL Integrity Constraints and Derived Types From Relational Databases. In the 13th International Workshop on OCL, Model Constraints and Query Languages, Miami, United States, (2013) pages 43-52.

[12] Rajani Chennamaneni, Emanuel S. Grant : Comparison and Evaluation of Methodologies for Transforming UML Models to ObjectRelational Databases, University of North Dakota. 2002.

[13] Wai Yin Mok, David P. Paper: On Transformations from UML Models to ObjectRelational Databases. In the 34th Annual Hawaii International Conference on system science. (HICSS-34) Volume 3 (Jan 2001) p.3046

[14] Cabot J., Gomez C., Planas E. and Rodriguez M. E., Reverse Engineering of OO Constructs in Object-Relational Database Schemas, JISBD 2008, (2008)

[15] Object Constraint Language Specification v 2.4 OCL.2.4 http://www.omg.org/spec/OCL/2.4/PDF/

[16] C. Haitao, "A Survey to Conceptual Modeling for XML", Proc, 2010 3rd International Conference on Computer Science and Information Technology, Volume 8, 2010, pp 473-477.

[17] M. Necasky, "Reverse Engineering of XML Schemas to Conceptual Diagrams", Proceedins 6th Asia Pacific Conference on Conceptual Modeling, Volume 96, 2009, pp 117-128.

[18] Aman, H., Ibrahim, R XML Schema Reverse Transformation: A Case Study, in D. Taniar, C. Torre, computational Science and Its Application – ICCSA 2015, Volume 9158 of Lecture Note in Computer Science , Springer 2015, pp 575-586.

[19] Aman, H., Ibrahim, R.: Formalization of transformation rules from XML schema to UML class diagram. International Journal Software Engineering and it Application 8(12), 75–90 (2014)

[20] Aman, H., Ibrahim, R.: Reverse engineering: from XML to Uml for generation of software requirement specification. In: 2013 8th International Conference on Information Technology in Asia - Smart Devices Trend: Technologising Future Lifestyle, Proceedings Of CITA 2013 (2013)

[21] Grady Booch, Magnus Christerson, Mathew Fuchs, Jari Koistinen; "UML for XML Schema Mapping Specification"; Rational Software Corp. and CommerceOne Inc., December 1999.

[22] BIRD, L., GOODCHILD, A. and HALPIN, T. (2000): Object Role Modeling and XML Schema. Proc. International Conceptual Modeling Conference, Salt Lake City, USA, 309-322, Springer.

[23] I-Chen Wu, Shang-Hsien Hsieh; An UMLXML-RDB Model Mapping Solution for Facilitating Information Standardization and Sharing in Construction Industry Proceedings. National Institute of Standards and Technology, Gaithersburg, Maryland. September 23-25, 2002, pp. 317-321

[24] Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach Information and Software Technology, Volume 53, Issue 1, January 2011, Pages 34-50

[25] Thomas Kudrass, Tobias Krumbein, RuleBased Generation of XML DTDs from UML Class Diagrams in L. Kalinichenko,R. Manthey ,B. Thalheim,U. Wloka, Advances in Databases and Information Systems, Volume 2798 of Lecture Notes in Computer Science, Springer,2003, pp. 339-354

[26] E. Kuikka, A. Eerola, A Correspondence between UML Diagrams and SGML/XML DTDs in P. King, E. V. Munson, Digital Documents: Systems and Principles, volume 2023 of Lecture Notes in Computer Science; Springer, 2004, pp. 161-175

[27] Mikael R. Jensen, Thomas H. Moller, Torben Bach Pedersen. Converting XML Data to UML Diagrams for Conceptual Data Integration. DIWeb'2001. pp.17~31

[28] M. Bernauer, G. Kappel, G. Kramler, Representing XML Schema in UML – A Comparison of Approaches, in: N. Koch, P. Fra-ternali, M. Wirsing (Eds.), Web Engineering, Vol. 3140 of Lecture Notes in Computer Science, Springer, 2004, pp. 440- 444.

[29] F.Salim, R. Price, M. Indrawan, S. Krishnaswamy, Graphical Representation of XML Schema, in Xuemin Lin, Hongjun Lu, Yanchun Zhang, Advanced Web Technologies and Applications, volume 3007 of Lecture Notes in Computer Science, Springer, 2004, pp. 234-245

[30] Object Constraint Language Specification v 2.4 OCL.2.4 URL:http://www.omg.org/spec/OCL/2.4/PDF/

[31] Apache Software Foundation http://sqoop.apache.org/

[32] M. Casters, R. Bouman, and J. Van Dongen, Pentaho Kettle solutions: building open source ETL solutions with Pentaho Data Integration. John Wiley & Sons, 2010.

[33] Chung, Wu-Chun, LIN, Hung-Pin, Chen, Shih-Chang, et al. JackHare: a framework for SQL to NoSQL translation using MapReduce. Automated Software Engineering, 2014, vol. 21, no 4, p. 489-508.

[34] C. Li, "Transforming relational database into hbase: A case study," in Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on. IEEE, 2010, pp. 683–687.

[35] Lawrence, Roger. Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB. In : Computational Science and Computational Intelligence (CSCI), 2014 International Conference on. IEEE, 2014. p. 285-290.

[36] T. Jia, X. Zhao, Z. Wang, D. Gong, G. Ding, "Model transformation and data migration from relational database to mongodb", 2016 IEEE International Congress on Big Data (BigData Congress), pp. 60-67, June 2016.

[37] G. Zhao, W. Huang, S. Liang, and Y. Tang, "Modeling mongodb with relational model," in Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth InternationalConference on. IEEE, 2013, pp. 115–121.

[38] David W. Embley, Stephen W. Liddle, and Reema Al-Kamha. Enterprise Modeling with Conceptual XML. In Proceedings of the 23rd International Conference on Conceptual Modeling (ER2004), pages 150–165, Shanghai, China, November 2004.

[39] R. Al-Kamha, D. W. Embley, and S. W. Liddle. Representing Generalization/Specialization in XML Schema. In EMISA, 2005

[40] Z.Gansen, L.Qiaoying, L.Libo, L.Zijing, Schema Conversion Model of SQL Database to NoSQL, 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing.

[41] MongoDB Documentation https://docs.mongodb.com/

[42] https://www.json.org/

[43] http://bsonspec.org/

[44] Q. Liu, W. Zheng, M. Zhang, Y. Wang and K. Yu, "Docker-Based Automatic Deployment for Nuclear Fusion Experimental Data Archive Cluster," in *IEEE*

Transactions on Plasma Science, vol. 46, no. 5, pp. 1281-1284, May 2018, doi: 10.1109/TPS.2018.2795030.

[45] https://github.com/MaksymBilenko/docker-oracle-12c.

[46] Toufik Fouad and Bahaj Mohamed. 2019. Model Transformation From Object Relational Database to NoSQL Document Database. In Proceedings of the 2nd International Conference on Networking, Information Systems & Security (NISS19). Association for Computing Machinery, New York, NY, USA, Article 49, 1–5. DOI:https://doi.org/10.1145/3320326.3320381.

[47] Karnitis, Girts et Arnicans, Guntis. Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation. In : Computational Intelligence, Communication Systems and Networks (CICSyN), 2015 7th International Conference on. IEEE, 2015. p. 113-118.

[48] Wei-Ping, Zhu, Ming-Xin, Li, et Huan, Chen. Using MongoDB to implement textbook management system instead of MySQL. In : Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on. IEEE, 2011. p. 303-305.

[49] Mior, Michael J. Automated schema design for NoSQL databases. In : Proceedings of the 2014 SIGMOD PhD symposium. ACM, 2014. p. 41-45.

[50] Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, vol. 26, no 2, p. 4.

[51] LOESING, Simon, PILMAN, Markus, ETTER, Thomas, et al. On the Design and Scalability of Distributed Shared-Data Databases. In : Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015. p. 663-676.

[52] Rupali Arora ,Rinkle Rani Aggarwal; An Algorithm for Transformation of Data from MySQL to NoSQL (MongoDB); IJASCSE, Volume 2, Special Issue 1, 2013

[53] Parker, Zachary, POE, Scott, et Vrbsky, Susan V. Comparing nosql mongodb to an sql db. In : Proceedings of the 51st ACM Southeast Conference. ACM, 2013. p. 5.

[54] CASTELLTORT, Arnaud et LAURENT, Anne. Representing history in graph-oriented nosql databases: A versioning system. In : Digital Information

Management (ICDIM), 2013 Eighth International Conference on. IEEE, 2013. p. 228-234.

[55] HSU, Jen-Chun, HSU, Ching-Hsien, CHEN, Shih-Chang, et al. Correlation Aware Technique for SQL to NoSQL Transformation. In : Ubi-Media Computing and Workshops (UMEDIA), 2014 7th International Conference on. IEEE, 2014. p. 43-46

[56] STONEBRAKER, Michael. SQL databases v. NoSQL databases. Communications of the ACM, 2010, vol. 53, no 4, p. 10-11.

[57] F. Bugiotti, L. Cabibbo, R. Torlone, and P. Atzeni. Database design for NoSQL systems.Inria & Universite Paris-Sud and Universita Roma Tre.

[58] MANOJ, V. Comparative study of NoSQL Document, Column Store Databases And Evaluation Of Cassandra. International Journal of Database Management Systems (IJDMS), 2014, vol. 6, no 4, p. 11-26.

[59] ABADI, Daniel, BONCZ, Peter, HARIZOPOULOS, Stavros, et al. The design and implementation of modern column-oriented database systems. Now, 2013.

[60] Rastegar-Mojarad, Majid, LI, Dingcheng, et LIU, Hongfang. Operationalizing Semantic Medline for meeting the information needs at point of care. AMIA Summits on Translational Science Proceedings, 2015, vol. 2015, p. 152.

[61] Lee, Rubao, Luo, Tian, Huai, Yin, et al. Ysmart: Yet another sql-to-mapreduce translator. In : Distributed Computing Systems (ICDCS), 2011 31st International Conference on. IEEE, 2011. p. 25-36.

[62] LIU, Si, NGUYEN, Son, GANHOTRA, Jatin, et al. Quantitative Analysis of Consistency in NoSQL Key-Value Stores. In : Quantitative Evaluation of Systems. Springer International Publishing, 2015. p. 228-243.

[63] Clarence J M Tauro, Baswanth Rao Patil, K. R. Prashanth; A Comparative Analysis of Different NoSQL Databases on Data Model, Query Model and Replication Model; International Conference on Emerging Research in Computing, Information, Communication and Applications, ERCICA 2013, At Bangalore, India, Volume: 1

[64] Bogdan George Tudorica, Cristian Bucur; A comparison between several NoSQL databases with comments and notes ; In Roedunet International Conference (RoEduNet), 2011 10th (pp. 1-5). IEEE.

[65] LOURENÇO, João Ricardo, ABRAMOVA, Veronika, VIEIRA, Marco, et al. NoSQL Databases: A Software Engineering Perspective. In : New Contributions

in Information Systems and Technologies. Springer International Publishing, 2015. p. 741-750.

[66] Gajendran, S.: A Survey on NoSQL Databases, http://ping.sg/story/A-Survey-on-NoSQL-Databases---Department-of-Computer-Science, 2012

[67] BOAG, Scott, CHAMBERLIN, Don, FERNÁNDEZ, Mary F., et al. XQuery 1.0: An XML query language. 2002.

[68] XML (2008). World Wide Web Consortium (W3C). Extensible Markup Language (XML). https://www.w3.org/TR/REC-xml/.

[69] WANG, Chunyan, LO, Anthony Chiu Wa, ALHAJJ, Reda, et al. Converting Legacy Relational Database into XML Database through Reverse Engineering. In : ICEIS (1). 2004. p. 216-221.

[70] Chodorow, Kristina. MongoDB: the definitive guide. " O'Reilly Media, Inc.", 2013.

[71] REESE, George. Database Programming with JDBC and JAVA. " O'Reilly Media, Inc.", 2000.