Cycle Pluridisciplinaire d'Etudes Supérieures
Paris Sciences et Lettres
Université Paris-Dauphine

Le *mapping* objet-relationnel *ou* Le problème de la sauvegarde des objets

mémoire de recherche effectué dans le cadre du cours de Recherche du S6 par

Kamil Moreau

Tuteur de mémoire : Maude Manouvrier
Maître de Conférences, Université Paris-Dauphine

Doyenne de la formation : Isabelle Catto

Date de soutenance : 1er juillet 2016

Remerciements

Je tiens particulièrement à remercier Maude Manouvrier qui, malgré toutes les obligations prenantes qu'elle a eues cette année, a accepté de m'encadrer pour ce mémoire, et m'a proposé un sujet intéressant, et très appliqué. Le sujet n'était pas évident à traiter, car enseigné à partir du Master 2 et au-delà ; mais l'aide qu'elle m'a fournie lorsque j'en ai eu besoin m'a permis de dépasser un certain nombre de difficultés et d'arriver à comprendre les problèmes d'un domaine en plein essor, qu'est la persistance des données. Bien entendu, je suis toujours loin de maîtriser le sujet, qui nécessiterait qu'on s'y accorde à plein temps, ce que je n'ai hélas pu faire cette année ; mais j'ai pu acquérir des bases qui me serviront certainement par la suite, vu le parcours dans lequel je m'engage et l'avenir qu'a le monde des données.

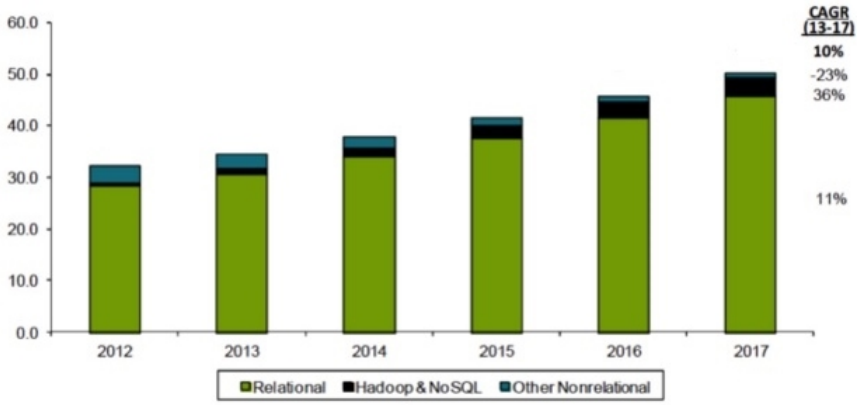
Plan

1 Introduction

Avec le développement de l'informatique, et la création du langage informatique Java, on a pu représenter des objets réels sous forme d'objets informatiques, Ainsi, dans la vie courante, un objet peut être un compte bancaire, un client, un produit... les exemples sont multiples. Le succès du langage informatique Java, et de ce qu'on appelle la programmation objet, est bien dû à toutes les applications concrètes qu'ils permettent ; comme s'en vante Oracle (la maison mère des logiciels Java) sur son site internet [21], plus de 9 millions de développeurs écrivent en Java, et des milliards d'appareils l'utilisent aujourd'hui !

Or, un objet informatique disparaît à la fermeture de l'application, si l'utilisateur ou le développeur n'ont pas explicitement choisi de le sauvegarder dans un fichier sur le disque, autrement dit de rendre la donnée *persistante*. Il faut donc construire des ponts entre l'application et la base de données ; mais le modèle objet (sur lequel se basent les langages de programmation comme Java) et le modèle relationnel (sur lequel se basent les bases de données) n'ont pas de concepts communs, et passer de l'un à l'autre demande une certaine analyse.

Arriver à créer des outils efficaces est primordial pour les années à venir : avec le développement des outils informatiques et la volonté généralisée de conserver des données à diverses fins, le marché des bases de données est en pleine expansion, comme le montre le graphique ci-dessous [26]. Les fonds dévoués aux bases de données vont déjà passer au-dessus des 50 milliards de dollars pour 2017, et ils vont continuer à augmenter avec le fameux problème du Big Data, ou comment traiter les milliards de données qui vont devoir être enregistrées durant les prochaines années.



Marché global des bases de données (en milliards de $), de [26]

Comment peut-on contourner cet obstacle structurel sans pour autant alourdir le programme ou allonger le temps d'exécution ? Ce mémoire de recherche a pour objectif d'expliquer, dans un premier temps, les notions importantes concernant les modèles objet et relationnel. Dans un second temps, il présente ensuite les solutions existantes et les nouveaux problèmes qui se posent, à l'aide d'articles de recherches récents.

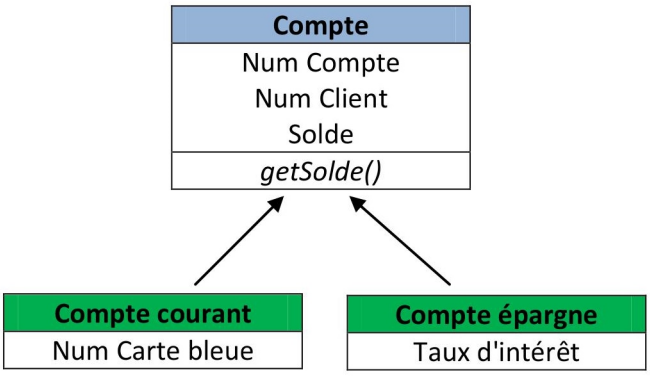
2 Définitions et présentation du problème

2.1 Le modèle orienté-objet

Vers 1990, un langage changea profondément la programmation informatique : le Java. Ce langage vérifie le paradigme de la "programmation orientée objet" : il permet de créer des objets, qui sont des structures de données valuées et privées qui répondent à un ensemble de messages.

Les principes de la programmation objet sont les suivants :

- On construit des classes (p.ex. : Compte) avec des attributs (p.ex. : Num Compte) qui décrivent les objets appartenant à la classe, et des méthodes (p.ex. : *getSolde()*) qui simulent son comportement ou permettent d'agir sur lui.
- Ces classes peuvent êtres hiérarchisées ou non (un Compte Courant est un Compte), être internes d'autres classes, être visibles ou non de l'extérieur ; de nombreuses possibilités sont ouvertes.
- Un élément d'une classe est appelé instance ou objet, il possède les attributs de sa classe et on peut lui appliquer les méthodes de celle-ci.



2.2 Le modèle relationnel

En 1970, IBM crée System/R, le premier prototype de base de données [16]. Depuis, le monde des bases de données relationnelles n'a cessé de se développer, et a probablement atteint l'optimalité au vu des possibilités offertes.

Les principes d'une base de données relationnelle sont les suivants :

- Les données sont stockées dans des relations (des tables) sous forme de n-uplets (des lignes de données). Chaque relation modélise généralement une instance du monde réel.
- Chaque relation a une clé primaire (composée d'un ou plusieurs attributs), dont la valeur identifie les n-uplets, et d'autres attributs décrivant l'instance modélisée (ex : pour un client : son adresse, son téléphone...).
- Certains attributs d'une relation peuvent faire référence à une clé primaire d'une autre relation : ceux-ci sont alors "clés étrangères". (ex : dans la relation Compte, le numéro de client fait référence à la clé primaire de la relation Client ; on peut donc retrouver un client à partir de son numéro).
- On peut faire des opérations sur les relations, par exemple des différences, unions, divisions, jointures, produit cartésien. Les instructions pour modifier et interroger la base de données sont exprimées en SQL (*Structured Query Language*), un langage standard pour les bases de données [3].

Relation Client

Num Client **(Clé primaire)**	Adresse	Téléphone
101	1, place du Cerisier 53254 Prisson	02 70 58 96 87
102	225, route du Pré aux Vaches 33544 Narciac	05 74 12 14 52

Relation Compte

Num Compte **(Clé Primaire)**	Num Client **(Clé étrangère)**	Solde en euros
1452156	101	2432,10
1894628	102	4153,25
45678541	101	3756,59

Dans l'exemple ci-dessus, la base est constituée de deux relations : Client et Compte. Client a pour clé primaire Num Client, et a deux autres attributs : Adresse et Téléphone. Ainsi, dans notre modèle, deux clients peuvent avoir la même adresse et le même téléphone ; pour éviter cela, on peut ajouter ce qu'on appelle une contrainte d'unicité sur l'attribut l'un des deux attributs, ou le couple (deux clients peuvent avoir alors même adresse mais doivent avoir deux numéros différents, ou inversement).

Quant à Compte, elle a pour clé primaire Num Compte, et a deux autres attributs : Num Client et Solde. Num Client est une clé étrangère faisant référence à la clé primaire de Client ; on peut donc retrouver les informations sur le Client associé à un numéro de compte.

2.3 Comparaison objet-relationnel

Au vu de ce descriptif, il apparaît que la programmation objet et les bases de données sont deux mondes différents du point de vue de leur construction, et qu'il est compliqué de les mettre en relation ; c'est la non-correspondance objet-relationnel. Par exemple, comment représenter l'héritage de classes dans une base de données (exemple des comptes courants/d'épargne)? Comment faire pour l'unicité d'attributs d'une relation dans un schéma de classes (Adresse et Téléphone), ou bien lorsqu'on est dans le cas d'une clé étrangère (Num Client) ?

Or, avec notamment le nombre croissant d'applications smartphones en Java qui ont besoin d'enregistrer les objets créés, il est intéressant d'un point de vue pratique de pouvoir fabriquer une base de données à partir de classes pour stocker les objets créés, tout comme inversement de pouvoir créer des classes hiérarchisées à partir d'une base de données. Il existe une solution créée par Sun Microsystems il y a plusieurs années et proposée par Oracle, qu'est le JDBC (Java Database Connectivity), qui permet d'introduire des instructions en SQL dans un programme Java (cf Annexe), mais le développement est généralement long et laborieux – tout est à faire manuellement – et donc coûteux (30% du coût de création d'un programme, [13]). Les recherches se focalisent donc sur l'idée de construire un outil qui permette aux applications de créer et d'accéder rapidement et facilement à des données relationnelles d'un point de vue objet, et inversement. C'est ainsi qu'on est arrivé au *mapping* objet-relationnel (ORM), dont le principal outil est Hibernate. Par définition, "un *mapping* objet-relationnel est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé." [29]

3 Les outils de *mapping* objet-relationnel

3.1 Outils ORM

TopLink (du nom de l'entreprise créatrice qui le créa, The Object People [4]) fut le premier outil ORM à sortir sur le marché, mais il fut vite dépassé par les Enterprise Java Beans (EJB) développés en 1997 par IBM et adoptés par Sun Microsystems deux ans plus tard. Un EJB est une sorte de navette entre l'utilisateur et la base de données (aucun lien direct entre eux), et il en existe deux types principaux : les EJB session, qui ne vivent que le temps d'un échange avec le client, et les EJB entité, qui permettent de représenter et de gérer des données stockées dans une base de données [5]. Le gain de temps pour le développeur se fait à ce niveau-là : alors qu'avec JDBC celui-ci devrait taper toutes les instructions en SQL (cf Annexe), les EJB prennent en charge certains services.

Le grand reproche qui a été fait aux EJB – et notamment aux EJB 2 - est que ceux-ci nécessitent encore beaucoup de codage, ce qui les rend tout de même relativement lourds. De même, la structure-même est complexe. C'est de cette détractation qu'est née l'idée d'Hibernate dans l'esprit de Gavin King au début des années 2000 – c'est ce que son créateur explique lorsqu'il annonce ses débuts le 19 janvier 2002 sur TheServerSide.com [10] : "*I just released hibernate 0.9 on sourceforge. This is based on some code I originaly wrote to do BMP in websphere when I got sick of EJB 1.x CMP limitations (no collections or dependent objects). Its now a great alternative to things like CMP + JDO.*" [Je viens de mettre Hibernate 0.9 en ligne sur sourceforge. C'est basé sur un code que j'ai initialement écrit pour faire des bmp sur websphere quand j'en ai eu marre des limitations EJB 1.x CMP (pas de collections ou d'objets dépendants). C'est maintenant une merveilleuse alternative aux choses comme CMP ou JDP."]

3.2 Hibernate

Hibernate est une révolution dans le monde des ORM, c'est une passerelle efficace entre deux mondes si opposés par définition, comme vu précédemment. D'après son créateur [2], il permet de "libérer le développeur de 95% du travail de persistance objet, comme écrire des instructions compliquées en SQL avec de nombreuses jointures de tables et copier des valeurs de listes de résultats de JDBC à des objets ou des graphes d'objets" ; en effet, Hibernate fait lui-même les requêtes JDBC. D'autre part, le développeur n'a besoin que de connaissances en programmation objet, alors qu'avec JDBC il lui faut en plus connaître le modèle relationnel et le SQL.

Les quatre avantages mis en avant par les auteurs sont :

- la productivité : Hibernate réduit le temps de codage, concentrant les efforts des développeurs sur le vrai fond du programme, ou sur d'autres questions comme la sécurité ou l'efficacité ;
- la simplification : de nombreuses lignes de code utilisant JDBC sont évitées, les requêtes en SQL étant générées automatiquement par Hibernate ;
- la performance : Hibernate optimise les requêtes, alors qu'un code écrit pour JDBC est suivi à la lettre par la machine, sans que celui-ci soit forcément optimal ;
- l'indépendance du vendeur : Hibernate est gratuit, et est applicable sur un certain nombre de systèmes de gestion de bases de données, ce qui le rend particulièrement attractif auprès des entreprises qui souhaitent minimiser leurs coûts.

Pour convertir des classes Java en relations de bases de données, Hibernate se sert de métadonnées, comme les annotations des entités, faites par l'utilisateur au préalable. Les métadonnées sont de l'information ajoutée aux classes lors de la construction de celles-ci, sans que cela ait un impact sur leur rôle technique ou fonctionnel, elles permettent ensuite à un outil ORM comme Hibernate de "mapper" les classes en un schéma de relations convenant aux préférences de l'utilisateur. Le gain de temps se fait ici : on ajoute des métadonnées tout en construisant les classes Java, et Hibernate s'occupe de générer le code SQL permettant de créer les relations dans la base de données et les requêtes de lecture et de mise à jour associées.

L'exemple suivant permet de créer une classe et une relation Client. Le numéro du client est un attribut de la classe, et la clé primaire de la relation (ici, le num_client de cette dernière est généré automatiquement avec l'annotation @GeneratedValue{strategy=GenerationType.AUTO}).

```
@Entity
public class Client{
  @Id
  @GeneratedValue{strategy=GenerationType.AUTO}
  @Column(name="'Num_Client"')
  private int num_client;
...
}
```

Cette utilisation des métadonnées a été reprise et généralisée avec Java Persistance API (JPA) de la norme EJB 3 ; désormais Hibernate implémente cette nouvelle norme JPA, mais a quelques fonctionnalités supplémentaires.

3.3 Positionnement d'Hibernate et des autres outils ORM

Hibernate est devenu un standard dans le domaine de la persistance, et a été adopté par les développeurs : d'après [7], Hibernate est le 5e outil ORM utilisé dans les projets présents sur Github (un site sur lequel on trouve des projets opensource). En revanche, JDBC reste largement utilisé, puisqu'il permet d'exprimer des requêtes bien plus compliquées pour des outils ORM plus poussés (d'après [7]). Toujours, on constate dans le tableau qui suit que Spring est à la deuxième position ; Spring est un projet opensource aidant à construire la structure d'une application Java, et qui utilise JDBC pour accéder à la base de données. L'outil JPA est l'ORM standard pour les Java Platforms (il suit la norme JPA); quant à Vaadin, qui est 4e, c'est un outil Web pour la construction d'applications Internet.

Framework name	URL	Occurs if the project contains at least a file	#projects
JDBC	www.oracle.com/technetwork/java/javase/jdbc	importing java.sql	2,271
Spring	projects.spring.io/spring-framework	importing org.springframework	1,562
JPA	www.tutorialspoint.com/jpa	importing javax.persistence	1,168
Vaadin-GWT	vaadin.com	importing com.google.gwt	361
Hibernate	hibernate.org	whose name ends with .hbm.xml	238

Les cinq outils ORM les plus utilisés sur Github, de [7]

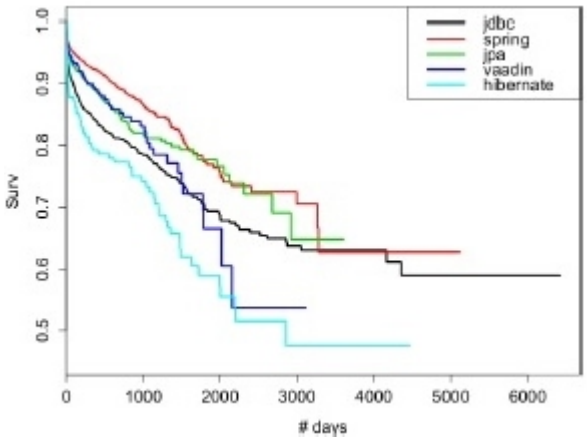
Ce que veut montrer [7], c'est que les outils ORM (opensource) s'utilisent les uns les autres, et que les développeurs se servent fréquemment de plusieurs outils dans leurs projets. Le tableau ci-dessous indique le nombre de co-occurrences relevées parmi les 3 707 projets de Github utilisant les cinq principaux outils ORM (JDBC, Spring, JPA, Vaadin-GWT, Hibernate).

	Spring	JPA	Vaadin	Hibernate
JDBC	645	565	143	192
Spring		558	76	156
JPA			98	105
Vaadin				22

Nombre de co-occurrences des outils ORM sur Github, de [7]

JDBC a tendance à être plutôt utilisé en complément d'autres outils ORM (1545 co-occurrences au total) qu'indépendamment : d'après les auteurs, 80.1% des projets qui utilisent Hibernate utilisent JDBC, 48.4% pour JPA, 41.3% pour Spring, 39.6% pour Vaadin.

Les auteurs se sont aussi intéressés à "l'espérance de vie" des outils ORM : au bout de combien de temps sont-ils supprimés, remplacés par un autre dans les projets considérés ? Le résultat est sans appel : Spring et JPA restent présents dans 80% des projets où ils ont été introduits cinq ans plus tôt. En comparaison, JDBC est à 70% et Hibernate à 60%. De manière générale, Spring est l'outil ORM qui statistiquement reste le plus longtemps dans un projet donné.



Courbes de survie des différents outils ORM dans les projets considérés, de [7]

L'explication qu'on peut donner à ce phénomène est l'évolution des technologies : Hibernate évolue peut-être moins vite que Spring ou JPA, et les développeurs préfèrent alors se tourner vers d'autres outils ORM ; d'autre part, Spring est utilisé pour construire la structure des applications, il peut donc être naturel, pour simplifier le développement, de se servir aussi de l'outil ORM de Spring lorsque les fonctionnalités principales (construction des applications) sont utilisées.

3.4 Les limites actuelles d'Hibernate

Par ailleurs, plusieurs recherches se sont intéressées au temps d'exécution des outils ORM, point important quand on sait combien de données à la fois vont être manipulées ces prochaines années. Un utilisateur n'aimera pas attendre un quart d'heure pour obtenir une réponse à sa requête, il faut donc optimiser chaque action.

En 2014, Shahram Ghandeharizadeh et Ankit Mutha, chercheurs à la Université of Southern California de Los Angeles, ont publié un article montrant certaines limites d'Hibernate, dans un article intitulé : *An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions*, [6].

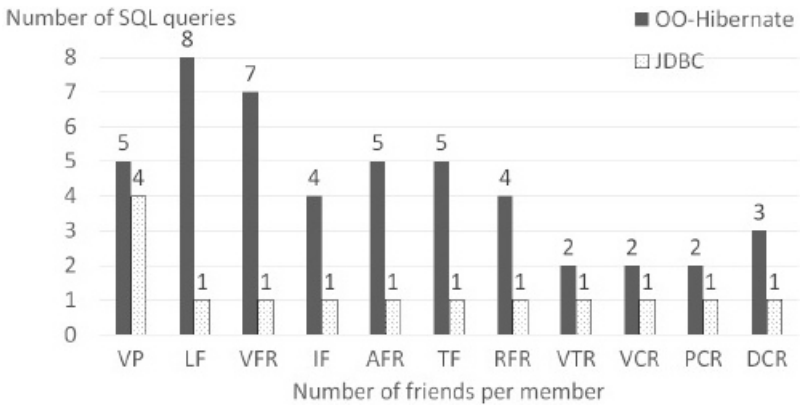
Le but de l'article est "d'évaluer les performances d'Hibernate en le comparant avec une implémentation JDBC d'origine, en utilisant une référence dénommée BG". BG est une sorte de réseau social simplifié [1], dont la base de données est constituée d'un nombre fixé de membres avec un profil enregistré, tout comme de relations représentant les invitations, amitiés, ou données sur les utilisateurs ; ces relations comportent des cardinalités de *mapping* différentes (one-to-many[1] et many-to-many[2]) On peut consulter cette base de données, tout comme générer une demande d'ajout d'ami, en accepter, en refuser, ou en supprimer une. Son générateur produit, au hasard, un ensemble de tâches, dont chacune des tâches imite une série de membres qui effectuent une action. On a alors un réseau social avec de nombreuses requêtes, qui permet de tester la rapidité de réponse d'Hibernate ou de JDBC.

[1] one-to-many : un compte est lié à un unique client mais un client peut avoir plusieurs comptes
[2] many-to-many : un sportif peut adhérer à plusieurs clubs et un club a plusieurs adhérents

L'expérimentation consiste à mesurer le nombre de requêtes effectuées par Hibernate/JDBC, le temps de réponse moyen par action, tout comme le Taux d'action sociale (Social Action Rating, SoAR) qui est le nombre maximal d'actions qu'on peut faire tout en ayant 95% des requêtes effectuées en moins de 100 ms (le SoAR mesure donc la qualité du service).

Le résultat est sans appel : d'après le graphique ci-après, Hibernate utilisé tel quel génère largement plus de requêtes que JDBC. JDBC ne demande qu'une ou deux requêtes selon le type d'action, alors qu'Hibernate en effectue entre deux et huit. Le graphique ci-dessous ne considère que 5 amis par membre et 5 demandes en amis par membre ; pour bien plus d'amis, le manque d'optimalité d'Hibernate serait dramatique.

Il est toutefois possible de se ramener à une optimalité proche de JDBC à l'aide de requêtes en HQL (*Hibernate Query Language*), langage permettant d'exprimer des requêtes manuellement avec Hibernate.



Nombre de requêtes SQL effectuées par Hibernate et JDBC pour chaque action de BG, en considérant 5 amis par membre et 5 demandes en ami par membre, de [6].

Il n'y a pas que l'outil ORM qui entre en jeu dans le temps d'exécution : le langage utilisé pour les requêtes est lui aussi un facteur. Il existe actuellement deux grands langages pour écrire des requêtes dans Hibernate, à savoir HQL et Criteria API. Le premier a une syntaxe proche de celle du SQL, mais il utilise des noms de classes et d'attributs au lieu de noms de relations et de colonnes ; quant au second, il a une syntaxe orientée objet, plus proche du Java.

Requête en HQL permettant d'obtenir tous les objets d'une relation Client :

```
Query query = session.createQuery
("from Client client")
```

La même requête en Criteria API :

```
Criteria cr = session.CreateCriteria
("Client.class");
```

Dans leur article *The Hibernate Framework Query Mechanisms Comparison*, [27], Tisinee Surapunt et Chartchai Doungsa-Ard étudient le temps d'exécution d'un grand nombre de requêtes aléatoires en fonction du langage choisi pour ces requêtes (HQL ou Criteria API). On génère deux échantillons de données dans la base : l'un de 30 000 données, l'autre de 100 000

données. Ensuite, on effectue les mêmes requêtes sur un nombre variable de données de ces échantillons, et on compare le temps mis en fonction du langage dans lequel sont exprimées les requêtes. Dans un premier temps, ces requêtes sont sans condition ; dans un second temps, elles possèdent une condition WHERE (c'est-à-dire qu'on sélectionne des éléments dont un attribut en particulier vérifie une certaine condition fixée).

On obtient que les temps d'exécution sont à peu près semblables dans tous les cas où la requête est sans condition. Dans le cas de la condition WHERE, les requêtes rédigées en Criteria API sont exécutées significativement plus vite lorsqu'elles s'appliquent à un grand nombre de données (plus d'une dizaine de secondes).

Query Mechanisms	Command without WHERE condition				Command with WHERE condition			
	Small queried records set (100 – 1,000 records)		Large queried records set (1,000 – 30,000 records)		Small queried records set (100 – 1,000 records)		Large queried records set (1,000 – 30,000 records)	
	30,000 records in the database	100,000 records in the database	30,000 records in the database	100,000 records in the database	30,000 records in the database	100,000 records in the database	30,000 records in the database	100,000 records in the database
HQL Query	x	x - only 500-700 queried records set	x	x	x	x – only 700 and 1,000 queried records set		x - only 1,000 queried records set
Criteria API		x	x - only 1,000-3,000 queried records set	x - only 1,000-4,000 queried records set	x – only 700-800 queried records set	x	xx	xx

x means the optimal mechanism with each environment of queried data and database.
xx means the percentage of time difference value is significant. Thus, the results show the appropriate query approach obviously.

Langage à utiliser en fonction de la requête et du nombre de données, [27]

Ces deux articles de recherche nous montrent combien l'efficacité d'un outil ORM dépend de nombreux facteurs. Actuellement, les développeurs (et chercheurs) ont tendance à raisonner au cas par cas, et à ne pas forcément toujours utiliser le même outil ORM mais à en combiner plusieurs ou bien à écrire des lignes de code à la main qui permettront d'accélérer les requêtes. C'est cette tendance qui a amené deux chercheurs du département de mathématiques et d'informatique de l'Open University à considérer que la recherche actuelle sur les problèmes de persistance objet/relationnel s'affairait à chercher des solutions aux symptômes des non-correspondances plutôt qu'aux causes elles-même. L'article *Exposing the Myth : Object-Relational Impedance Mismatch is a Wicked Problem*, [11] de Christopher Ireland et David Bowers (2015), a pour but de recadrer les recherches dans le domaine et à reconsidérer le problème autrement.

Les auteurs de [11] font référence à un article de 1973, *Dilemmas in a General Theory of Planning*, [24], qui traite a priori de sociologie. Mais dans cet article, Rittel et Webber ont montré qu'il existait des problèmes "résistants", c'est-à-dire non résolubles de manière linéaire : ils sont trop difficiles et complexes pour qu'on leur trouve une résolution exacte algorithmique. Comme le définissent Rittel et Webber, "les solutions des problèmes résistants ne sont pas de la forme vrai-faux, mais bon-mauvais", et les solutions sont donc multiples. Ce que font remarquer Ireland et Bowers, c'est que la non-correspondance objet/relationnel serait un problème résistant, et donc encore loin d'être résolu.

Pour rechercher les causes de cette non-correspondance, le couple de chercheurs a classifié les symptômes dans des catégories qu'ils dénomment "types de problèmes" (cf tableau ci-dessous) : la structure, l'instance, l'encapsulation, l'identité, le modèle de traitement, et l'appartenance du schéma.

Type de problème	Détail
Structure	Les problèmes de structure comprennent toute différence de structure de données entre le schéma d'un programme orienté-objet et le schéma d'une base de données relationnelle.
Instance	L'essence du problème d'instance est de savoir où se trouve la copie canonique des données.
Encapsulation	Alors qu'on peut protéger l'accès aux données dans un modèle orienté-objet, cela est impossible dans le cas d'une base de données relationnelles. Comment interdire la modification d'une donnée ?
Identité	Comment identifier de manière unique une collection de données, et ce à la fois dans le modèle orienté objet et dans le modèle relationnel ?
Modèle du processus	Il s'agit de l'efficacité technique de l'outil ORM.
Appartenance du schéma	Le problème se pose lorsque le projet est élaboré par plusieurs équipes : l'équipe qui construit le modèle orienté-objet peut faire des choix différents de ceux faits par l'équipe qui gère le modèle relationnel.

Différents types de problèmes de non-correspondance, d'après [11]

Ils relient ensuite les différents types entre eux, en se demandant à chaque fois si un problème d'un certain type peut avoir des conséquences sur un autre type. Ainsi, ces chercheurs proposent qu'au lieu de continuer les recherches sur la manière de dépasser diverses non-correspondances, il faudrait considérer des problèmes plus généraux qui englobent et classent les non-correspondances, catégories qui ont été listées dans le tableau précédent. Résoudre les causes et non les symptômes.

Parallèlement, une année plus tôt, Torres, Galante et Pimental publiaient *ENORM : An Essential Notation for Object-Relational Mapping* [28], dans lequel ils proposent une nouvelle manière de représenter les classes et associations, tout en faisant un lien entre la représentation objet et le schéma de la base de données. Ils présentent leur nouvelle représentation comme une représentation UML étendue : "ENORM, une notation à usage général qui représente les concepts structurels essentiels de l'ORM en étendant le modèle des classes UML avec un profil, et offrant un ensemble concis de nouveaux éléments visuels spécifiques aux designs ORM". Autrement dit : créer une nouvelle manière de représenter le *mapping* objet-relationnel. Celle-ci pourrait aider à traiter le problème de Structure énoncé dans le précédent tableau.

4 Conclusion

Dans la première partie de ce mémoire, nous avons constaté que les modèles objet et relationnel sont difficiles à relier, en raison de leurs différences structurelles. Dans la seconde partie, nous avons vu que le *mapping* objet-relationnel est un problème complexe, qui n'est toujours pas résolu [11]. On arrive aujourd'hui à dépasser certains obstacles, mais on n'arrive pas à créer d'outil ORM construisant automatiquement un pont efficace sur tous les points entre les deux mondes, que sont l'objet et le relationnel. Selon les fonctionnalités recherchées, les outils ORM mettent plus ou moins de temps, comme l'ont noté [6] [27]. De même, comme l'ont constaté les auteurs de [7], les outils ORM s'utilisent les uns les autres pour tenter d'optimiser leurs requêtes, mais ne risque-t-on pas de créer une boucle infinie ?

Ainsi, plusieurs questions restent sans réponse. Est-il possible d'inventer l'outil ORM parfait ? Faudrait-il créer une nouvelle manière de représenter tout en un les objets et les bases de données en informatique, quitte à abandonner toutes les avancées faites dans ces deux domaines depuis plus de vingt ans ? Il est certain que plusieurs recherches se feront à ce sujet ces prochaines années. Si ce n'est pas un des fameux problèmes du millénaire, on peut tout de même le ranger dans les problèmes du siècle.

5 Annexe

Il existe depuis les débuts de Java (version 1.1) une solution pour stocker dans une base de données relationnelles des objets construits en Java : Java DataBase Connectivity (JDBC). C'est un ensemble de classes dont les applications peuvent se servir pour agir sur des SGBD, qui permettent d'entrer des requêtes en SQL dans un code en Java. L'utilisateur n'a pas d'accès direct à la base de données, c'est JDBC qui est l'intermédiaire entre la base et l'application. L'exemple ci-dessous permet d'illustrer la manière dont on se sert de JDBC, ici pour créer une classe "Etudiant". Il n'a pas été jugé pertinent de développer tout le code, ainsi nous ne présentons ici que les parties principales et intéressantes des instructions JDBC.

La première étape consiste à initialiser les variables de connection.

```
Connection          db=null;
Statement           sql=null;
DatabaseMetaData dbmd;
```

Statement va contenir les requêtes en SQL, les deux autres éléments sont relatifs à la connection avec la base de données.

Après avoir défini argv comme une liste contenant les informations sur l'utilisateur et sur le chemin du fichier, on connecte l'application au SGBD :

```
String username = argv[0];
String password = argv[1];
String fichierProp = argv[2];
db = ConfigConnection.getConnection(fichierProp,username,password);

sql = db.createStatement();
```

sql est un moyen d'exécuter sur la base de données une requête en SQL. On voit son utilité avec l'exemple ci-dessous.

On crée une table Etudiant, avec Etudiant_ID commme clé primaire, et Nom et Prenom sont deux attributs obligatoirement renseignés.

```
String sqlText = "CREATE TABLE Etudiant (Etudiant_ID NOT NULL, " +
        "Nom NOT NULL," +
        "Prenom NOT NULL," +
        "CONSTRAINT PK_Etudiant PRIMARY KEY (Etudiant_ID))"
```

On exécute la commande SQL.

```
sql.executeUpdate(sqlText);
db.commit();
```

On se déconnecte de la base de données.

```
if (sql != null) sql.close();
if (db != null) db.close();
```

References

[1] BARAHMAND Sumita, GHANDEHARIZADEH Shahram : *BG: A Benchmark to Evaluate Interactive Social Networking Actions*, in : Database Laboratory Technical Report, Computer Science Department, USC, June 2012.

[2] BAUER Christian, KING Gavin, GREGORY Gary : *Java Persistence with Hibernate*, Manning Publications, 2015.

[3] CHAMBERLIN Donald D. et BOYCE Raymond F. : *SEQUEL : A structured English query language*, in : Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, p. 249-264, ACM, 1974.

[4] Cunningham & Cunningham, "The Object People", juin 2016 : `http://c2.com/cgi/wiki?TheObjectPeople`

[5] DOUDOUX Jean-Michel, "Développons en Java - les EJB", juin 2016 : `http://www.jmdoudoux.fr/java/dej/chap-ejb.htm`

[6] GHANDEHARIZADEH Shahram et MUTHA Ankit : *An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions*, iiWAS '14, 4-6 December 2014.

[7] GOEMINNE Matthieu et MENS Tom : *Towards a Survival Analysis of Database Framework Usage in Java Projects*, ResearchGate, February 2016.

[8] Hibernate, "Hibernate. Everything data.", juin 2016 : `http://www.hibernate.org`

[9] HOCK-CHUAN Chua, "Java Programming Tutorial ; OOP - Composition, Inheritance & Polymorphism", avril 2016 : `http://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html`

[10] KING Gavin, "A new java persistance tool", janvier 2002 : `http://www.theserverside.com/discussions/thread.tss?thread_id=11367`

[11] IRELAND Christopher, BOWERS David : *Exposing the Myth : Object-Relational Impedance Mismatch is a Wicked Problem*, in : DBKDA 2015 : The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications.

[12] MANOUVRIER Maude : *Polycopié du cours de Bases de données relationnelles*, Université Paris-Dauphine, 2015/2016.

[13] MANOUVRIER Maude : *Transparents du cours de Persistance objet-relationnel/Hibernate*, Université Paris-Dauphine, 2014/2015.

[14] MANOUVRIER Maude : *TP JDBC*, Université Paris-Dauphine, 2014/2015.

[15] MASSON Michel : *Polycopié du cours de Java-Objet*, Université Paris-Dauphine, 2015/2016.

[16] McJONES Paul, "System R", avril 2008 : `http://www.mcjones.org/System_R/`

[17] NAUGHT Aleph, "JavaOne Conference Trip Report - Enterprise JavaBeans Technology: Developing and Deploying Business Applications as Components", juin 2016 : `http://www.alephnaught.com/WorkRelated/JavaOne1998/session6.html`

[18] Oracle, "EJB Deployment Description Reference", octobre 2009 : `https://docs.oracle.com/cd/E13211_01/wle/dd/ddref.htm#1043516`

[19] Oracle, "Java SE Technologies - Database", juin 2016 : http://www.oracle.com/technetwork/java/javase/jdbc/index.html

[20] Oracle, "A Sample Application using Object-Relational Features", décembre 2002 : https://docs.oracle.com/cd/A87860_01/doc/appdev.817/a76976/adobjxmp.htm

[21] Oracle, "Logiciel Java | Oracle France", juin 2016 : https://www.oracle.com/fr/java/index.html

[22] PATRICIO Anthony : *Java Persistence et Hibernate*, Eyrolles, 2007.

[23] PASTRE Dominique : *Logique et Principe de Résolution*, Université René Descartes – Paris 5, UFR de mathématiques et informatique.

[24] RITTEL Horst and WEBBER Mevin : *Dilemmas in a General Theory of Planning*, in Policy Sciences 4, 155-169, Elsevier, 1973.

[25] RUSSEL Craig : *Bridging the Object-Relational*, ACM Queue, May/June 2008.

[26] SHIELDS Anne, "Is Oracle's Position Secure in the Database Space?", janvier 2016 : http://marketrealist.com/2016/01/oracles-position-secure-database-space/

[27] SURAPUNT Tisinee, DOUNGSA-Ard Chartchai : *The Hibernate Framework Query Mechanisms Comparison*, in : Lecture Notes on Software Engineering, Vol. 2, No. 3, August 2014.

[28] TORRES Alexandre, GALANTE Renata, PIMENTA Marcelo : *ENORM: An Essential Notation for Object-Relational Mapping*, SIGMOD Record, Vol. 43 no. 2, June 2014.

[29] Wikipedia, "Mapping objet-relationnel", juin 2016 : https://fr.wikipedia.org/wiki/Mapping_objet-relationnel