Université de Namur
Faculté d'informatique
Année académique 2017–2018

Archipelago : Un framework de persistance pour bases de données orientées graphe.

Gilles Bodart

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
CLEVE Anthony

Co-promoteur : LAMBIOTTE Renaud

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Ce mémoire va se concentrer sur les implémentations de la théorie des graphes dans l'univers des bases de données, et développer un framework Java présentant les qualités suivantes, l'abstraction, la simplification et la configurabilité.

Archipelago est un framework dont le but est d'abstraire l'utilisation d'une base de données orientée graphe. Cet outil dans son état actuel, possède trois méthodes fonctionnelles, à savoir une méthode permettant de persister l'information, une autre permettant de lier deux noeuds à l'aide d'une relation, et une dernière permettant de récupérer les différents objets dans la base de données. Elle peut effectuer ces trois opérations sur `Néo4J` et sur `OrientDB`.

Ce travail présente les points forts et les points faibles de l'implémentation de ce framework ainsi que des pistes d'amélioration pour une version ultérieure.

mots-clés : Framework, Base de données, Java, Graphes, Neo4J, OrienDB, Archipelago

Abstract

Avant-propos

Nous tenons à remercier tout particulièrement mes promoteurs, Messieurs A. CLEVE et R. LAMBIOTTE pour nous avoir autorisé à réaliser cette recherche et conjointement, à nous avoir soutenu dans le développement de celle-ci. Nous les remercions aussi pour le temps consacré à la supervision de ce mémoire.

Nous remercions également les entreprises dans lesquelles nous avons travaillé, de nous avoir laissé l'opportunité de poursuivre le cursus universitaire.

Et enfin, nous tenons tout autant à remercier ma belle-mère C. NIGOT pour avoir pris du temps à nous relire, et singulièrement ma compagne M. VAN CUTSEM qui m'a soutenu au quotidien pendant ces deux années, y compris pour sa patience actuelle durant sa grossesse en cours.

Table des matières

Glossaire

SQL	Structured Query Language
NoSQL	Not only SQL
SGBD	Système de gestion de base de données
SGBDR	Système de gestion de base de données relationnelles
BDOG	Base de donnée orientée graphe
JSON	JavaScript Object Notation
Vertex	Sommet
Edge	bord
Cluster	littéralement grappe, cela représente en informatique un groupe d'éléments
API	Application Programming Interface
REST	Representational state transfer, style d'architecture pour les systèmes hypermédia distribués

Introduction

De nos jours, les unités de stockage sont de plus en plus accessibles au grand public, les différentes entreprises impliquées dans le développement des systèmes de gestion de base de données l'ont bien compris. Depuis plusieurs années, l'hégémonie des bases de données relationnelles se fait de moins en moins écrasante, les solutions alternatives du NoSql séduisent jour après jour de grandes entreprises.

La grande force des SGBDR provient des fondements théoriques mis en place en 1970 par Edgar Frank Codd. IBM définit le langage "Structured Query Language" (SQL) pour utiliser ce système. Cette normalisation de la représentation de l'information génère un sentiment de sûreté pour les architectes de logiciels.

A contrario, le mouvement NoSql pour "Not Only SQL" ne se base pas sur un fondement théorique commun. Que ce soient la théorie des graphes, les systèmes de hash key/value avec partitions, ... chaque système choisit ce sur quoi il va se baser.

Mais s'il existait une sorte de contrat liant les SGBD NoSql entre eux afin d'assurer une cohérence d'utilisation, ce mémoire va se concentrer sur les implémentations de la théorie des graphes, en essayant de mettre en place un système de comparaison. Nous allons tenter de spécifier certains objectifs communs et développer un framework Java permettant d'utiliser de manière agnostique l'une ou l'autre base de données.

En outre, nous aimerions produire un framework qui présente les qualités suivantes :

- L'abstraction
  L'utilisateur pourra employer ce framework sans même connaitre l'existence de librairies spécifiques.
- La simplification
  Les différentes méthodes seront explicites et faciles à intégrer dans une application existante.
- La configurabilité
  Le framework réagira en fonction d'un fichier de configuration clair et simple.

Première partie

État de l'art

Chapitre 1

L'évolution du NoSql

Un SGBD est par définition un ensemble de procédés permettant d'organiser et de stocker des informations (potentiellement de gros volumes). Si stocker et retrouver l'information est un des plus grand challenge d'un SGBD, une communauté de développeur pense que ces systèmes devraient pouvoir offrir d'autres fonctionnalités.

A partir des années 1980, le modèle relationnel supplante les autres formes de structures de données.

Les évolutions logicielles suivant naturellement les évolutions matérielles, la généralisation des interconnexions des réseaux, l'augmentation de la bande passante, la diminution du coût des machines, la miniaturisation des espaces de stockage, ... de nouvelles opportunités sont apparues au XXI^{e} siècle.

Les entreprises comme Google, Amazon, Facebook, Twitter, ... sont tour à tour arrivées aux limites du modèle relationnel. Que ce soit à cause de volumes astronomiques (plus de 100 pétaoctets) ou du nombre de requêtes par seconde, il fallut développer une nouvelle façon de gérer les données.

Le NoSql découle de ce genre de problèmes, ces modèles arrivent avec des approches optimisées pour des secteurs spécifiques.
Comme ces modèles représentent ce qui n'est pas relationnel, par souci de classification, nous allons distinguer 4 usages principaux :

— Performances : L'objectif du SGBD sera d'augmenter au maximum les performances de la manipulation des données.
— Structures simples : Pour s'affranchir de la rigidité du modèle relationnel, la structure sera généralement simplifiée, en utilisant une représentation plus souple comme le JSON par exemple.
— Structures spécifiques : Certains moteurs NoSql sont liés à des besoins spécifiques. La structure de représentation de données sera dès lors focalisée sur un cas d'utilisation.
— Volumétries : Un des principaux aspects importants des SGBD NoSql est leur capacité de gérer la montée en charge de données. La distribution des traitements au travers de plusieurs clusters est un facteur essentiel dans la plupart des applications BigData.

Nous allons aussi distinguer 4 grandes familles de représentation de schéma de données :

— Document : L'utilisation de formats spécifiques tels que le très répandu "JSON" permet de stocker les données sur base de fichier.
— Clé / Valeur : Le système le plus simple, il manipule des paires de clé/valeur, ou accède à un élément en fonction d'une table de hachage.
— Colonne : Inspiré de Google BigTable, la structure ressemble à la table relationnelle. On peut la comparer à une table de hachage qui va référencer une ou plusieurs colonnes.
— Graphe : La famille Graphe se distingue du fait que les entités ne sont pas considérées comme des entités indépendantes, mais que la relation entre ces objets est tout aussi importante que le contenu.

Les implémentations de bases de données de types graphe sont de plus en plus nombreuses, les relations entre les éléments permettent de parcourir le graphe de manière très performante le rendant de plus en plus intéressant pour les entreprises possédant des millions de données. L'utilisation de ce genre de SGBD est dès lors tout à fait recommandée pour des entreprises intéressées entre les relations de ces données, telles que des profils sociaux, des liens de cause à effet, des liens géographiques et bien d'autres encore.[4][15][3]

1.1 Questionnement

Pourquoi utiliser une BDOG plutôt qu'une BD relationnelle comme Oracle ou MySql ?

L'utilisation des SGBD relationnelles pour tout type de problème est révolue. L'apparition de SGBD NoSQL spécifiques, permet dès à présent une approche plus personnalisée. En effet, comme tout système, le relationnel a ses limites. L'approche actuelle des entreprises est plus orientée "Big Data" ; les entreprises veulent tout stocker afin d'avoir le plus d'informations possible. Or comme le relationnel cadenasse les données dans une table préalablement définie, on se doit de tronquer l'information ou de mettre à jour le schéma de définition des tables. Dans une BDOG telle que Neo4J, nous pouvons ajouter toutes les informations que nous voulons sans condition préalable. Cette approche permet davantage un contrôle de l'intégrité des données aux applications.

Si l'objectif de l'applicatif est de représenter un système récursif comme une arborescence de fichier, de la généalogie, ... le modèle relationnel se basera sur une table faisant référence à elle-même, l'utilisateur devra donc réaliser une jointure par profondeur. Si l'arborescence descend jusqu'à 15 niveaux, cela peut devenir problématique. L'approche BDOG permet de parcourir une arborescence sans pour autant charger l'ensemble des nœuds ayant le même label. Dès lors, plus besoin de projection "d'ensemble" pour pouvoir continuer le chemin.

En relationnel, il existe de nombreux moyens d'historiser les données, cependant, lorsque le schéma comprend des relations cycliques, cette opération devient plus complexe. L'utilisation d'un BDOG rend la démarche plus simple, il suffit de créer un nouveau nœud avec les anciennes données, lier ce nœud avec une relation historique en y spécifiant la date de mise à jour, et ensuite de changer les valeurs du nœud référencé par le cycle. Ce procédé peut être mis au point dans n'importe quelle représentation et ne nécessite aucune refonte globale du modèle de données.[17]

On parle de base de données orientée graphe mais quelle est la différence entre une relation entre deux nœuds et une relation entre deux tables ?

Les relations entre deux tables se font à l'aide de clé étrangère définie lors de la création du schéma de données. Elles apportent de l'information supplémentaire en permettant de réaliser des jointures entre plusieurs tables.

La relation dans les BDOG est une information à part entière, elle peut posséder autant de données que le nœud lui-même, la force de ce genre de système est de pouvoir lier deux nœuds distincts par n'importe quelle relation et ce, à tout moment.

Chapitre 2

Backgroud technique

2.1 Neo4J

2.1.1 Description

Créé par Neo Technologie, une société américaine et suédoise, elle est actuellement la base de données orientée graphe, la plus utilisée dans le monde[14]. Développé en java sous licence GPL V3, AGPL ou licence commerciale, Neo4J représente les données sous forme de "Nœuds" et de "Relations". Chacun de ces éléments peuvent contenir une ou plusieurs propriétés. Les propriétés sont des couples clés/valeurs de type simple, comme des chaines de caractères ou des valeurs numériques, des coordonnées spatiales, ... [13]

Une des particularités de Neo4J est l'absence de structure définie, un nœud peut être labellisé afin de permettre de travailler sur un ensemble d'éléments, mais il n'y aura aucune contrainte sur les propriétés du nœud. Cette particularité rend ce SGBD bien adapté pour les modèles évoluant fréquemment.

2.1.2 Langage de requête

Le langage propre à Neo4J se nomme "Cypher", il a pour but de réaliser plus simplement que SQL les opérations de parcours ou d'analyse de proximité.

```
1 CREATE (mamours:Person {name:"Mamours"})
2 CREATE (mamyco:Person {name:"Mamyco"})
3 CREATE (gilles:Person {name:"Gilles"})
4 CREATE (marie:Person {name:"Marie"})
5 CREATE (bebe:Person {name:"Enfant"})
6 CREATE (mamours)-[:PARENT_OF]->(gilles)
7 CREATE (mamyco)-[:PARENT_OF]->(marie)
8 CREATE (marie)-[:PARENT_OF]->(bebe)
9 CREATE (gilles)-[:PARENT_OF]->(bebe)
```

Ces requêtes vont créer 5 nœuds et 4 relations PARENT_OF, nous pouvons aisément comprendre que "Mamyco" est parente de "Marie"

```
MATCH (p:Person)-[:PARENT_OF]->(c:Person)
RETURN DISTINCT (p)
```

Cette query va retourner tous les nœuds distincts qui ont une relation :PARENT_OF avec un autre nœud.

```
MATCH (gp:Person)-[:PARENT_OF*2]->(c:Person)
RETURN DISTINCT (gp)
```

Celle-ci quant à elle va retourner toutes les personnes qui sont "parent de parent" et donc grands-parents.

Ces deux exemples peuvent montrer la force de l'utilisation d'un SGBD de type graphe pour représenter un ensemble hiérarchique de données par rapport aux SGBD relationnels qui nécessiteraient une double jointure sur la Table "Person".

2.1.3 Communication

Neo4J peut être utilisé sous plusieurs formes.

La première option est une solution embarquée, ce choix peut être très intéressant en alternatif au très célèbre SQLite relationnel.

La deuxième, pour toute application distribuée, est une solution autonome pouvant tourner comme un service sur tout type de plateforme. Le protocole "Bolt", développé par "Néo Technologie" est grandement conseillé pour communiquer avec ces serveurs distants.

Son utilisation est simple grâce à l'utilisation de la librairie native à Néo4J pour java (neo4j-java-drive) ou avec l'utilisation de l'API Rest déployée en même temps que le SGBD.

2.2 OrientDB

2.2.1 Description

OrientDB est un SGBD initialement développé en C++ (Orient ODBMS), repris ensuite en 2010 en Java par Luca Garulli dans une version multi-modèle sous licence Apache 2.0, GPL et AGPL. Actuellement 3ème mondial (selon db-engines.com) il offre de nombreuses fonctionnalités intéressantes.[12]

OrientDB est une base de données associant Document et Graphe. Elle combine la rapidité et la flexibilité du type document ainsi que les fonctionnalités de relations des bases de données graphe.

Ce SGBD est composé de trois grands éléments

— Document & Vertex : Source de contenu, ces éléments peuvent être considérés comme des containers de données. On peut le comparer avec la ligne d'une base de données

relationnelles.
— Links & Edge : Une arête orientée reliant deux éléments non nécessairement distincts.
— Property : Typée ou embarquée dans un document JSON, ceci va représenter le contenu de l'information. Ces propriétés sont bien entendu primordiales pour ordonner, rechercher, ...

Chaque Document ou Vertex appartient à une "Class", celle-ci peut être strictement définie ou plus laxiste. Comme dans la programmation orientée objet, OrientDB offre le principe de polymorphisme avec un système d'héritage entre les classes.

OrientDB vient dans sa version community avec un système de clustering permettant à l'utilisateur de gérer correctement les montées en charge. Chaque document est identifié avec une partie désignant le cluster dans lequel l'information est stockée et une autre partie désignant sa position dans ce dernier (exemple @rid : 10 :12). Chaque classe peut être associée à un ou plusieurs clusters, permettant d'optimiser les accès dans des ensembles plus petits.

2.2.2 Langage de requête

OrientDB utilise une sorte de SQL avancée pour interpréter les requêtes. On peut de plus utiliser le langage Gremlin.

Voici quelques exemples d'utilisation du SQL avancé dans OrientDB.

```
1 CREATE CLASS Person EXTENDS V
2 CREATE CLASS Company EXTENDS V
3 CREATE CLASS WorkAt EXTENDS E
4 CREATE PROPERTY Person.firstname string
5 CREATE PROPERTY Person.lastname string
6 CREATE PROPERTY Company.name string
7 INSERT INTO Person(firstname, lastname) VALUES
      ("Gilles","Bodart"),("Marie","Van Cutsem")
8         ou
9 INSERT INTO Company set name = "ACME"
```

Cet ensemble de requêtes ressemblant au langage SQL permet de créer deux vertex, Person et Company, un Edge WorkAt et leur associe certaines propriétés. Les deux types d'insert différents permettent comme en SQL d'ajouter un nœud.

```
1 SELECT FROM V
```

Metadata			Properties		
@rid	@version	@class	firstname	lastname	name
10 :0	1	Person	Marie	Van Cutsem	
10 :1	1	Person	Gilles	Bodart	
11 :0	1	Company			ACME

Nous pouvons observer qu'OrientDB se charge de qualifier les documents de plusieurs métadata en leur octroyant par exemple un identifiant unique (n° cluster :position), un numéro de version pour permettre une gestion de transaction "full optimistic" [1] et le class représente la structure

1. On laisse l'utilisateur continuer sa transaction et le refus se fera lors du "commit" s'il y a eu une modification

du document.

```
1 CREATE Edge WorkAt from 10:1 to 11:0
```

Cette petite requête va lier le document ayant l'id 10 :1 au document dont l'id est 11 :0 par une relation WorkAt. Nous pouvons traduire en français que Marie Van Cutsem travaille chez ACME.[16]

2.2.3 Communication

OrientDB embarque une API Rest complète[7], toute action pouvant être faite sur les interfaces web et consoles, peut être reproduite au travers de cette API.

concurrente du même document

Chapitre 3

Solutions existantes

3.1 Les librairies

TODO livre

3.1.1 Hibernate

Hibernate, soutenu et développé par JBoss, a mis au point un système de relation entre le code Java et le SGBD Néo4J, en réutilisant les mêmes annotations de relations que celles employées pour les bases de données relationnelles, à savoir :

- @OneToOne : Relation 1-1
- @ManyToOne : Relation n-1
- @OneToMany : Relation 1-n
- @ManyToMany : Relation n-n

Bien qu'à l'heure actuelle, aucune intégration avec OrientDB, un ticket est ouvert (OGM-855) depuis 2 ans auprès d'Hibernate.

3.1.2 Spring Data

Spring data à nouveau possède un module de communication avec le SGBD Néo4J, il intègre au célèbre Framework Spring, les fonctionnalités de Néo4J-OGM [1], permettant de qualifier une classe avec certaines annotations facilitant la sérialisation et la dé sérialisation vers la représentation graphique.

Exemple :

```
1 @NodeEntity(label="Film")
2 public class Movie {
3
```

1. Object Graph Mapper

```
    @GraphId Long id;

    @Property(name="title")
    private String name;
}

@NodeEntity
public class Actor extends DomainObject {

    @GraphId
    private Long id;

    @Property(name="name")
    private String fullName;

    @Relationship(type="ACTED_IN", direction=Relationship.OUTGOING)
    private List<Role> filmography;

}

@RelationshipEntity(type="ACTED_IN")
public class Role {
    @GraphId
    private Long relationshipId;
    @Property
    private String title;
    @StartNode
    private Actor actor;
    @EndNode
    private Movie movie;
}
```

3.1.3 Librairie Neo4J

Une librairie java développée par NeoTechnologie existe, elle permet de travailler aisément avec une base de données Neo4J embarquée ou à distance. Elle est en constante évolution et se trouve sur une plateforme open source.

Exemple de communication avec la librairie Neo4J le protocol "Bolt" :

```
Driver driver = GraphDatabase.driver("bolt://localhost:7687",
    AuthTokens.basic("matrix", "neo"));
Session session = driver.session();
session.run("CREATE (a:Person {firstName:{name},lastName:{lastName}})",
    parameters("firstName","Gilles","lastName","Bodart"));
session.close();
driver.close();
```

Bien que cette libraire soit efficace, elle ne peut être utilisée avec un autre SGBD.

3.1.4 Librairie OrientDB

La librairie OrientDB pour java est composée de trois éléments :

- `graph API`
- `document API`
- `object API`

La `graph API` d'OrientDB couvre 80% des uses cases classiques d'un utilisateur, elle supporte tous les modèles de représentations comme un unique multi-modèle de données. Elle travaille sur des "Vertex" et des "Edges", respectivement nœuds et relations, et est compatible avec le standard TinkerPop

La `document API` permet de couvrir les 20% restants, il est plus simple d'utilisation mais offre une utilisation plus atomique, les relations ne peuvent être par exemple qu'unidirectionnelles tandis que la bidirectionnalité est offerte dans l'API précédent. Un document API est un élément unique pour gérer les relations entre les différents nœuds de notre modèle, il faudra gérer manuellement chaque connexion.

La dernière librairie se focalise sur les objets java, elle n'a plus été améliorée depuis la version 1.5 d'OrientDB [2]

3.2 Les frameworks

3.2.1 Apache TinkerPop

Apache TinkerPop est un Framework open source d'utilisation de graphe, il regroupe un grand ensemble de fonctionnalités et d'algorithmes. Son écosystème est de plus en plus agrémenté de librairies externes développées par de tierces personnes.

Il se considère lui-même comme étant d'utilisation complexe pour les nouveaux développeurs car il nécessite l'utilisation d'un environnement qui lui est propre, la "Gremlin-console".

Les développeurs de ce framework ont mis au point un langage de travail sur les graphes appelé "Gremlin". Le langage dont nous avons parlé en section 2.1.2 s'est grandement inspiré de ce dernier.[8]

Exemple d'utilisation

```
gremlin> g.V() /* récupère tous les nœuds du graphe */
==>v[1]
==>v[2]
==>v[3]
==>v[4]
==>v[5]
==>v[6]
```

2. À l'heure de l'écriture de ce mémoire, nous sommes à la version 2.2 et la version 3 a bientôt terminé sa phase de "bêta test"

```
gremlin> g.V(1) /* récupère le nœud d'Id 1 du graphe*/
==>v[1]
gremlin> g.V(1).values('name') /* récupère le nœud d'Id 1 du graphe et
    renvoie sa propriété 'name' */
==>marko
gremlin> g.V(1).outE('knows') /* récupère les relations 'knows' partant du
    noeud d'Id 1 */
==>e[7][1-knows->2]
==>e[8][1-knows->4]
gremlin> g.V(1).outE('knows').inV().values('name') /* retourne le nom des
    personnes que le nœud 1 connait */
==>vadas
==>josh
gremlin> g.V(1).out('knows').values('name') /* même résultat */
==>vadas
==>josh
gremlin> g.V(1).out('knows').has('age', gt(30)).values('name') /* retourne le
    nom des personnes âgées de plus de 30 ans que le nœud 1 connait */
==>josh
```

Comme expliqué plus haut, le langage doit être exécuté dans une invite de commande ou dans un serveur Gremlin, nous sommes dès lors contraints d'installer et de configurer ce dernier pour pointer vers l'un ou l'autre SGBD.

Deuxième partie

Contribution

Chapitre 4

Applications possibles des BDOG

4.1 L'exemple parfait

Pour la suite de ce mémoire, il est primordial d'avoir un exemple concret de modèle habituellement persisté avec un modèle relationnel mais qui pourrait néanmoins être transposable dans une BDOG.

L'école

Imaginons la représentation d'une école composée d'élèves, de professeurs, de classes, de salles techniques, de travailleurs, ... dont le diagramme de classes se trouve en Annexe (8.1).

Les liens entre les différents objets peuvent être très nombreux, nous pouvons imaginer que les étudiants sont amis, membres de famille, ennemis, amoureux, ... il en est de même pour les professeurs, ils peuvent avoir les mêmes relations que les étudiants mais en outre ils vont être assignés à des cours, à des classes, ... toutes ces liaisons peuvent être fortement compliquées à représenter sur une base de donnée relationnelles. L'utilisation d'une BDOG dans ce cas, semble tout à fait adéquate. Durant la suite de ce mémoire, nous utiliserons donc ce modèle.

4.2 Critères de comparaison

Dans un premier temps, il est primordial de mettre en place une liste de critères permettant de faciliter le choix de la BDOG.

4.3 Comparaison des plus grandes BDOG

BDOG / Critère	Neo4J	OrientDB
Schéma de donnée strict	X	✓
Format de données	JSON	JSON
Principe d'héritage	X	✓
Clusturing	Configurable	Natif
Communication	BOLT (protocole propriétaire)	REST (via HTTP)
Évolutivité	L'absence de schéma strict entraîne une grande liberté dans la création et la modification des nœuds	Les classes doivent préalablement être configurées sur le serveur
Rapidité [1]	7m 10s	4m 31s
Modèle de représentation	Uniquement graphe	multimodèle

Une idée d'arbre de décision devrait, à la fin de cette section, permettre à un utilisateur muni de ses besoins, de choisir la BDOG la plus adaptée à son projet.

4.4 Piste de normalisation

Comme nous l'avons présenté dans le point 1 nous nous apercevons que le problème principal du NoSql est le manque de standard d'utilisation de ces SGBD. Pour ce faire, nous allons tenter de lister un des objectifs de normalisation.

- Utilisation d'un langage unique
- Système de transaction
- Opération de base de persistance CRUD
- Possibilité de créer des méthodes stockées
- Contrôle sur les propriétés.

La suite de ce mémoire portant sur le développement d'un framework java, tentera d'offrir ces objectifs de manière abstraite aux développeurs.

1. 10 000 requêtes sur le même graphe de données avec la même requête au travers d'un appel Java

Chapitre 5

Le framework

5.1 Utilisation

5.1.1 La configuration

Archipelago permet de se lier à Néo4J ou à OrientDB sans changer le code source de l'application. Cependant, une légère configuration est nécessaire pour initialiser les paramètres d'exécution.

Le fichier de configuration doit se nommer "config.archipelago.json" et être présent dans les ressources du projet.

Voici le canevas de ce fichier :

```
{
  "database": {
    "type": "", // Choix entre NEO4J et ORIENT_DB
    "username": "",
    "password": "",
    "url": "",
    "name": "", // Nom de la base de donnée
    "embedded": false, // boolean Optionnel
    "port": 1234 //
  },
  "deepness": 3, // Profondeur de persistance voir ci-après
  "domainRootPackage": "org.archipelago.test.domain.school"
  // package où se trouvent les classes du domaine
}
```

Pour la suite, ce framework a pricipalement été basé sur l'utilisation de l'annotation *@Bridge*.

5.1.2 La persistance

La persistance est l'élément principal de ce framework. Afin de ne pas être tenu par une syntaxe appartenant à une BDOG propre, nous nous sommes basés sur une approche de représentation

orientée objet.

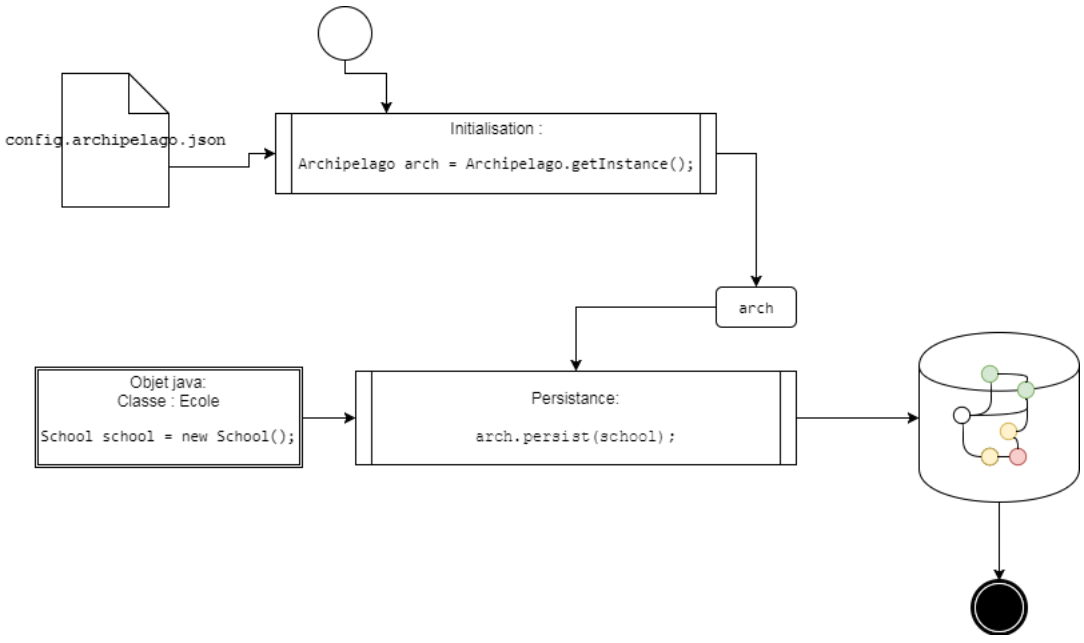
La persistance de ces objets ainsi que des éléments qui leur sont liés dépend en partie du fichier de configuration qui vient d'être décrit dans la section précédente.

Afin d'assurer la persistance des éléments liés, nous avons opté pour une approche récursive en "depth-first" avec marquage de sommet. En effet, pour créer un lien entre deux éléments, il faut absolument que ceux-si soient présents dans la base de données.

La propriété "deepness" est primordiale pour la performance de cet outil, elle permet de définir la profondeur de persistance d'un objet. Ce procédé empêche de tomber dans des boucles infinies, problème courant lorsque nous travaillons avec de modèles ayant des relations bidirectionnelles comme suit :

```
1 Archipelago arch = Archipelago.getInstance();
2
3 Student gilles = new Student();
4 Student antoine = new Student();
5
6 gilles.setFriend(antoine);
7 antoine.setFriend(gilles);
8
9 arch.persist(gilles);
```

Ainsi avec la valeur sentinelle contenue dans la propriété "deepness", nous rompons la boucle de persistance après avoir atteint la profondeur souhaitée.



Pour ce qui est de la persistance du lien entre deux objets, nous avons fait le choix de créer des relations unidirectionnelles. Afin de créer un lien, il faut apposer l'annotation *@Bridge* sur la propriété d'où il part.

Nous avons cependant ajouté la possibilité d'obtenir une relation réciproque en associant la valeur `true` dans l'attribut `biDirectionnal` de l'annotation.

Exemple avec la Classe *Student.java*

```java
public class Student extends Person {

    @Bridge(descriptor = "Follow")
    private List<Lesson> lessons = new ArrayList<>();

    @Bridge(descriptor = "FriendOf")
    private List<Student> friends = new ArrayList<>();

    @Bridge(descriptor = "FamilyMember", biDirectionnal = true)
    private List<? extends Person> familyMember = new ArrayList<>();

    @Bridge(descriptor = "PromotedIn")
    private Promotion prom;

    ...
```

Le `descriptor` qui est obligatoire, se doit d'être unique, il va représenter dans le BDOG le type de lien entre deux nœuds.

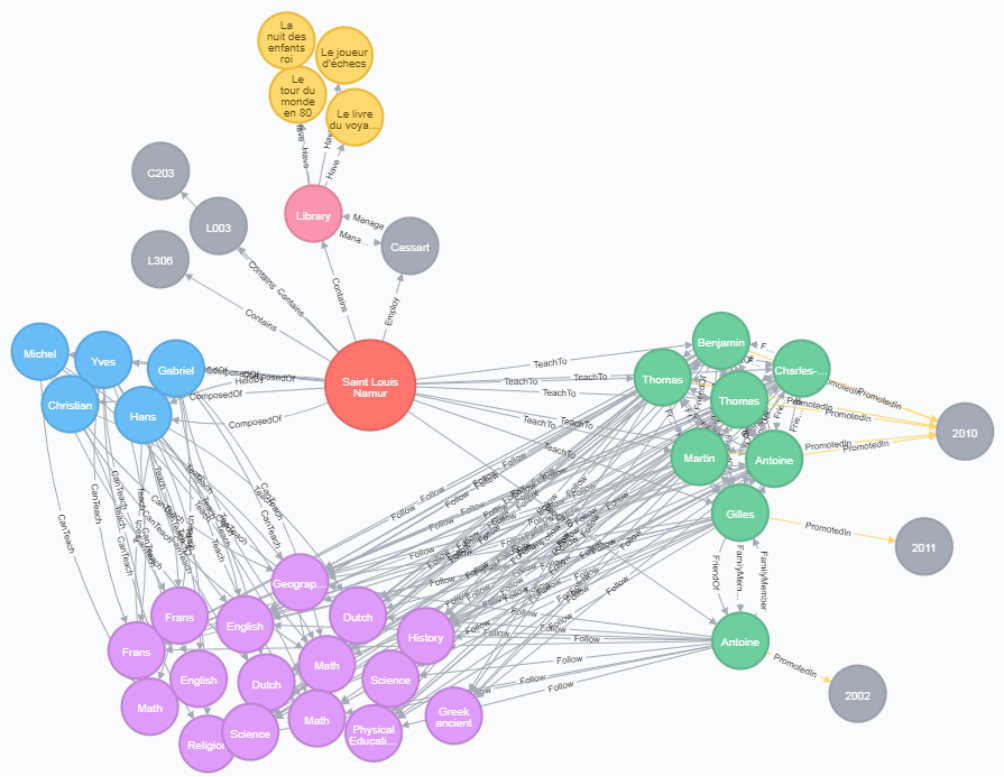
La beauté derrière cette utilisation est qu'au moindre changement de modèle, comme un ajout de propriété, l'ensemble de la couche de persistance sera adaptée à ce changement sans devoir exécuter le moindre script (à l'exception de la création d'une nouvelle classe pour OrientDB).

Si nous prenons l'exemple de l'école (méthode en annexe III) nous pouvons réaliser qu'en un seul graphe, nous visualisons l'ensemble des relations, qu'elles soient uni ou bidirectionnelles.

```
Archipelago arch = Archipelago.getInstance();

School school = genSchool()

arch.persist(gilles);
```



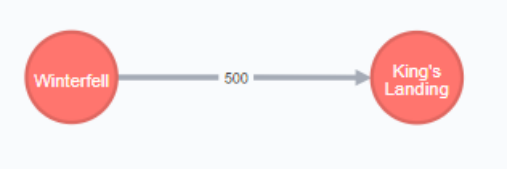
5.1.3 La liaison

Tout graphe se doit de pouvoir lier deux nœuds entre eux, Archipelago offre donc la possibilité de spécifier une liaison entre deux objets en utilisant un troisième.

Voici un exemple :

```
1
2  Archipelago arch = Archipelago.getInstance();
3
4  City winterfell = new City("Winterfell");
5  City kingsLanding = new City("King's Landing");
6
7  Road road= new Road();
8  road.setDistance(5001);
9  road.setUnit("leagues");
10
11 arch.link(winterfell, kingsLanding, road);
```

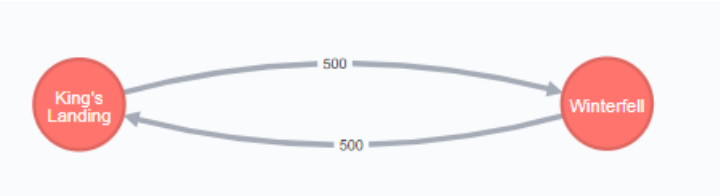
Comme nous pouvons aisément le comprendre, nous allons lier ces deux villes par une relation "Road" contenant les propriétés "distance" et "unité".



Toutefois si nous le spécifions à l'aide de la valeur "true" , le quatrième attribut de la méthode

nous permet de générer automatiquement une relation réciproque ou bidirectionnelle.

```
...
arch.link(winterfell, kingsLanding, road, true);
...
```



5.1.4 La récupération

La récupération de l'information est la raison pour laquelle nous la stockons. A quoi bon enregistrer une valeur dans une base de données si nous ne souhaitons jamais l'utiliser ultérieurement ?

Pour cela, Archipelago offre un système de création de requête. Grâce à ce procédé, que votre BDOG soit OrientDB ou Neo4J, vous serez en mesure de récupérer votre objet sans changer votre code source.

Voici un exemple :

```
Archipelago arch = Archipelago.getInstance();

ArchipelagoQuery aq = a.getQueryBuilder()
                .of(Student.class)
                .build();
List<Object> nodes = arch.execute(aq);
```

Cette requête va rechercher l'ensemble des étudiants de la base de données ainsi que les objets qui leur sont liés.

Un système de condition est également mis en place, il permet d'affiner le résultat obtenu :

```
Archipelago arch = Archipelago.getInstance();

ArchipelagoQuery aq = a.getQueryBuilder()
                .of(Student.class)
                .where(of("firstName","Gilles"), ConditionQualifier.Equal)
                .build();
List<Object> nodes = arch.execute(aq);
```

Cette requête va quant à elle rechercher les étudiants ayant "Gilles" comme prénom.

Nous pouvons aussi ajouter des opérateurs logiques "OU" et "ET" comme suit :

```
Archipelago arch = Archipelago.getInstance();

ArchipelagoQuery aq = a.getQueryBuilder()
                .of(Student.class)
```

```
                .where(of("lastName", "Bodart"), ConditionQualifier.EQUAL)
                .and(of("firstName", "Gilles"), ConditionQualifier.EQUAL)
                .or(of("firstName", "Thomas"), ConditionQualifier.EQUAL)
                .build();
List<Object> nodes = arch.execute(aq);
```

Le procédé mis en place à l'heure actuelle ne permet pas de créer des conditions complexes car il va ajouter chaque élément de condition à la suite de la requête courante.

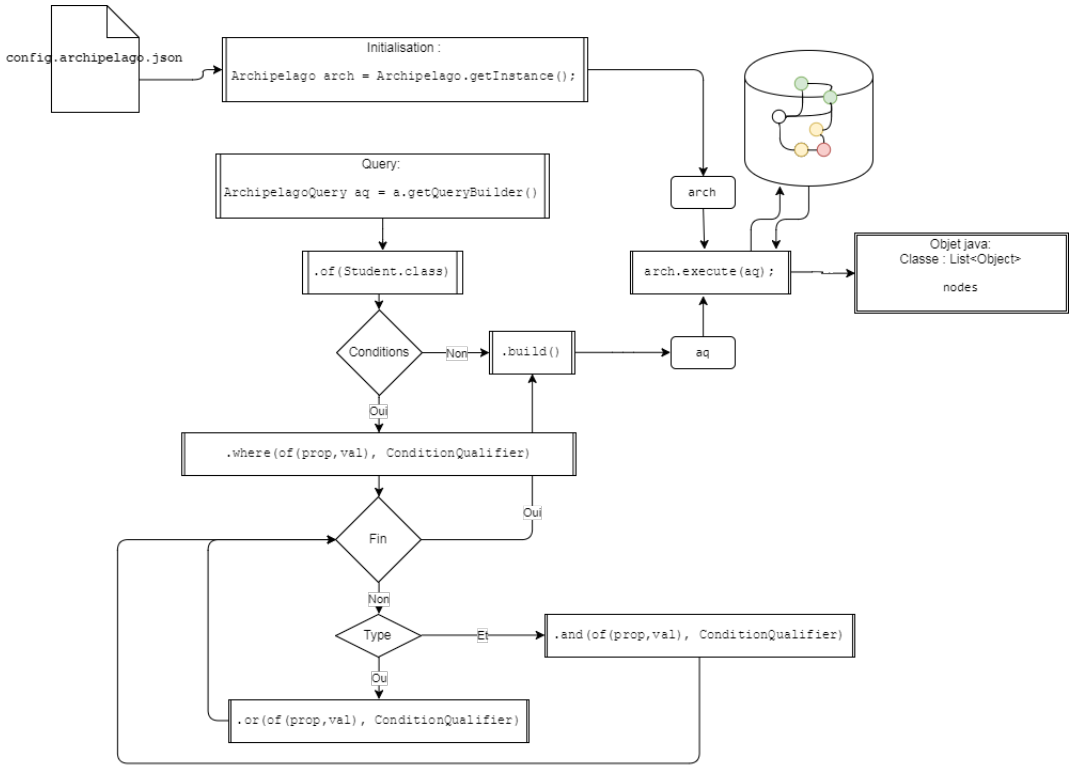
Pour l'exemple précédent, nous aurons donc :

```
(lastName = "Bodart" AND ( firstName = "Gilles" OR firstName = "Thomas"))
```

Comme expliqué plus haut, nous sommes actuellement dans l'incapacité de générer des requêtes comme suit :

```
(lastName = "Bodart" AND firstName = "Gilles")
 OR
(lastName = "Blondiau" AND firstName = "Thomas")
```

Dans notre troisième exemple, le système fournira donc la liste des étudiants ayant "Bodart" comme nom de famille et "Gilles" ou "Thomas" comme prénom. À L'heure actuelle, il n'est cependant pas possible d'effectuer des requêtes sur les relations entre les objets. Cette piste sera primordiale pour une deuxième version de ce framework.



5.2 Schéma conceptuel

La démarche suivie dans le cadre de ce travail est la suivante :

Abstraire la relation entre le code et la base de données, et accéder plus simplement à l'utilisation des BDOG.

Nous avons tenté de nous représenter le graphe dans l'univers de la programmation et nous nous sommes aperçu qu'il existait une grande divergence et une grande complexité entre les implémentations spécifiques des BDOG.

Nous nous sommes dès lors questionné sur la façon de s'éloigner de ces implémentations afin de ne plus être dépendants d'une représentation spécifique et de simplifier toute la procédure. Nous avons dès lors créé un ensemble d'éléments abstraits fournissant une interface entre des les différentes librairies préexistantes.

Archipelago est un framework qui permet d'utiliser ces librairies spécifiques des BDOG en reprenant sur base d'une configuration, les éléments nécessaires, et en simplifiant considérablement toutes les procédures de recherche et de création de requête.

Voici la démarche poursuivie :

5.2.1 Archipelago

L'objet principal du framework Archipelago est d'être une boite à outils comprenant les trois opérations décrites [1] :

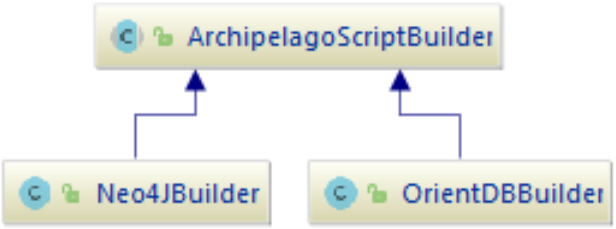
— La persistance
— La liaison
— La recherche

Grâce au fichier `config.archipelago.json`, l'environnement va se moduler automatiquement de manière à pouvoir exécuter le code spécifique adéquat.

Pour ces opérations, nous allons avoir besoin d'utilitaires nous permettant de générer des requêtes de persistance, de liaison et de recherche, pour chaque BDOG. Ces différentes implémentations doivent respecter les obligations décrites par les classes `ArchipelagoScriptBuilder` ainsi que `ArchipelagoQueryBuilder`.

Nous pensons que l'on pourrait fusionner ces deux classes. Néanmoins, nous avons opté pour le choix de les scinder afin de garder une différenciation entre les requêtes qui créent de l'information et celles qui en récupèrent.

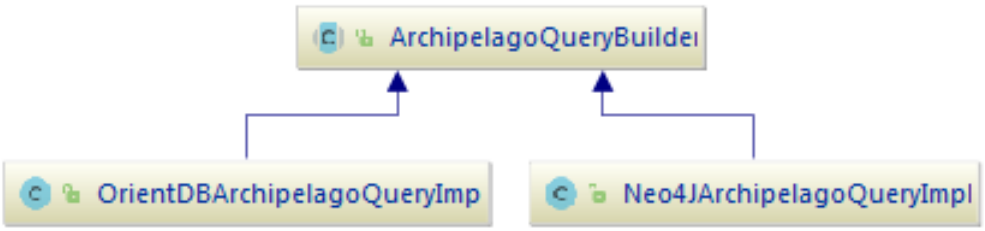
ScriptBuilders



Chaque classe doit correspondre à une base de données et implémenter trois méthodes [2] [3] :

Nom de méthode	Description
`makeCreate`	Cette méthode doit générer une requête permettant de créer un nœud.
`makeMatch`	Cette méthode doit générer une requête permettant de récupérer l'identifiant correspondant à un objet.
`makeRelation`	Cette méthode doit générer une requête permettant de lier deux nœuds.

1. voir 5.1
2. Plus un petit nombre d'overloading à implémenter
3. Chacune de ces requêtes doit être écrite dans le langage ciblé par la classe.



Comme précédemment, chaque classe doit correspondre à une base de données, la liste des méthodes à implémenter est légèrement plus importante.

En voici le détail :

Nom de méthode	**Description**
`init`	Initialisation de la requête de recherche.
`of`	Spécification de l'objet de la recherche.
`where`	Initialisation de conditions permettant de filtrer le résultat final.
`and`	Ajout d'une sous-condition liée avec un `ET` logique.
`or`	Ajout d'une sous-condition liée avec un `OU` logique.
`build`	Finalise la requête de recherche de nœuds et génère une `ArchipelagoQuery`.

Chapitre 6

Evaluation

Ce mémoire a été réalisé dans le but de développer un framework permettant d'abstraire les librairies spécifiques aux bases de données orientées graphe. Cependant, l'état actuel d'avancement de cet outil ne peut que le qualifier de "POC" [1] todo livre.

6.1 Points forts

6.1.1 La facilité

Grâce à Archipelago et sa méthode de persistance d'un objet java, il est aisé d'enregistrer des informations pouvant être fortement inter-connectées dans des bases de données orientées graphe. Le marquage des sommets lors de cette opérationTODO livre nous permet aussi d'améliorer les performances du procédé.

6.1.2 La configurabilité

Nous avons choisi de prendre un fichier JSON comme fichier de configuration car ce format est simple d'utilisation.

À l'heure actuelle, nous disposons d'un certain nombre de paramètres nous permettant d'orienter le framework vers l'une ou l'autre base de données et d'aider la récupération depuis ces dernières. De surcroît, le modèle mis en place permet aisément d'ajouter ou d'ôter de nouveaux paramètres en fonction des améliorations futures.

6.1.3 L'abstraction

L'utilisateur ne doit se soucier d'aucune mise en œuvre spécifique à la base de données ciblée.

Archipelago, grâce à ces implémentations concrètes cachées derrière les classes abstraites `ArchipelagoScrip`

1. Proof of concept - Terme donné à un projet dont le but est de démontrer la faisabilité d'une approche d'architecture, d'une technologie ou d'une solution

ainsi que `ArchipelagoQueryBuilder` permet à l'exploiteur de se concentrer sur la partie algorithmique de son application.

6.1.4 La simplification

La manière dont nous avons imaginé ce framework permet de réutiliser n'importe quel code existant.Il suffit alors à l'utilisateur d'ajouter les annotations `@Bridge` pour que son code soit compatible avec l'application de ce framework.

Il pourra ainsi profiter de bases de données NoSql orientées graphes, au lieu de s'en tenir aux bases de données relationnelles [2] ; les frameworks de communication avec ces dernières étant plus courantes et mieux connues dans l'univers de la programmation.

6.1.5 La modularité

Chaque utilisation de librairie spécifique de communication vers une base de données extérieure se trouve dans une classe séparée.

Ce genre d'architecture offre une maintenance aisée car elle permet de concentrer la recherche de problèmes dans un espace défini et bien cadré. De plus, si deux équipes sont affectées simultanément à ce framework, elles peuvent y ajouter de nouvelles destinations sans se gêner, telles que par exemple ArangoDB[10] ou encore JanusGraph[2].

6.2 Points faibles

6.2.1 Les performances

Bien que nous ayons essayé d'améliorer ce point grâce au marquage des sommets, nous n'avons pas eu le temps d'approfondir ce procédé avec un marquage des arêtes. Dans notre exemple de l'école, la partie reprenant les étudiants est fortement inter-connectée. En conséquence nous devons passer à de nombreuses reprises sur les mêmes arêtes du graphe pour vérifier si cette relation existe dans la base de données au lieu de retenir l'information et d'éviter la démarche. Ces nombreux passages ralentissent le processus [3].

6.2.2 Les conditions

Notre framework possède trois faiblesses principales en ce qui concerne la manière avec laquelle nous pouvons récupérer de l'information.

2. voir 1.1
3. Et plus particulièrement en ce qui concerne OrientDB

Sous-conditions

Comme nous l'avons évoqué dans le point 5.1.4, nous pouvons actuellement générer des conditions simples. Pour créer des conditions plus complexes, il faudrait effectuer une refonte du système d'imbrication afin permettre à l'utilisateur de créer des sous-conditions.

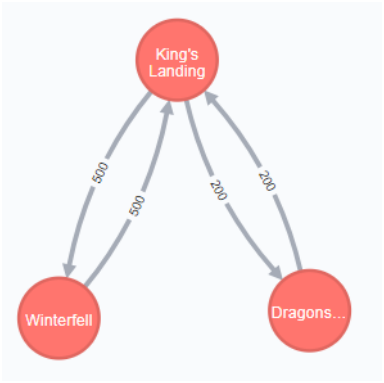
Exemple de condition impossible à générer actuellement :

```
(lastName = "Bodart" AND firstName = "Gilles")
 OR
(lastName = "Blondiau" AND firstName = "Thomas")
```

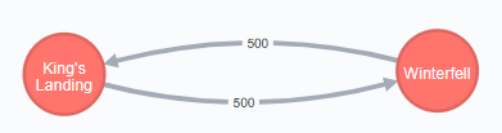
Conditions sur les relations

De plus il n'est pas possible de faire une condition sur une relation entre deux nœuds.

Par exemple nous pourrions avoir grâce à la requête Cypher suivante :




```
MATCH (d:City)-[rel:Road]->(a:City)
WHERE
           rel.distance > 300
       AND rel.unit = "leagues"
RETURN (a)
// Quelles sont les villes éloignées de plus de 300 lieues ?
```



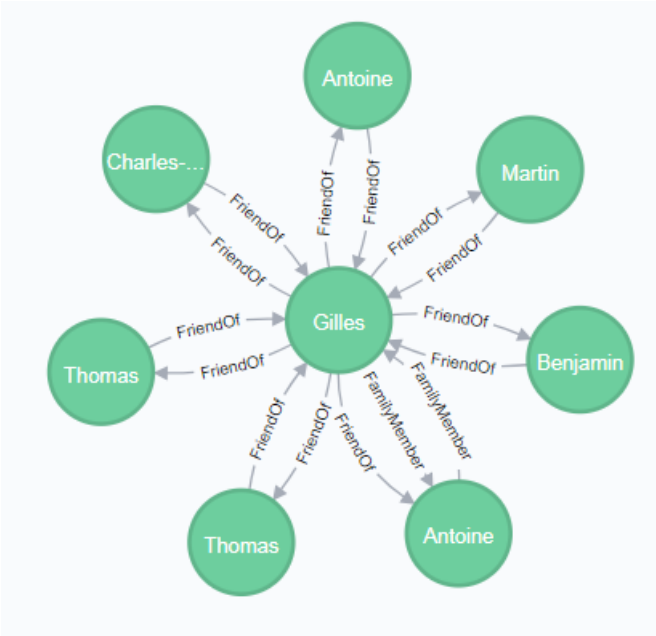
Les requêtes générées par Archipelago nous permettent donc de récupérer l'ensemble des villes liées par une route, mais nous ne pourrons pas nous limiter à celles situées à plus de 300 lieues.

Conditions sur les éléments liés

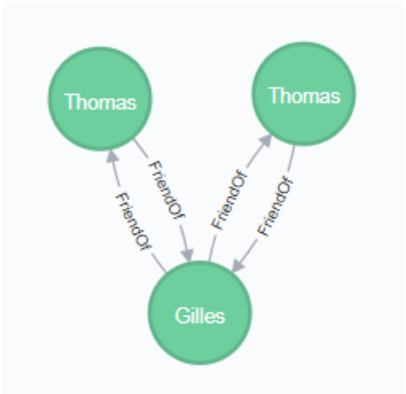
Actuellement l'ensemble des conditions s'appliquent sur le nœud principal de la requête.

À cause de cela, il nous est impossible de filtrer les éléments qui vont être liés avec celui-ci.

Voici un exemple en Cypher :




```
MATCH (me:Student)-[:FriendOf]->(f:Student)
WHERE
            me.firstName = "Gilles"
        AND me.lastName = "Bodart"
        AND f.firstName = "Thomas"
RETURN (me)-[]-(f)
// lie-moi sont mes amis qui se prénomment Thomas?
```



Archipelago à nouveau, ne permet pas de filtrer les amis liés au nœud principal. Du fait de ce manque de possibilité, nous sommes exposés à des problèmes de charge, dès lors que le nœud principal est lié à plusieurs millions d'autres.

6.2.3 La récupération d'informations

Dans l'état actuel d'avancement de ce framework, nous sommes en mesure d'enregistrer des informations sur les relations entre deux nœuds. Cependant il n'y a pas encore de procédé permettant de récupérer ces dernières dans un objet Java. Les liaisons internes dans un objet seront récupérées mais nous ne bénéficierons pas de toutes les informations fournies lors de la persistance.

```java
public class City {

    private String name;

    @Bridge(descriptor = "Road")
    private List<City> connected = new ArrayList<>();

        ...
```

```java
Archipelago arch = Archipelago.getInstance();
City winterfell = new City("Winterfell");
City kingsLanding = new City("King's Landing");
City dragonStone = new City("Dragonstone");

Road road= new Road();
road.setDistance(5001);
road.setUnit("leagues");

Road road2= new Road();
road2.setDistance(2001);
road2.setUnit("leagues");

a.persist(winterfell);
a.persist(kingsLanding);
a.persist(dragonStone);
a.link(kingsLanding,winterfell,road, true);
a.link(kingsLanding,dragonStone,road2, true);

ArchipelagoQuery aq = a.getQueryBuilder()
                .of(City.class)
                .where(of("name","King's Landing"), ConditionQualifier.EQUAL)
                .build();

List<Object> nodes = arch.execute(aq);
```

Nous récupèrerons le nœud correspondant à la ville "King's Landing" et nous aurons dans la propriété `connected` deux autres villes, à savoir "Dragonstone" et "Winterfell". Cependant, nous perdons dans l'application, les informations de distance et d'unité présentes dans les relations `Road`.

6.2.4 La propreté du code

Cette implémentation de ce framework pourrait être améliorée si l'on disposait de temps supplémentaire pour en évacuer les morceaux de codes inutiles. En outre, il serait primordial d'y insérer des tests unitaires afin de s'assurer du bon fonctionnement de chacune des méthodes implémentées.

Le code est actuellement disponible sur un "repository github" [4], de façon à pouvoir récupérer [5] un ensemble de commentaires constructifs par la communauté Java.

4. https ://github.com/GillesBodart/Archipelago
5. Nous n'en avons pas encore reçu.

Chapitre 7

Conclusion

Archipelago est un framework dont le but est d'abstraire l'utilisation d'une BDOG orientée graphe.

Le but de ce travail est de permettre à l'utilisateur de se centrer sur la représentation du modèle ainsi qu'au traitement métier de celui-ci, sans perdre du temps à implémenter la couche de persistance. L'état actuel du développement de ce framework peut être considéré comme un "POC".

Certaines fonctionnalités ont d'ores et déjà été implémentées telles que la persistance, la liaison et la récupération.

Comme signalé ci-dessus, nous pouvons reprendre les points forts de ce framework à savoir :

- **La facilité**
  La facilité d'enregistrement d'informations, pouvant être fortement inter-connectées dans des BDOG, ainsi que le marquage des sommets.
- **La configurabilité**
  Une aide à la récupération d'un certain nombre de paramètres depuis un fichier de configuration permet de changer le comportement du framework ainsi que d'en ajouter ou d'en ôter aisément de nouveaux, en fonction des améliorations futures.
- **L'abstraction**
  L'utilisateur ne doit se soucier d'aucune mise en œuvre spécifique à la base de données ciblée.
- **La simplification**
  Ce framework permet de réutiliser n'importe quel code existant. Il suffit alors à l'utilisateur d'ajouter les annotations `@Bridge` pour que son code soit compatible avec son application.
- **La modularité**
  Le genre d'architecture utilisée pour développer ce framework, offre une maintenance aisée car elle permet de concentrer la recherche de problèmes dans un espace défini et bien cadré.

Les points faibles quant à eux sont :

- **Les performances**

Bien que nous ayons essayé d'améliorer ce point grâce au marquage des sommets, nous n'avons pas eu le temps d'approfondir ce procédé avec un marquage des arêtes.

— **Les conditions**

Notre framework possède trois faiblesses principales en ce qui concerne la manière avec laquelle nous pouvons récupérer de l'information.

- Sous-conditions : nous pouvons actuellement générer des conditions simples. Pour créer des conditions plus complexes, il faudrait effectuer une refonte du système d'imbrication afin permettre à l'utilisateur de créer des sous-conditions.
- Conditions sur les relations : l'impossibilité de produire une condition sur une relation entre deux nœuds.
- Conditions sur les éléments liés : Actuellement l'ensemble des conditions s'appliquent sur le nœud principal de la requête , il nous est impossible de filtrer les éléments qui vont être liés avec celui-ci. Nous sommes dès lors exposés à des problèmes de charge si le nœud principal est lié à plusieurs millions d'autres.

— **La récupération d'informations**

Il n'existe pas encore de procédé permettant de récupérer des informations sur une relation entre deux nœuds, dans un objet Java. Les liaisons internes dans un objet seront récupérées mais nous ne bénéficierons pas de toutes les informations fournies lors de la persistance.

— **La propreté du code**

L'implémentation de ce framework pourrait être améliorée si l'on disposait de temps supplémentaire pour en évacuer les morceaux de codes inutiles. En outre, il serait primordial d'y insérer des tests unitaires afin de s'assurer du bon fonctionnement de chacune des méthodes implémentées.

L'ensemble de ces points faible à été pris en considération durant le développement de ce framework. Nous vous proposons dès lors des pistes d'améliorations réalistes telles que :

1. Amélioration du système de conditions
2. Amélioration des performances
3. Amélioration de l'exploitation des relations
4. Ajout d'algorithmes célèbres de théorie des graphes

Et nous les développerons dans le chapitre 8 de ce travail.

Si l'approche d'une mise en place d'un standard commun aux BDOG semblait intéressante en soi, elle irait apparemment à l'encontre même de ces systèmes. Même si Gremlin et OpenCypher sont en voie de s'imposer vu leur importance dans l'univers des BDOG, la plupart des leurs promoteurs souhaitent les considérer comme des bases de données spécialisées.

La solution proposée par ce mémoire permet quant à elle, d'abstraire les différentes implémentations standardisées ou non, et constituerait une approche unifiée pour l'univers des développeurs Java.

Chapitre 8

Perspectives

8.1 Amélioration du système de conditions

Pour améliorer le système de conditions, il est primordial de refondre la manière dont les conditions sont implémentées. Pour l'avenir de ce projet, il faut que l'utilisateur soit en mesure d'émettre des conditions sur les relations ainsi que sur les éléments liés.

Sur base du questionnaire que nous avons soumis, une idée de modification est apparue dans les suggestions et nous a particulièrement convaincu.

Le système pourrait être revu de la manière suivante :

```
Archipelago arch = Archipelago.getInstance();

ArchipelagoQuery aq = arch.getQueryBuilder()
    .of(City.class)
    .where(and(
              or(
                  eq("name","King's Landing"),
                  eq("name","Winterfell")
                ),
              neq("name","Dragonstone")
              ),
           ArchipelagoPosition.START
          )
    .where(and(
                gt("distance",300),
                eq("unit","leagues")
               ),
           ArchipelagoPosition.EDGE
          )
    .where(eq("name","Winterfell"),
           ArchipelagoPosition.END
          )
    .build();
List<Object> nodes = arch.execute(aq);
```

Avec ce procédé, chaque condition peut être imbriquée dans une autre sous-condition. Nous avons dès lors grâce à cette implémentation, la possibilité d'effectuer toutes les conditions logiques possibles sur le nœud de départ, sur l'arête ainsi que sur le nœud d'arrivée.

8.2 Amélioration des performances

Le premier effort, afin d'améliorer les performances, serait d'effectuer le marquage des arêtes.

Les algorithmes existant de la théorie des graphes, nous permettraient d'améliorer le parcours du graphe généré par les relations entre les différents objets du modèle.

théorie des graphes livre et algo

8.3 Amélioration de l'exploitation des relations

Un objet générique représentant un lien entre deux objets peut être une piste d'amélioration de l'exploitation des relations. Comme la librairie de Néo4J le fait, la récupération d'information s'effectue entre autres dans un objet contenant 3 propriétés intéressantes :

`start`	L'objet de départ.
`relation`	L'objet comprenant les informations de relation.
`end`	Le nœud d'arrivée.

Grâce à cette représentation, nous serons en mesure d'effectuer un grand nombre d'instructions sur ces relations essentielles dans l'utilisation d'une base de données orientée graphe.

8.4 Ajout d'algorithmes célèbres de théorie des graphes

Plus court chemin

L'utilisation des BDOG est entre autres un choix pris pour effectuer des recherches de proximité entre deux éléments de la base de données. Ces algorithmes sont souvent déjà implémentés du côté des BDOG mêmes. Il faudrait dès lors qu'Archipelago implémente une nouvelle fonction permettant d'appeler ces algorithmes tels que le très célèbre "Dijkstra"livre sur dijkstra.

Exemple :

```
1 Archipelago arch = Archipelago.getInstance();
2
3 ArchipelagoQuery aq = arch.getQueryBuilder()
4     .from(City.class)
5         .where(and(
6               or(
7                   eq("name","King's Landing"),
8                   eq("name","Winterfell")
9                ),
```

```
            neq("name","Dragonstone")
            ),
          ArchipelagoPosition.START
         )
    .to(City.class)
        .where(eq("name","Winterfell"),
          ArchipelagoPosition.END
         )
    .with(ArchipelagoAlgo.DIJKSTRA)
    .build();
List<Object> nodes = arch.execute(aq);
```

Coloration des nœuds

La coloration des nœuds est un procédé mathématique permettant de scinder un graphe en plusieurs groupes. TODO à compléter

Problème du voyageur de commerce

Le problème du voyageur de commerce est un problème d'optimisation visant à lier un ensemble de nœuds en formant le plus court chemin entre ceux-ci.

Une implémentation d'une heuristique telle que par exemple l'ACS[6] peut être une fonctionnalité intéressante pour une seconde version de ce framework. En effet, si l'utilisateur à la possibilité d'employer ce genre d'algorithme pour résoudre ce type de problème, Archipelago répondra plus facilement aux attentes d'applications logistiques.

bd orienté réseaux[9]

Bibliographie

[1] Rohit Aggarwal. Networkx to neo4j's documentation. https://neonx.readthedocs.io/en/latest/, 2013.

[2] JanusGraph Authors. Janusgraph official website. http://janusgraph.org/, 2017.

[3] Rudi Bruchez. Les bases de données nosql et le big data. Eyrolles,, 2016.

[4] traduit par Simon Baslé Écrit par Peter Neubauer. Les bases orientées graphes, nosql et neo4j. https://www.infoq.com/fr/articles/graph-nosql-neo4j, 20 mai 2010.

[5] NetworkX developers. Software for complex networks. https://networkx.github.io/, 2014-2017.

[6] Marco Dorigo and Luca Maria Gambardella. Ant colony system : a cooperative learning approach to the traveling salesman problem. IEEE Transactions on evolutionary computation, 1(1) :53–66, 1997.

[7] Roy T Fielding and Richard N Taylor. Architectural styles and the design of network-based software architectures. University of California, Irvine Doctoral dissertation, 2000.

[8] The Apache Software Foundation. Apache tinkerpop official website. http://tinkerpop.apache.org/, 2017.

[9] Georges Gardarin. Base de données. Éd. Eyrolles, 2003.

[10] ArangoDB GmbH. Arrangdb official website. https://www.arangodb.com/, 2017.

[11] Jure Leskovec. Stanford large network dataset collection. https://snap.stanford.edu/data/, 2017.

[12] OrientDB LTD. Orientdb official website. https://orientdb.com/, 2017.

[13] Inc Neo Technology. Neo4j official website. https://neo4j.com/, 2017.

[14] solid IT gmbh. Knowledge base of relational and nosql database management systems. https://db-engines.com, 2017.

[15] Mohamed Tarroumi. Nosql (not only sql). https://prezi.com/4flswlgipwbo/nosql-not-only-sql/, 10 Décembre 2014.

[16] OrientDB Team. Orientdb - getting started. https://www.udemy.com/orientdb-getting-started, 2017.

[17] Directeur France et Europe du Sud de MongoDB Yann Aubry. Pourquoi nosql s'impose face aux sgbdr traditionnelles (avis d'expert). http://www.silicon.fr/base-donnees-nosql-impose-sgbdr-93305.html, 19 mars 2014.

Troisième partie

Annexes

1. DIAGRAMME DE CLASSE "L'ÉCOLE"

2. CRÉATION DES OBJETS COMPOSANT L'ÉCOLE

```
1  private static School genSchool() throws ClassNotFoundException, IOException {
2
3      Lesson math8 = new Lesson("Math", 8l);
4      Lesson math6 = new Lesson("Math", 6l);
5      Lesson math4 = new Lesson("Math", 4l);
6      Lesson science6 = new Lesson("Science", 6l);
7      Lesson science3 = new Lesson("Science", 3l);
8      Lesson frans5 = new Lesson("Frans", 5l);
9      Lesson frans6 = new Lesson("Frans", 6l);
10     Lesson dutch2 = new Lesson("Dutch", 2l);
11     Lesson dutch4 = new Lesson("Dutch", 4l);
12     Lesson english2 = new Lesson("English", 2l);
13     Lesson english4 = new Lesson("English", 4l);
14     Lesson history = new Lesson("History", 2l);
15     Lesson geography = new Lesson("Geography", 2l);
16     Lesson religion = new Lesson("Religion", 2l);
17     Lesson pE = new Lesson("Physical Education", 2l);
18     Lesson greek = new Lesson("Greek ancient", 4l);
19
20     Promotion p2011 = new Promotion(2011);
21     Promotion p2002 = new Promotion(2002);
22     Promotion p2010 = new Promotion(2010);
23
24     Teacher gys = new Teacher("Hans", "Gys", null, "M",
           Lists.newArrayList(dutch2, dutch4, english2, english4),
           Lists.newArrayList(dutch2, dutch4,english2, english4), "Master");
25     Teacher goffin = new Teacher("Michel", "Goffin", null, "M",
           Lists.newArrayList(math8, math6, math4), Lists.newArrayList(math8,
           math6, math4), "Master");
26     Teacher massart = new Teacher("Gabriel", "Massart", null, "M",
           Lists.newArrayList(math8, math6, math4), Lists.newArrayList(math8,
           math6, math4), "Master");
27     Teacher gouthers = new Teacher("Yves", "Gouthers", null, "M",
           Lists.newArrayList(frans5, frans6, religion),
           Lists.newArrayList(frans5, frans6, religion), "Master");
28     Teacher jacques = new Teacher("Christian", "Jacques", null, "M",
           Lists.newArrayList(geography), Lists.newArrayList(geography),
           "Master");
29
30     Student gilles = new Student("Gilles", "Bodart", LocalDate.of(1992, 4,
           14), "M", Lists.newArrayList(math8, science6, dutch2, english4,
           history, geography, religion, frans5, pE), null, null, p2011);
31     Student thomasB = new Student("Thomas", "Blondiau", LocalDate.of(1992, 1,
           5), "M", Lists.newArrayList(math8, science3, dutch2, english4,
           history, geography, religion, frans5, pE, greek), null, null, p2010);
```

```
32      Student thomasR = new Student("Thomas", "Reynders", LocalDate.of(1992, 1,
            22), "M", Lists.newArrayList(math8, science6, dutch2, english4,
            history, geography, religion, frans5, pE), null, null, p2010);
33      Student charly = new Student("Charles-Antoine", "Van Beers",
            LocalDate.of(1992, 4, 28), "M", Lists.newArrayList(math8, science3,
            dutch2, english4, history, geography, religion, frans5, pE, greek),
            null, null, p2010);
34      Student antoine = new Student("Antoine", "Dumont", LocalDate.of(1992, 12,
            28), "M", Lists.newArrayList(math6, science3, dutch4, english4,
            history, geography, religion, frans5, pE, greek), null, null, p2010);
35      Student martin = new Student("Martin", "Périlleux", LocalDate.of(1992, 2,
            28), "M", Lists.newArrayList(math6, science3, dutch4, english4,
            history, geography, religion, frans5, pE, greek), null, null, p2010);
36      Student benjamin = new Student("Benjamin", "Leroy", LocalDate.of(1992,
            10, 31), "M", Lists.newArrayList(math8, science3, dutch2, english4,
            history, geography, religion, frans5, pE, greek), null, null, p2010);
37      Student antoineBo = new Student("Antoine", "Bodart", LocalDate.of(1985,
            10, 18), "M", Lists.newArrayList(math6, science6, dutch2, english4,
            history, geography, religion, frans5, pE), null, null, p2002);
38      Worker cassart = new Worker("", "Cassart", null, "M", null, null);
39      List<Room> rooms;
40      Room library = new org.archipelago.test.domain.school.Library("Library",
            Lists.newArrayList("Le livre du voyage", "Le tour du monde en 80
            jours", "La nuit des enfants roi", "Le joueur d'échecs"), cassart);
41      cassart.setInChargeOf(Lists.newArrayList(library));
42      Room l003 = new ClassRoom("L003", 30, true, false, true);
43      Room c203 = new ClassRoom("C203", 30, true, false, true);
44      Room l306 = new ClassRoom("L306", 50, true, true, false);
45
46      gilles.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,
            martin, benjamin, antoineBo));
47      gilles.setFamilyMember(Lists.newArrayList(antoineBo));
48      thomasB.setFriends(Lists.newArrayList(gilles, thomasR, charly, antoine,
            martin, benjamin));
49      thomasR.setFriends(Lists.newArrayList(thomasB, gilles, charly, antoine,
            martin, benjamin));
50      charly.setFriends(Lists.newArrayList(thomasB, thomasR, gilles, antoine,
            martin, benjamin));
51      antoine.setFriends(Lists.newArrayList(thomasB, thomasR, charly, gilles,
            martin, benjamin));
52      martin.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,
            gilles, benjamin));
53      benjamin.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,
            martin, gilles));
54
55      School school = new School();
56      school.setName("Saint Louis Namur");
57      school.setDirector(gys);
58      school.setTeachers(Lists.newArrayList(goffin, gouthers, jacques, gys,
            massart));
```

```
59      school.setStudents(Lists.newArrayList(gilles, thomasB, thomasR, charly,
            antoine, martin, benjamin, antoineBo));
60      school.setRooms(Lists.newArrayList(library, l003, l306, c203));
61      school.setWorkers(Lists.newArrayList(cassart));
62      return school;
63
64      }
```

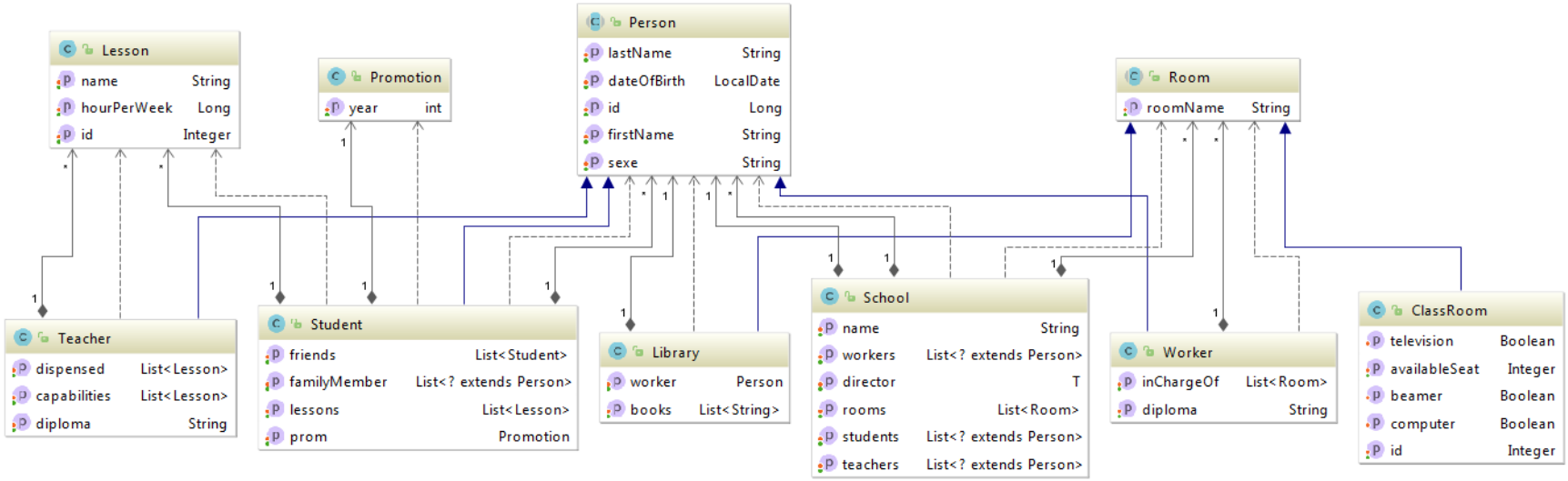


FIGURE 8.1 – Diagramme de Classe du modèle École

Code Sources