

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2016–2017

**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 le développement d'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 nœuds 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 `Neo4J` 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, OrientDB, Archipelago

ABSTRACT

In the scope of this master thesis, a new Java framework was developed to apply the graph theory to databases. This framework presents interesting qualities such as abstraction, simplification and configurability.

Archipelago framework abstracts graph-oriented databases. Currently, three methods are available : a first method that persists information, a second method that links nodes together with the help of a relation with or without properties and a third method that gives the ability to query databases and to retrieve persisted objects states. The framework can be used with both `Neo4J` and `OrientDB`.

This master thesis presents the framework implementation, its applications and its strengths and weaknesses.

Keywords : Framework, databases, Java, Graphs, Neo4J, OrientDB, Archipelago

Avant-propos

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

Je remercie également les entreprises dans lesquelles j'ai travaillé, de m'avoir laissé l'opportunité de poursuivre le cursus universitaire.

Et, enfin, je tiens tout autant à remercier ma belle-mère C. NIGOT pour avoir pris du temps à me relire, ma collègue de classe L. CASTIN 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

1	Introduction	1
I	État de l'art	3
2	L'évolution du NoSql	4
2.1	L'historique	4
2.2	Le questionnement	6
2.3	Le positionnement	7
3	Background technique	8
3.1	Neo4J	8
3.1.1	Description	8
3.1.2	Langage de requête	8
3.1.3	Communication	9
3.2	OrientDB	9
3.2.1	Description	9
3.2.2	Langage de requête	10
3.2.3	Communication	11
3.3	Comparaison	11
4	Solutions existantes	13
4.1	Les librairies	13
4.1.1	Librairie Neo4J	13
4.1.2	Librairie OrientDB	14
4.2	Les frameworks	15
4.2.1	Hibernate	15
4.2.2	Spring Data	15
4.2.3	Apache TinkerPop	16
II	Contribution	18
5	Le framework	19
5.1	Exemple illustratif	19
5.2	Piste de normalisation	19
5.3	Utilisation	20
5.3.1	La configuration	21

5.3.2	La persistance	22
5.3.3	La liaison	26
5.3.4	La récupération	27
5.4	Schéma conceptuel	30
5.4.1	La structure d'Archipelago	30
6	Evaluation	33
6.1	Points forts	33
6.1.1	La facilité	33
6.1.2	La configurabilité	33
6.1.3	L'abstraction	34
6.1.4	La simplification	34
6.1.5	La modularité	34
6.2	Points à améliorer	34
6.2.1	Les performances	34
6.2.2	Les conditions	35
6.2.3	La récupération d'informations	37
6.2.4	La propreté du code	39
6.3	Questionnaire	39
6.3.1	Contenu quantitatif	39
6.3.2	Contenu qualitatif	45
7	Conclusion	48
8	Perspectives	50
8.1	Amélioration des performances	50
8.2	Amélioration du système de conditions	50
8.3	Amélioration de l'exploitation des relations	52
8.4	Ajout d'algorithmes célèbres de la théorie des graphes	52
III	Annexes	56

Glossaire :

SQL	S tructured Q uery L anguage
NoSQL	N ot O nly S QL
SGBD	S ystème de g estion de b ase de d onnées
SGBDR	S ystème de g estion de b ase de d onnées r elationnelles
BDOG	B ase de d onnées o rientée g raphe
CRUD	C reate R ead U pdate D eleter : Ce sont les quatre opérations principales pour la persistance de données.
JSON	J ava S cript O bject N otation
Vertex	Sommet
Edge	Bord
Cluster	Littéralement grappe, cela représente en informatique un groupe d'éléments
API	A pplication P rogramming I nterface
REST	R epresentational S tate T ransfer, style d'architecture pour les systèmes hyper-média distribués
GDL	G arlic D efinition L anguage (variante de ODMG [4])
OO	O rienté O bjets
POC	P roof O f C oncept
YAML	Y et A nother M arkup L anguage

Introduction

L'idée de réaliser ce mémoire nous est venue de notre travail en entreprise où nous utilisons un modèle de données très connectées. Nous avons comme mission de remanier ce modèle de données en le rendant totalement historisable. Pour ce faire, nous utilisons une base de données relationnelle mais ce choix nous posait problème, notamment au niveau des références circulaires. Ce genre de difficulté survient quand certains éléments identifiants se retrouvent dépendants d'eux-mêmes et forment ainsi un cycle¹.

Nous recevions nos informations en provenance de plusieurs sources, chacune avait bien entendu sa propre représentation. Un ensemble de méthodes devait donc être mis en place et maintenu pour faire correspondre ces modifications avec la représentation interne que nous avions de l'information. Toutes ces petites modifications rendaient le précédent modèle de moins en moins cohérent et stable. Il nous fallait donc penser à une structure évolutive permettant, pour ces demandes en provenance d'une source spécifique, ces ajouts qui alourdissaient les tables.

Suite à ce questionnement, nous nous sommes interrogé d'une part, sur la pertinence de notre modèle car nous nous heurtions sans cesse à des difficultés multiples (niveau d'abstraction trop faible, changement d'identifiant lors de l'historisation, ...) et, d'autre part, à « l'entêtement » de l'utilisation de bases de données relationnelles.

C'est alors que nous avons pensé à la mise en place d'un outil permettant d'éviter toutes les démarches auxquelles notre équipe s'était confrontée. Cette réflexion nous a fait découvrir les bases de données orientées graphes et les librairies qui permettent de les utiliser.

Nous avons pu observer que les bases de données orientées graphes (BDOG cf. glossaire) étaient de plus en plus nombreuses. Nous constatons que chacune développait ses propres fonctionnalités. Nous avons dès lors voulu trouver un framework permettant de communiquer avec ces dernières tel que « JPA » le fait avec les bases de données relationnelles. A notre grande surprise, nous nous sommes aperçu que ce framework semblait ne pas exister.

1. Une entreprise publiait un acte générant des évènements sur une entreprise (potentiellement la même)

Le but de ce mémoire est précisément d'offrir une réponse à ce manque, en produisant un framework qui tentera de présenter les qualités suivantes :

- **Abstraction** : L'utilisateur pourra employer ce framework sans même connaître l'existence de librairies spécifiques.
- **Simplicité** : Les différentes méthodes seront explicites et faciles à intégrer dans une application existante.
- **Configurabilité** : Il réagira en fonction d'un fichier de configuration clair et simple.
- **Évolutivité** : Il permettra de faire évoluer la couche de persistance en fonction du modèle de données.

Nous l'avons nommé **Archipelago**, traduction anglaise du mot archipel signifiant :
« *Ensemble d'îles disposées en groupe et qui, de fait, forment une unité géographique* »[2].

Première partie

État de l'art

L'évolution du NoSql

Dans ce chapitre, nous allons retracer dans un premier temps l'historique des bases de données de type NoSQL. Dans un second temps, nous aborderons les différences entre les bases de données relationnelles et les bases de données orientées graphe. Enfin, au vu de ce que nous avons appris précédemment, nous spécifierons notre recherche par rapport à celles existantes et décrites dans la section 2.1.

2.1 L'HISTORIQUE

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 grands challenges d'un SGBD, une communauté de développeurs pense que ces systèmes devraient pouvoir offrir d'autres fonctionnalités [6].

La grande force des SGBDR¹ 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[3].

A partir des années 1980, le modèle relationnel supprime 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, etc. ; bref de nouvelles opportunités sont apparues au XXI^e siècle.

De nos jours, les unités de stockage sont de plus en plus accessibles au grand public. Les entreprises comme Google, Amazon, Facebook, Twitter, etc., sont tour à tour arrivées aux limites du modèle relationnel [14] comme la partition, l'évolutivité en fonction de la charge, etc. 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.

Les différentes entreprises impliquées dans le développement des systèmes de gestion de base de

1. Systeme de **G**estion de **B**ase de **D**onnées **R**elationnelles

données l'ont bien compris et recherchent des solutions alternatives telles que le NoSql[21]. Ce mouvement NoSql pour « Not Only SQL » se base sur différentes représentations en provenance du théorème de CAP² aussi connu sous le nom de théorème de Brewer. Ce théorème prouve qu'il est impossible de garantir simultanément les trois contraintes suivantes :

- La cohérence
- La haute disponibilité
- La tolérance au partitionnement

Comme le mouvement NoSQL veut promouvoir la haute disponibilité, il doit dès lors faire « main basse » soit sur la cohérence, soit sur la tolérance au partitionnement. Cette possibilité de pouvoir choisir entre trois éléments ne rassure toutefois pas les adeptes du relationnel [9].

Par souci de classification, nous allons distinguer quatre usages principaux du NoSql :

- **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 à 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 distinguer aussi quatre 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** : Inspirée 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 type graphe sont de plus en plus nombreuses, les relations entre les éléments permettent de parcourir le graphe de manière très performante [28], le rendant de plus en plus intéressant pour les entreprises possédant des millions de données. Dès lors, l'utilisation de ce genre de SGBD est dès lors tout à fait recommandée pour des entreprises intéressées par les relations entre ces données, telles que des profils sociaux, des liens de cause à effet, des liens géographiques et bien d'autres encore [7][26][3].

L'idée de mettre en place des méthodologies de simplification entre les applications et les bases de données n'est pas nouvelle. En 1998, Tova Milo et Sagit Zohar ont proposé un système permettant de convertir des valeurs reçues depuis un fichier XML et son schéma DTD, en

2. Acronyme de « Coherence », « Availability » et « Partition tolerance »

instance de OODB[20].

En 1999, nous pouvons observer que la mise en place d'intergiciels³ permet de générer des « Wrapper » à l'aide du GDL⁴, et offre la possibilité d'abstraire l'information[23]. Les « Wrapper » ainsi que les « Unwrapper » procurent aux frameworks d'automatisation un moyen de contrôler la structure de représentation des données. Grâce à ce contrôle, il est aisé de les utiliser et de les adapter pour les rendre conformes à une vue, une table, un schéma, ... Ces procédés de « Wrapper » et « Unwrapper » ont constitué des solutions pour communiquer avec des systèmes dits « legacy ».

Nous pouvons nous apercevoir qu'il existe des systèmes de correspondance entre différents schémas de représentation. Par le biais d'XML, il est possible de faire transiter une information depuis un ancien schéma de données vers un nouveau[19].

Nous avons pu observer également qu'il existe des frameworks permettant de se protéger contre l'évolution du modèle de représentation. En effet, une étude montre que près de 70% des requêtes[8] sont perdues à chaque nouvelle version de schéma. Pour pallier cette perte, certains frameworks génèrent des API Java sur base du schéma ER⁵ de la base de données[5].

2.2 LE 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 davantage orientée « Big Data » ; les entreprises veulent tout stocker afin d'avoir le plus d'informations possible[16]. Or comme le relationnel cadenasait 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 un meilleur contrôle de l'intégrité des données aux applications.

Si l'objectif de l'application est de représenter un système récursif, comme par exemple, une arborescence de fichiers, des données généalogiques, etc., 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[18]. 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 ensembliste 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

3. Logiciel permettant l'échange d'informations entre différentes applications informatiques.

4. **G**arlic **D**efinition **L**anguage

5. **E**ntity **R**elation

d'une 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[29].

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 clés étrangères définies 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[29].

2.3 LE POSITIONNEMENT

Le framework que nous souhaitons développer ne se focalisera pas sur l'une ou l'autre spécification. Tel que décrit dans la section 2.1, certains frameworks permettent d'aider l'application lors de son évolution, d'autres permettent de réconcilier d'anciennes bases de données, ou encore aident au design du modèle. Dans notre cas, nous aimerions fournir un système simple permettant de faciliter le design d'une application, en cachant intégralement le schéma de données au niveau de la couche de persistance (abstraction de la couche de persistance).

L'utilisation de ce framework permettra aussi de faire évoluer le modèle d'information, sans perte de fonctionnalité.

Background technique

Lors de ce point, nous vous présenterons les différentes fonctionnalités qu'offrent les bases de données Neo4J et OrientDB. Nous entreprendrons également une comparaison entre ces deux BDOG afin de mettre en lumière ce qui nous semble être les points forts et les points faibles de chacune.

3.1 NEO4J

3.1.1 DESCRIPTION

Créée par Neo Technologie, une société américaine et suédoise, elle est actuellement la base de données orientée graphe la plus utilisée dans le monde[24]. Développée 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 peut contenir une ou plusieurs propriétés. Les propriétés sont des couples clé/valeur de type simple, comme des chaînes de caractères ou des valeurs numériques, des coordonnées spatiales, etc[22].

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

3.1.2 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'analyses de proximité.

```
1 CREATE (James:Person {name:"James"})
2 CREATE (John:Person {name:"John"})
3 CREATE (gilles:Person {name:"Gilles"})
4 CREATE (jane:Person {name:"Jane"})
5 CREATE (tim:Person {name:"Tim"})
6 CREATE (James)-[:PARENT_OF]->(gilles)
```

```

7 CREATE (John)-[:PARENT_OF]->(jane)
8 CREATE (jane)-[:PARENT_OF]->(tim)
9 CREATE (gilles)-[:PARENT_OF]->(tim)

```

Ces requêtes vont créer 5 nœuds et 4 relations :PARENT_OF, nous pouvons aisément comprendre que « James » est parent de « Gilles ».

```

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

```

La requête ci-dessus, va retourner tous les nœuds distincts qui ont une relation :PARENT_OF avec un autre nœud.

```

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

```

Celle-ci, quant à elle va retourner toutes les personnes qui sont « parents de parents » 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 ».

3.1.3 COMMUNICATION

La BDOG Neo4J peut être utilisée sous plusieurs formes.

La première option est une solution embarquée, ce choix peut être très intéressant en alternative 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 « Neo Technologie », est grandement conseillé pour communiquer avec ces serveurs distants.

Son utilisation est simple grâce à l'utilisation de la librairie native à Neo4J pour Java (neo4j-java-drive) ou avec l'utilisation de l'API Rest déployée en même temps que le SGBD.

3.2 ORIENTDB

3.2.1 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[24], elle offre de nombreuses fonctionnalités intéressantes que nous allons décrire[17].

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** : sources de contenus, pouvant être considérées comme des containers de données. On peut les comparer avec les lignes 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, etc.

Chaque Document ou Vertex appartient à une « Class » qui 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.

3.2.2 LANGAGE DE REQUÊTE

OrientDB utilise une sorte de SQL avancé 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
```

Nous pouvons observer qu’OrientDB se charge de qualifier les documents de plusieurs méta-data 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, quant à lui représente la structure du document.

1. On laisse l’utilisateur continuer sa transaction et le refus se fera lors du « commit » s’il y a eu une modification concurrente du même document

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

TABLE 3.1 – Résultat de la requête « SELECT FROM V » dans le BDOG OrientDB.

```
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 [27].

3.2.3 COMMUNICATION

OrientDB embarque une API Rest complète[12]. Toute action pouvant être faite sur les interfaces web et consoles peut être reproduite au travers de cette API.

Outre la communication via le protocole **http**, nous pouvons aussi accéder à la base de données via une des trois librairies Java développée par OrientDB.

3.3 COMPARAISON

Suite aux recherches effectuées dans la section 2.1, nous avons tenté de dresser un tableau comparatif permettant de distinguer Neo4J et OrientDB sur base des critères nous semblant les plus pertinents.

Critère \ BDOG	Neo4J	OrientDB
Format de données	JSON	JSON
Langage	Cypher	SQL ²
Schéma de donnée strict	X	✓
Principe d'héritage	X	✓
Clustering	Configurable	Natif
Communication	BOLT (protocole propriétaire)	REST (via HTTP)
Environnement	Client/Server ou embarquée	Client/Server uniquement
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
Modèle de représentation	Uniquement graphe	Multimodèle

2. Langage fortement similaire à SQL avec certaines légères modifications

Création de user	En cours de développement	✓
Sécurité	Par défaut : simple connexion donnant un accès total	Présence de groupe d'utilisateurs ainsi qu'un ensemble de droits singuliers
Interface web	✓	✓
Implémentation de fonctions	✓	✓
Rapidité ³	7m 10s	4m 31s

TABLE 3.2 – Tableau comparatif entre Neo4J et OrientDB.

Au vu de ce tableau comparatif, nous nous apercevons qu'OrientDB, grâce à son système d'autorisation, semble ainsi correspondre à une approche plus standardisée d'une base de données. En utilisant le langage SQL comme langage de requête, OrientDB conforte à nouveau les utilisateurs de bases de données relationnelles en leur permettant d'utiliser un langage connu. La rapidité d'exécution est elle aussi favorable à OrientDB. Cependant le fait de devoir configurer quelque peu le serveur pourrait limiter son intérêt lors de l'implémentation de petites applications.

Neo4J a lui préféré jouer la carte de la spécificité :

- Nouveau langage
- Nouveau protocole
- Accès simple à l'information

Grâce à ses fonctionnalités et ses structures de représentation de l'information totalement évolutives, Neo4J se fait une place à part entière dans l'univers des BDOG.

Avec Archipelago, l'utilisation de Neo4J ou d'OrientDB ne nécessitera aucun apprentissage supplémentaire ni aucun changement dans l'application.

3. 10 000 requêtes sur le même graphe de données avec la même requête au travers d'un appel Java depuis un ordinateur vers un serveur distant (même hébergement pour les deux BDOG).

Solutions existantes

« *Un peuple qui oublie son passé se condamne à le revivre.* »¹

Dans le même état d’esprit, nous tenterons de vous décrire les solutions existantes et ce, dans le but que notre recherche puise le potentiel de ces librairies afin de construire un framework qui aspire à les utiliser à bon escient. À l’inverse, certaines librairies ne seront pas exploitées car elles ne nous permettent pas d’atteindre nos objectifs de simplification et/ou d’abstraction (cf. 2.3).

4.1 LES LIBRAIRIES

Afin que nous parlions d’un même concept, nous allons définir le terme « librairie » comme suit :

« *Le Java Class Library (JCL) est une bibliothèque standard du langage Java, qui contient un ensemble de bibliothèques logicielles chargées dynamiquement. ... Comme les bibliothèques standards des autres langages, il fournit aux programmeurs un ensemble de fonctions bien connues (par exemple les expressions régulières).* »²

4.1.1 LIBRAIRIE NEO4J

Une librairie Java développée par Neo Technologie existe et 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 et le protocole « Bolt » :

```

1 Driver driver = GraphDatabase.driver("bolt://localhost:7687",
    AuthTokens.basic("matrix", "neo"));
2 Session session = driver.session();

```

1. Winston Churchill

2. https://fr.wikipedia.org/wiki/Java_Class_Library

```
3 session.run("CREATE (a:Person {firstName:{name},lastName:{lastName}})",
    parameters("firstName","Gilles","lastName","Bodart"));
4 session.close();
5 driver.close();
```

Bien que cette librairie permette d'accéder directement au langage Cypher depuis un code Java, elle ne peut malheureusement être utilisée avec une autre BDOG que Neo4J.

4.1.2 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 use cases classiques d'un utilisateur et 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, elle est plus simple d'utilisation mais offre une utilisation plus atomique. Par exemple, les relations ne peuvent être qu'unidirectionnelles tandis que la bidirectionnalité est offerte dans l'API précédente (**graph API**). Un **document** est un élément représentant l'information. Par défaut, aucune relation ne sera construite, nous devons dès lors gérer manuellement chaque lien entre documents.

La dernière librairie se focalise sur les objets Java, elle n'a plus été améliorée depuis la version 1.5 d'OrientDB³.

3. 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 »

4.2 LES FRAMEWORKS

Ainsi que nous avons défini le terme « librairie » dans la section 4.1 , nous allons préciser le terme « framework », utilisé à de nombreuses reprises dans ce travail.

*« Un framework désigne en programmation informatique un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application. C'est le framework, ou structure logicielle en français, qui établit les fondations d'un logiciel ou son squelette. »*⁴

4.2.1 HIBERNATE

Hibernate, soutenu et développé par JBoss, a mis au point un système de relations entre le code Java et le SGBD Neo4J, 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 n'ait été implémentée pour communiquer avec OrientDB, un ticket est ouvert (OGM-855)⁵ depuis 2 ans auprès d'Hibernate.

En voyant la syntaxe utilisée dans ce framework, nous sommes dans l'obligation de spécifier le type de relation entre deux nœuds. Ceci freine l'évolution aisée de l'application dans le temps. Nous aimerions permettre au développeur de créer une relation, en utilisant une unique annotation que nous appellerions **@bridge** en référence aux ponts qui lient les îles d'un archipel.

4.2.2 SPRING DATA

Spring data, comme Hibernate, possède un module de communication avec le SGBD Neo4J. Il intègre au célèbre Framework Spring, les fonctionnalités de Neo4J-OGM⁶ qui permettent de qualifier une classe avec certaines annotations facilitant la sérialisation et la désérialisation vers la représentation graphique.

Exemple :

```
1 // Spécification du nom du Label du nœud
2 @NodeEntity(label="Film")
3 public class Movie {
4 // Spécification de l'identifiant technique donné par Neo4J
5     @GraphId Long id;
```

4. <http://www.journaldunet.com/solutions/pratique/dictionnaire-du-webmastering/outils/19466/framework-definition-traduction.html>

5. <https://hibernate.atlassian.net/browse/OGM-855>

6. Object Graph Mapper

```

6 // Binding de la propriété "title" spécifiée dans l'annotation
7   @Property(name="title")
8   private String name;
9 }
10
11 // Pas de spécification du nom du Label,
12 // il prendra par défaut le nom de la classe
13 @NodeEntity
14 public class Actor extends DomainObject {
15
16     ...
17
18 // Spécification d'une relation partante de type "ACTED_IN"
19   @Relationship(type="ACTED_IN", direction=Relationship.OUTGOING)
20   private List<Role> filmography;
21 }
22 // définition du type de Relation "ACTED_IN"
23 @RelationshipEntity(type="ACTED_IN")
24 public class Role {
25     @GraphId
26     private Long relationshipId;
27     @Property
28     private String title;
29 // Spécification du nœud de départ
30     @StartNode
31     private Actor actor;
32 // Spécification du nœud d'arrivée
33     @EndNode
34     private Movie movie;
35 }

```

Nous nous inspirerons grandement de ce framework pour le développement futur d'Archipelago, tout en y ajoutant la fonctionnalité d'interchangeabilité entre les différents BDOG.

4.2.3 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 bibliothèques externes développées par des 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 3.1.2 s'est grandement inspiré de ce dernier[13].

Exemple d'utilisation :

```

1 gremlin> g.V() /* récupère tous les nœuds du graphe */

```

```

2 ==>v[1]
3 ==>v[2]
4 ==>v[3]
5 ==>v[4]
6 ==>v[5]
7 ==>v[6]
8 gremlin> g.V(1) /* récupère le nœud d'Id 1 du graphe*/
9 ==>v[1]
10 gremlin> g.V(1).values('name') /* récupère le nœud d'Id 1 du graphe et
    renvoie sa propriété 'name' */
11 ==>marko
12 gremlin> g.V(1).outE('knows') /* récupère les relations 'knows' partant du
    nœud d'Id 1 */
13 ==>e[7] [1-knows->2]
14 ==>e[8] [1-knows->4]
15 gremlin> g.V(1).outE('knows').inV().values('name') /* retourne le nom des
    personnes que le nœud 1 connaît */
16 ==>vadas
17 ==>josh
18 gremlin> g.V(1).out('knows').values('name') /* même résultat */
19 ==>vadas
20 ==>josh
21 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 connaît */
22 ==>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.

Nous souhaitons créer Archipelago pour permettre la réutilisation des modèles OO⁷ existant et permettre ainsi aux développeurs, de découvrir les BDOG sans avoir besoin d'utiliser un autre langage et/ou environnement, comme TinkerPop l'exige.

Deuxième partie

Contribution

Le framework

Au sein de cette section, nous décrivons notre framework nommé « Archipelago ». Nous l'introduisons par un exemple illustratif d'utilisation des bases de données orientées graphes et nous retraçons l'ensemble des fonctionnalités que permet ce framework.

5.1 EXEMPLE ILLUSTRATIF

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 données 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.

5.2 PISTE DE NORMALISATION

Comme nous l'avons présenté dans le point 2.1, nous pouvons constater de par l'implémentation du théorème de CAP, que chaque SGBD NoSQL se concentre sur l'une ou l'autre fonctionnalité.

Dans cette partie, nous allons tenter d'abstraire un ensemble d'éléments afin de permettre aux futurs utilisateurs de pouvoir communiquer avec des BDOG sans pour autant devoir connaître leurs spécifications.

Notre démarche dans la proposition de cette fonctionnalité d'abstraction, est une démarche favorisant la découverte de nouvelles implémentations de BDOG afin de satisfaire au mieux les attentes des entreprises. En plus de l'abstraction des librairies spécifiques aux BDOG, Archipelago devra contenir les fonctionnalités suivantes :

1. Utilisation d'une librairie unique pour toutes BDOG.
2. Contrôle sur les propriétés et les relations.
3. Opération de base de persistance CRUD.
4. Système de transaction.

La suite de ce mémoire portant sur le développement de ce framework Java, vous montrera comment utiliser Archipelago.

5.3 UTILISATION

Les figures 5.1 et 5.2 offrent une approche schématisée de l'utilisation d'Archipelago. La première vous décrit le fonctionnement de l'étape de persistance, tandis que la deuxième vous expose la manière dont nous allons récupérer l'information. Plus tard, dans les sections 5.3.2 et 5.3.4, vous pourrez apercevoir des schémas plus détaillés de ces fonctionnalités.

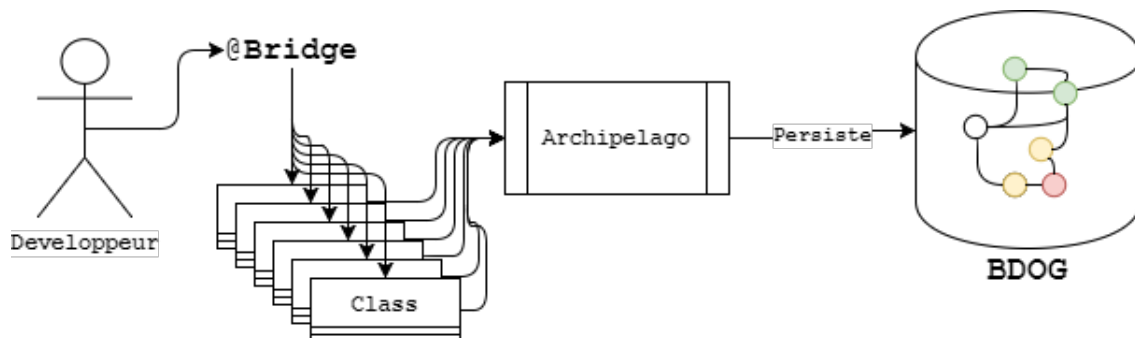


FIGURE 5.1 – Un développeur applique les annotations bridge sur les classes et persiste les objets Java avec Archipelago dans la BDOG.

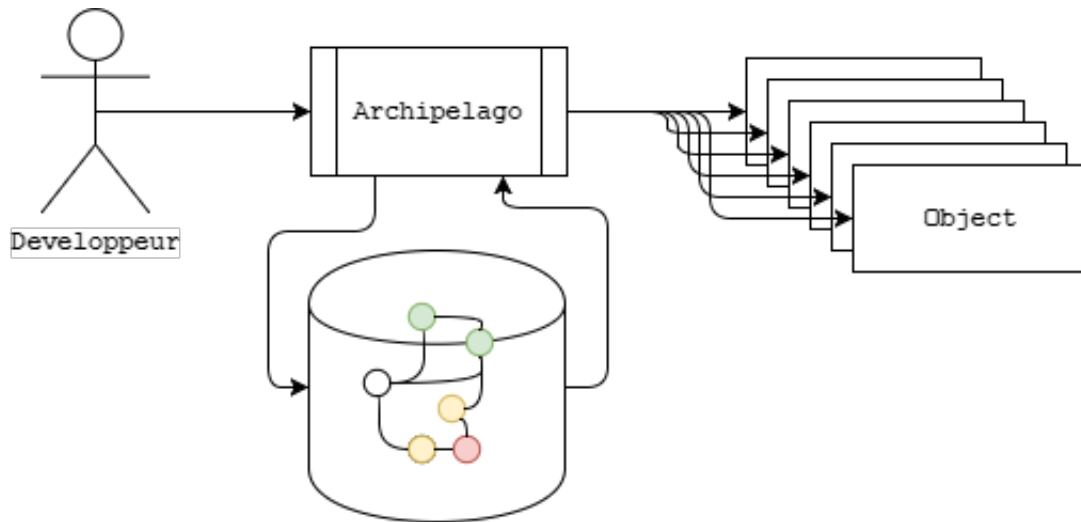


FIGURE 5.2 – Un développeur récupère des objets depuis une BDOG en utilisant Archipelago.

5.3.1 LA CONFIGURATION

Archipelago permet de se lier à Neo4J 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. Par facilité de développement et de testing, nous avons décidé arbitrairement de figer le nom ainsi que l'emplacement de ce fichier dans la première version de ce framework. Il est cependant aisé de modifier cette obligation en la rendant paramétrable dans une potentielle deuxième version.

Voici le canevas de ce fichier :

```

1 {
2   "database": {
3     "type": "", // Choix entre NEO4J et ORIENT_DB
4     "username": "",
5     "password": "",
6     "url": "", // URL de la base de données
7     "name": "", // Nom de la base de données
8     "port": 1234 // Port de communication
9   },
10  "deepness": 3, // Profondeur de persistance voir ci-après
11  "domainRootPackage": "org.archipelago.test.domain.school"
12    // Package où se trouvent les classes du domaine
13 }

```

La seule autre modification à apporter, est l'apposition de l'annotation `@Bridge`, comme expliqué dans la section 4.2.1. Cette annotation sera utilisée par Archipelago afin de spécifier les relations que nous souhaitons créer entre les différents objets.

5.3.2 LA PERSISTANCE

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

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 (5.3.1).

Afin d'assurer la persistance des éléments liés, nous avons opté pour une approche récursive en « depth-first » avec marquage de sommet. Lors du parcours du graphe de données, représenté par les différentes références entre les objets Java, nous enregistrons l'identifiant venant d'être généré par la BDOG dans une `Map<Object, Object>`¹. En effet, pour créer un lien entre deux éléments, il faut absolument que ceux-ci soient présents dans la base de données. Garder la trace des identifiants nous permet aussi d'accélérer la phase de liaison entre les nœuds, car cela nous évite un appel supplémentaire à la BDOG.

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 des modèles ayant des relations bidirectionnelles comme suit :

```
1 Archipelago arch = Archipelago.getInstance();
2 // Création d'un objet Student
3 Student gilles = new Student();
4 // Création d'un autre objet Student
5 Student thomas = new Student();
6
7 // Lien entre Student et un autre Student
8 gilles.setFriend(thomas);
9 // Lien entre Student et un autre Student
10 thomas.setFriend(gilles);
11
12 arch.persist(gilles);
13
14 // Boucle infinie : gilles -ami-> thomas -ami-> gilles -ami-> thomas -> ...
15 // Comme "deepness" vaut 3 : gilles -ami-> thomas -ami-> gilles --> STOP
```

En effet, comme nous utilisons la récursivité au travers de l'introspection des classes, nous pouvons tomber sur une classe faisant référence à elle-même², ce qui entraîne la dite boucle infinie. Ainsi avec la valeur sentinelle contenue dans la propriété « deepness », nous rompons la boucle de persistance générée lors du parcours des références après avoir atteint la profondeur souhaitée. Archipelago continue à l'aide du back-tracking la suite de la persistance si nécessaire.

1. Représentation d'un ensemble de clé/valeur dont la clé est l'objet que nous venons de persister et la valeur est l'identifiant.

2. voir exemple ci-dessus

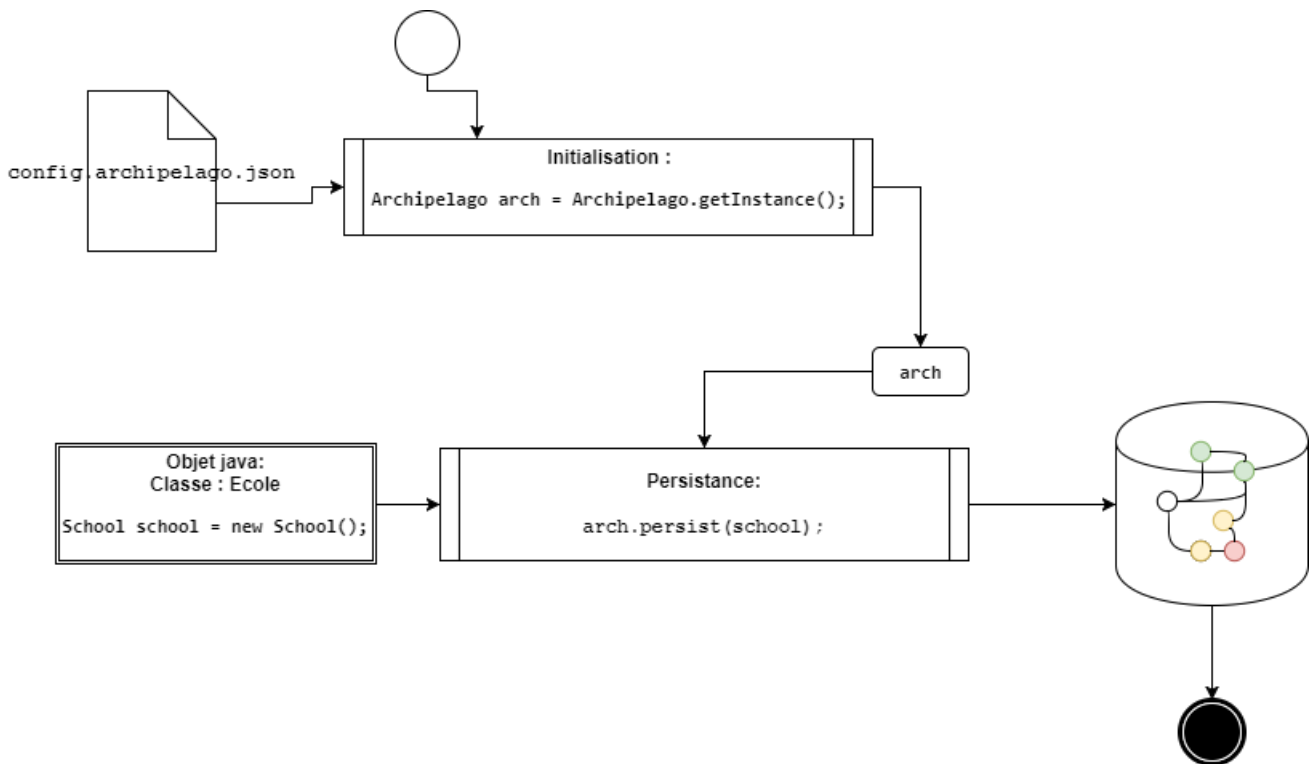


FIGURE 5.3 – Flux de persistance d’un objet au travers du framework Archipelago.

Pour ce qui est de la persistance des liens potentiels entre deux objets, nous avons fait le choix de créer des relations unidirectionnelles. Afin de les créer, il faut apposer l’annotation `@Bridge` dans les classes possédant des liens vers d’autres classes ; ces annotations spécifieront les origines des futures relations.

Il est nécessaire de décrire le nom de la future relation dans l’annotation par le biais de l’attribut `descriptor`. Lors du développement du framework, nous avons éprouvé quelques difficultés à récupérer ces relations depuis la base de données, dès lors, nous imposons que la valeur de cet attribut soit unique au sein même d’une classe.

Nous avons cependant ajouté la possibilité d’obtenir automatiquement une relation réciproque lors de la persistance, en associant la valeur `true` dans l’attribut `biDirectionnal` de l’annotation.

Exemple avec la Classe *Student.java*

```

1 public class Student extends Person {
2
3     // Spécification d’un lien entre Student et Lesson,
4     // le nom de cette relation sera Follow
5     @Bridge(descriptor = "Follow")
6     private List<Lesson> lessons = new ArrayList<>();
7
8     // Spécification d’un lien entre Student et Student,
9     // le nom de cette relation sera FriendOf
10    @Bridge(descriptor = "FriendOf")
11    private List<Student> friends = new ArrayList<>();
12
13    // Spécification d’un lien entre Student et une sous classe de Person,

```

```

14 // le nom de cette relation sera FamilyMember et sera bidirectionnelle
15 @Bridge(descriptor = "FamilyMember", biDirectionnal = true)
16 private List<? extends Person> familyMember = new ArrayList<>();
17
18 // Spécification d'un lien entre Student et Promotion,
19 // le nom de cette relation sera GraduatedIn
20 @Bridge(descriptor = "GraduatedIn")
21 private Promotion prom;
22
23 ...

```

L'intérêt de cette utilisation est qu'au moindre changement de modèle tel qu'un ajout de propriété, l'ensemble de la couche de persistance sera adapté 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 unidirectionnelles ou bidirectionnelles.

```
1 Archipelago arch = Archipelago.getInstance();
2
3 // Génération d'un objet School avec l'ensemble de ses liens
4 School school = genSchool()
5
6 arch.persist(gilles);
```

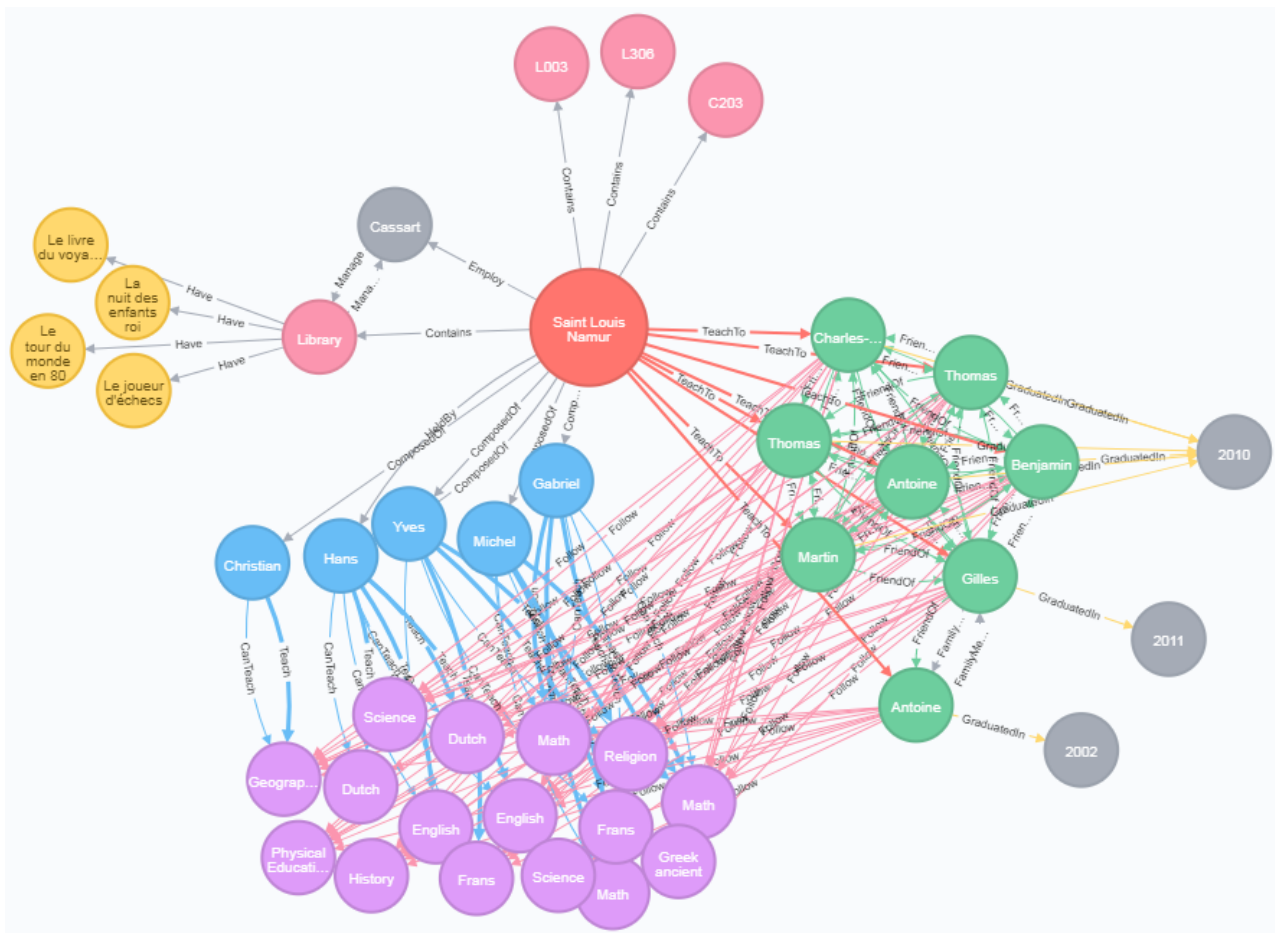


FIGURE 5.4 – Graphe généré dans Neo4J après appel de la méthode persist du framework Archipelago sur un objet School.

5.3.3 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, dont voici un exemple :

```
1
2 Archipelago arch = Archipelago.getInstance();
3 // Création d'un objet City
4 City winterfell = new City("Winterfell");
5 // Création d'un autre objet City
6 City kingsLanding = new City("King's Landing");
7
8 // Création d'un objet Road
9 Road road= new Road();
10 // spécification de la propriété distance à500 (de type long)
11 road.setDistance(500l);
12 // spécification de la propriété unité à"leagues"
13 road.setUnit("leagues");
14
15 // création d'un lien entre l'objet winterfell et l'objet kingsLanding
16 // le lien aura les propriétés contenues dans l'objet road
17 // winterfell -road-> kingsLanding
18 arch.link(winterfell, kingsLanding, road);
```

Comme nous l'avons décrit dans l'exemple précédent, nous allons lier ces deux villes par une relation « Road » contenant les propriétés « distance » et « unité ».



FIGURE 5.5 – Résultat obtenu après utilisation de la méthode link d'Archipelago (avec Neo4J).

Il est toutefois possible, si nous le spécifions à l'aide de la valeur `true` dans le quatrième attribut de la méthode, de générer automatiquement la relation réciproque et ainsi créer une relation bidirectionnelle.

```
1 // winterfell -road-> kingsLanding et kingsLanding -road-> winterfell
2 arch.link(winterfell, kingsLanding, road, true);
```

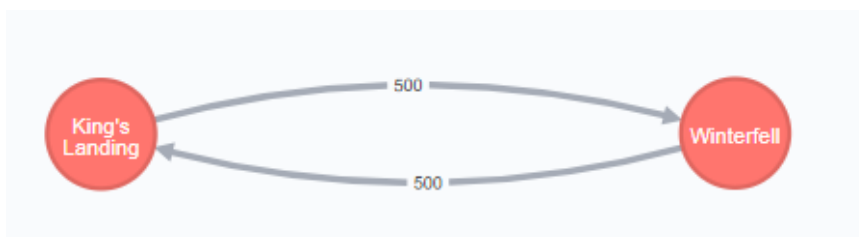


FIGURE 5.6 – Résultat obtenu après utilisation de la méthode link bidirectionnelle d'Archipelago (avec Neo4J).

5.3.4 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 la BDOG soit OrientDB ou Neo4J, nous serons en mesure de récupérer notre objet sans changer notre code source. Dans le fichier de configuration spécifié dans la section 5.3.1, une propriété nommée `domainRootPackage` est présente. Celle-ci est utilisée par notre utilitaire de réflexion afin de connaître l'ensemble des classes potentiellement récupérables depuis la BDOG. Il sera dès lors impossible de récupérer un objet si la classe qui le décrit, ne fait pas partie du package ou d'un sous-package exposé par la valeur contenue dans cette propriété.

Voici un exemple :

```
1 Archipelago arch = Archipelago.getInstance();
2
3 // Initialisation de la requête
4 ArchipelagoQuery aq = a.getQueryBuilder()
5 // Spécification du type de nœud
6     .of(Student.class)
7 // Génération de la requête
8     .build();
9 List<Object> nodes = arch.execute(aq);
```

La requête, générée par l'exemple précédent va rechercher l'ensemble des étudiants de la base de données ainsi que les objets qui leur sont liés.

Un système de conditions est également mis en place ; il permet d'affiner les résultats obtenus :

```
1 Archipelago arch = Archipelago.getInstance();
2
3 // Initialisation de la requête
4 ArchipelagoQuery aq = a.getQueryBuilder()
5 // Spécification du type de nœud
6     .of(Student.class)
7 // Initialisation de la condition
8     .where(
9 // Élément de condition "firstName" = "Gilles"
10         of("firstName", "Gilles"), ConditionQualifieur.Equal)
11 // Génération de la requête
12     .build();
13 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 :

```
1 Archipelago arch = Archipelago.getInstance();
2
3 // Initialisation de la requête
4 ArchipelagoQuery aq = a.getQueryBuilder()
5 // Spécification du type de nœud
6     .of(Student.class)
7 // Initialisation de la condition
8     .where(
9 // Élément de condition "lastName" = "Bodart"
10         of("lastName", "Bodart"), ConditionQualifier.EQUAL)
11 // Ajout d'un lien logique ET avec l'élément "firstName" = "Gilles"
12     .and(of("firstName", "Gilles"), ConditionQualifier.EQUAL)
13 // Ajout d'un lien logique OU avec l'élément "firstName" = "Thomas"
14     .or(of("firstName", "Thomas"), ConditionQualifier.EQUAL)
15 // Génération de la requête
16     .build();
17 List<Object> nodes = arch.execute(aq);
```

Dans notre troisième exemple, le système fournira donc la liste des étudiants ayant « Bodart » comme nom de famille et « Gilles » ou « Thomas » comme prénom.

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 généré la condition :

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

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

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

À L'heure actuelle, il n'est pas possible d'effectuer des requêtes sur les relations entre les objets. Cette piste sera primordiale pour une deuxième version de ce framework.

Comme sus-cité dans la section 5.3, voici un schéma récapitulatif de l'utilisation de la méthode de récupération d'Archipelago.

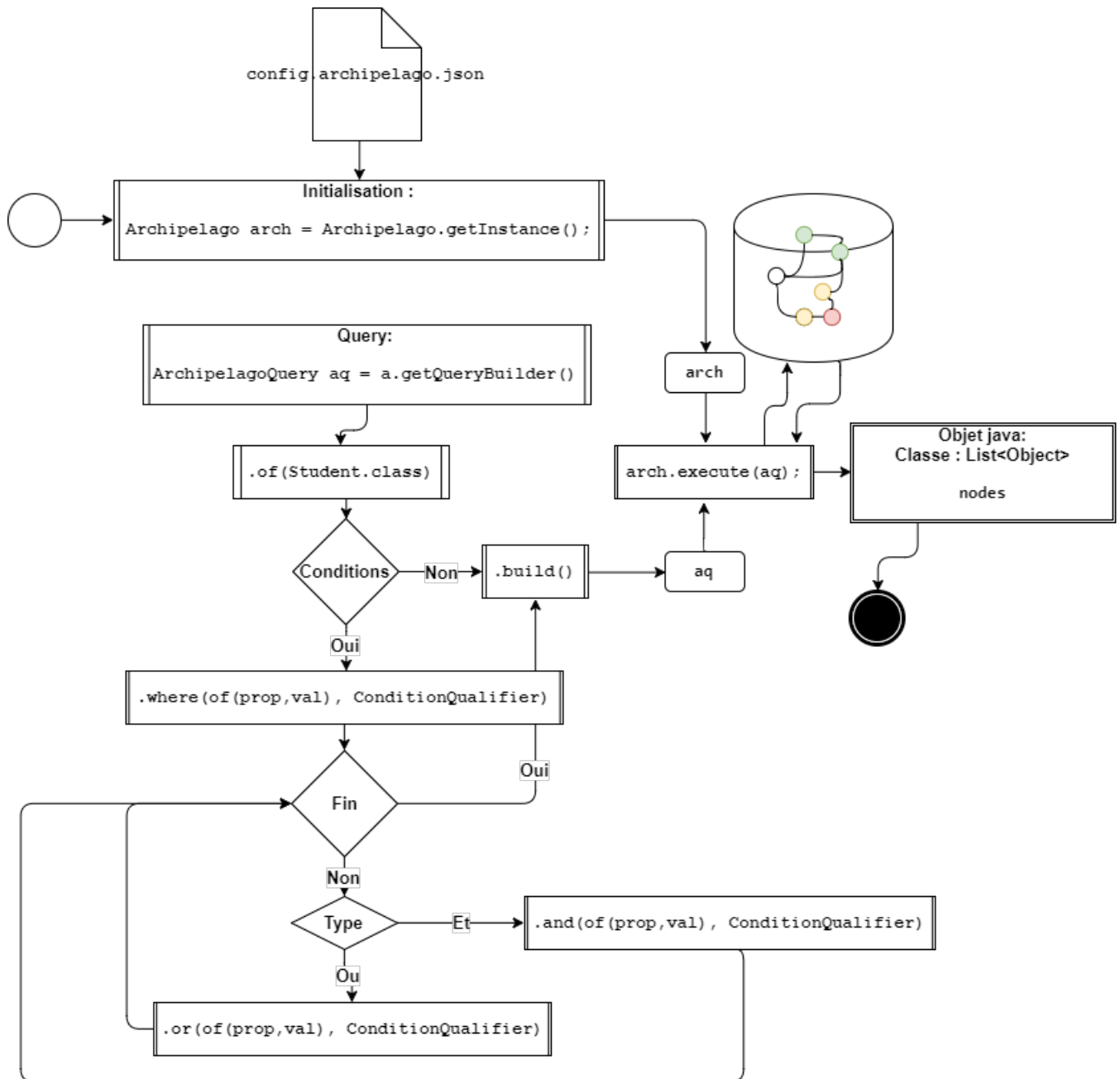


FIGURE 5.7 – Flux de traitement d'une requête de recherche dans Archipelago.

5.4 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 aperçus qu'il existait une grande divergence et une grande complexité entre les implémentations spécifiques des BDOG.

Nous nous sommes dès lors questionnés 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 bibliothèques préexistantes.

Archipelago est donc un framework qui permet d'utiliser ces bibliothèques 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.

5.4.1 LA STRUCTURE D'ARCHIPELAGO

L'objet principal du framework Archipelago est d'être une boîte à outils comprenant les trois opérations décrites³ :

- 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.

3. voir 5.3

SCRIPTBUILDERS

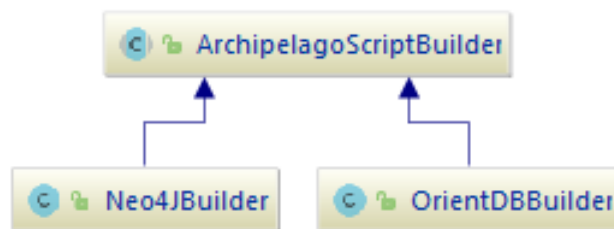


FIGURE 5.8 – Diagramme de classe représentant l’héritage des générateurs de script de persistance composant Archipelago.

Chaque classe doit correspondre à une base de données et implémenter trois méthodes^{4 5} :

Nom de méthode	Description
<code>makeCreate</code>	Cette méthode doit générer une requête permettant de créer un nœud.
<code>makeMatch</code>	Cette méthode doit générer une requête permettant de récupérer l’identifiant correspondant à un objet.
<code>makeRelation</code>	Cette méthode doit générer une requête permettant de lier deux nœuds.

TABLE 5.1 – Relevé des méthodes à implémenter pour hériter de `ArchipelagoScriptBuilder`.

4. Plus un petit nombre d’overload à implémenter.

5. Chacune de ces requêtes doit être écrite dans le langage ciblé par la classe.

QUERYBUILDERS

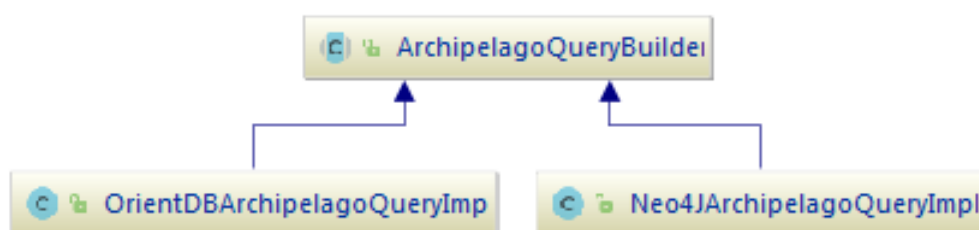


FIGURE 5.9 – Diagramme de classe représentant l’héritage des générateurs de script de recherche composant Archipelago.

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

En voici le détail :

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

TABLE 5.2 – Relevé des méthodes à implémenter pour hériter de `ArchipelagoQueryBuilder`.

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.

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 interconnectées dans des bases de données orientées graphe. Le marquage des sommets lors de cette opération 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 noter que l'utilisation d'un fichier YAML aurait pu fonctionner également. Néanmoins, au vu des faibles différences de représentations, nous ne voyons pas l'intérêt d'employer l'un plutôt que l'autre. C'est pourquoi nous avons opté pour le format JSON que nous affectionnons davantage.

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.

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

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 `ArchipelagoScriptBuilder` 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 relationnelles²; 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 BDOG sans se gêner, telles que par exemple ArangoDB[15] ou encore JanusGraph[1].

6.2 POINTS À AMÉLIORER

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 interconnectée. 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 processus³.

2. voir 2.2

3. Et plus particulièrement en ce qui concerne OrientDB

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.

Sous-conditions

Comme nous l'avons évoqué dans le point 5.3.4, nous pouvons actuellement générer des conditions simples. Pour créer des conditions plus complexes, il faudrait effectuer une refonte du système d'imbrication afin de permettre à l'utilisateur de créer des sous-conditions.

Exemple de condition impossible à générer actuellement :

```
1 (lastName = "Bodart" AND firstName = "Gilles")
2 OR
3 (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 :

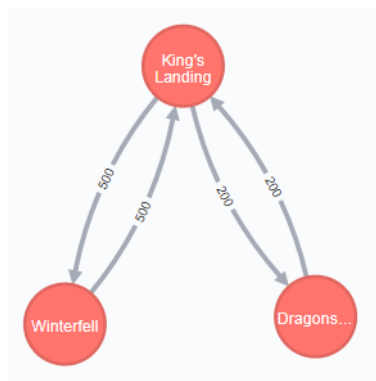


FIGURE 6.1 – Graphes Neo4J composé de 3 noeuds et de 4 relations.

```
1 MATCH (d:City)-[rel:Road]->(a:City)
2 WHERE
3     rel.distance > 300
4     AND rel.unit = "leagues"
5 RETURN (a)
6 // 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 pourrions pas nous limiter à celles situées à plus de 300 lieues.

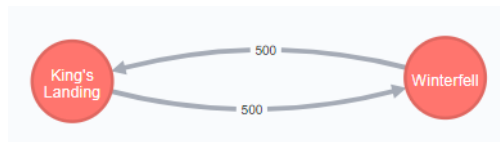


FIGURE 6.2 – Graphes Neo4J composé de 2 nœuds et de 2 relations après exécution du filtre généré par Archipelago.

Conditions sur les éléments liés

Actuellement, l'ensemble des conditions s'applique 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 :



FIGURE 6.3 – Graphe représentant les amis de « Gilles ».

```

1 MATCH (me:Student)-[:FriendOf]->(f:Student)
2 WHERE
3     me.firstName = "Gilles"
4     AND me.lastName = "Bodart"
5     AND f.firstName = "Thomas"
6 RETURN (me)-[]-(f)
7 /* Retourne les étudiants qui se nomment "Gilles Bodart" et lie ses 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.

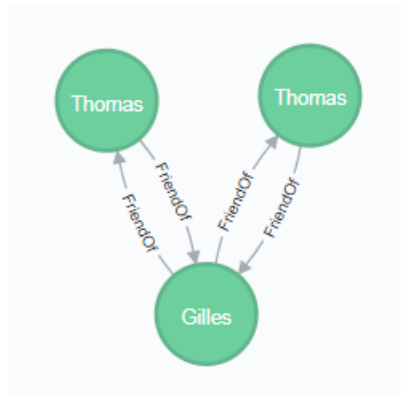


FIGURE 6.4 – Graphe représentant les amis de « Gilles » après exécution du filtre généré par Archipelago.

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.

```

1 public class City {
2
3     private String name;
4
5     @Bridge(descriptor = "Road")
6     private List<City> connected = new ArrayList<>();
7
8     ...

```

```

1 Archipelago arch = Archipelago.getInstance();
2 City winterfell = new City("Winterfell");
3 City kingsLanding = new City("King's Landing");
4 City dragonStone = new City("Dragonstone");
5
6 Road road= new Road();
7 road.setDistance(5001);
8 road.setUnit("leagues");
9
10 Road road2= new Road();
11 road2.setDistance(2001);
12 road2.setUnit("leagues");
13
14 a.persist(winterfell);
15 a.persist(kingsLanding);
16 a.persist(dragonStone);
17 a.link(kingsLanding,winterfell,road, true);
18 a.link(kingsLanding,dragonStone,road2, true);
19
20 ArchipelagoQuery aq = a.getQueryBuilder()
21     .of(City.class)

```

```
22         .where(of("name","King's Landing"), ConditionQualifier.EQUAL)  
23         .build();  
24  
25 List<Object> nodes = arch.execute(aq);
```

Nous récupérerons 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 récupérer⁵ un ensemble de commentaires constructifs par la communauté Java.

6.3 QUESTIONNAIRE

Comme objectif d’évaluation externe, nous avons rédigé un questionnaire afin de soumettre un aperçu de ce framework à des utilisateurs potentiels. Nous l’avons diffusé par le biais d’envois d’e-mail et de partage sur facebook.

Nous avons principalement ciblé des collègues de cours, des collègues de travail ainsi que des connaissances susceptibles d’être intéressées par les BDOG.

Nous avons recueilli 51 réponses. Cependant nous devons soustraire de celles-ci, un certain nombre d’entre elles qui ne présentait aucun intérêt exploitable (réponses fantaisistes notamment). Notre ambition n’était pas de réaliser un questionnaire exhaustif mais bien de prendre l’avis d’utilisateurs éventuels. Il a été diffusé pendant 7 jours environ.

6.3.1 CONTENU QUANTITATIF

Les quatre premières questions concernent l’ensemble des répondants. Nous avons décidé de ne pas filtrer les questions ayant trait à leur identification afin qu’elles témoignent de l’intérêt porté au travail (51 réponses).

Les suivantes concerneront uniquement 32/51 réponses. 19 ayant été considérées comme sans objet. Un second filtrage sera appliqué durant l’analyse du questionnaire. Il est expliqué en légende de la figure 6.10.

4. <https://github.com/GillesBodart/Archipelago>

5. Nous n’en avons pas encore reçu.

Etes-vous ?

51 réponses

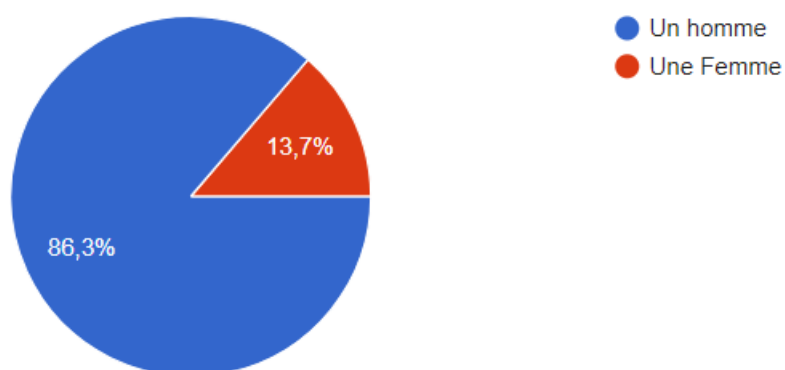


FIGURE 6.5 – Question 1) répartition homme/femme ayant répondu au questionnaire.

Dans quelle tranche d'âge vous situez-vous ?

51 réponses

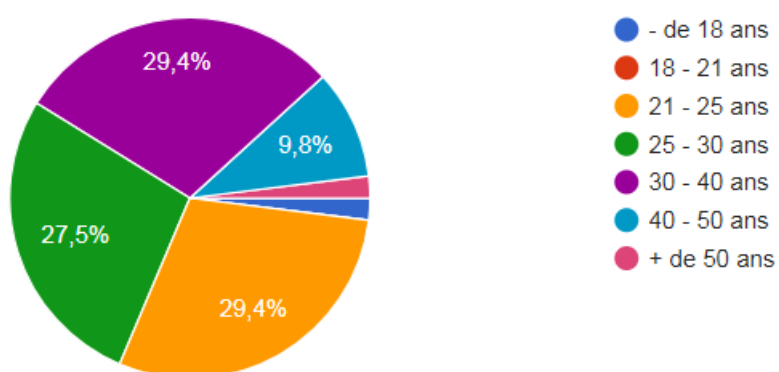


FIGURE 6.6 – Question 2) Tranche d'âge des répondants.

Etes-vous ?

51 réponses

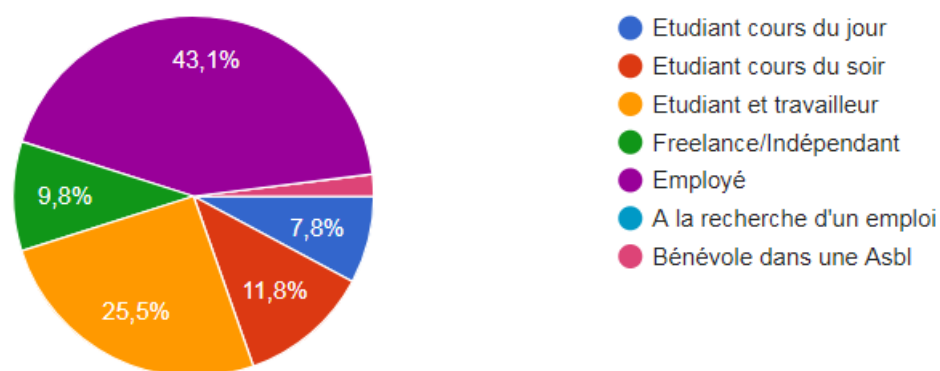


FIGURE 6.7 – Question 3) Situation scolaire et/ou professionnelle des répondants.

Dans quel(s) domaine(s) travaillez-vous ?

51 réponses

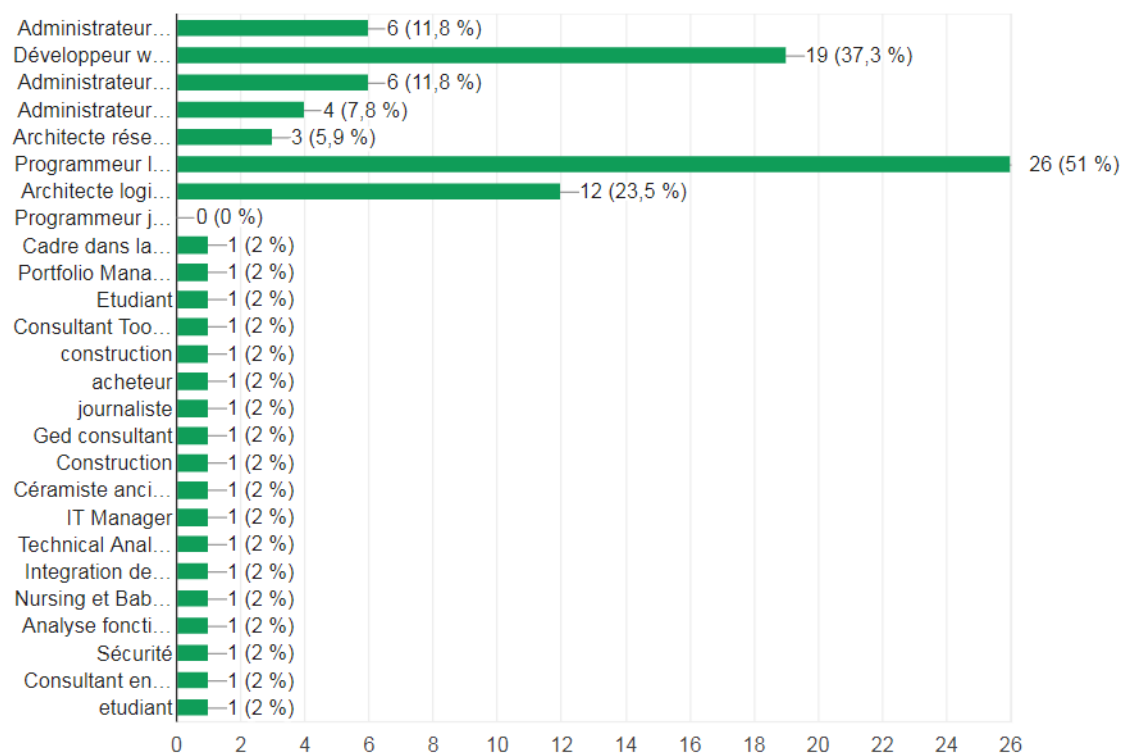


FIGURE 6.8 – Question 4) Domaines d'activité des répondants.

CONNAISSEZ-VOUS L'EXISTENCE DES BDOG ?

32 réponses



FIGURE 6.9 – Question 5) [Données filtrées] Répartition des répondants connaissant les BDOG.

SI VOUS AVEZ RÉPONDU "NON" À LA QUESTION PRÉCÉDENTE : AVEZ-VOUS L'INTENTION DE VOUS Y INTÉRESSER ?

16 réponses



FIGURE 6.10 – Question 6.a) [Données filtrées] Parmi les répondants ne connaissant pas les BDOG (+2 personnes ayant répondu indûment à cette question), nous remarquons que près de la moitié d'entre eux (7/16) ne comptent pas s'intéresser aux bases de données graphes ou n'en voient pas l'utilité actuellement. Dans la suite de l'analyse de ce questionnaire, nous filtrerons davantage encore ceux pour qui les BDOG ne présentent aucun intérêt. C'est la raison pour laquelle notre échantillon ne comptera plus que 25 réponses.

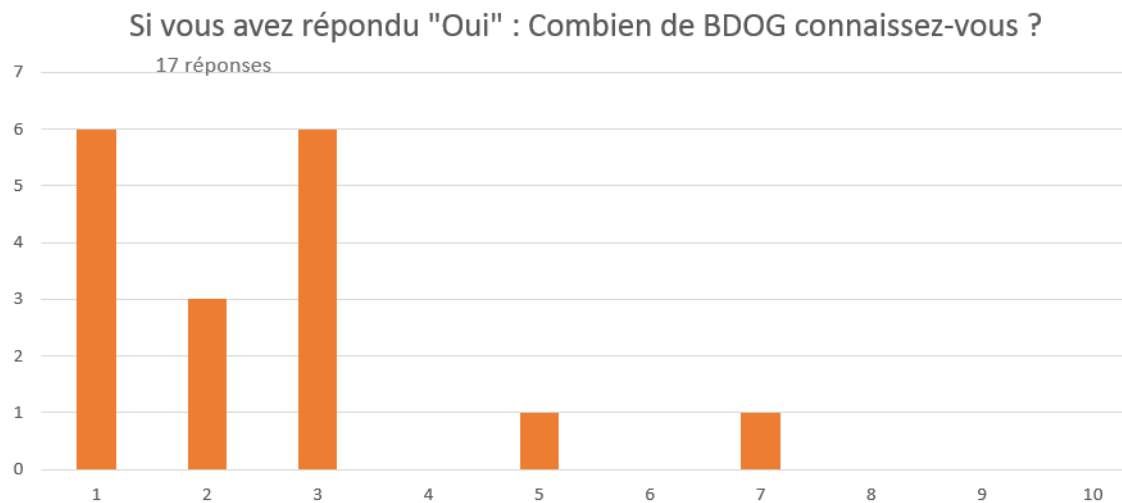


FIGURE 6.11 – Question 6.b) [Données filtrées] Nous avons 18 réponses « Oui » à la question 5, mais suite au deuxième filtre, l'échantillon ne comporte plus que 17 réponses (voir remarque ci-avant). Parmi les répondants connaissant les BDOG, nous remarquons que la majorité d'entre eux (11/17) en connaissent plusieurs.

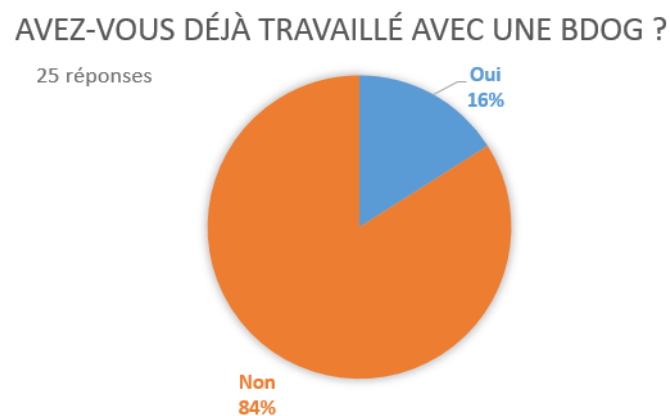


FIGURE 6.12 – Question 7) [Données filtrées] Pourcentage des répondants ayant déjà travaillé avec une BDOG.

CONNAISSEZ-VOUS LA MANIÈRE DE FAIRE INTERAGIR DU CODE JAVA ET CETTE/CES BDOG ?

25 réponses

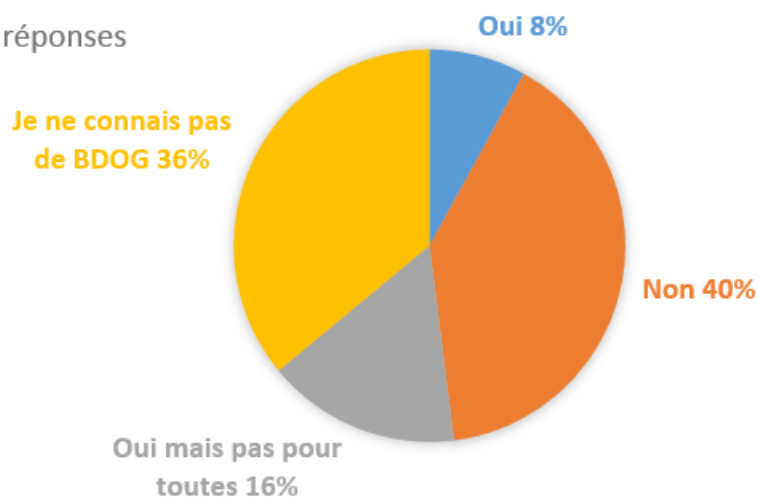


FIGURE 6.13 – Question 8) [Données filtrées] Seuls 24% des répondants connaissent la manière de faire interagir du code Java et des BDOG. Cette réponse nous conforte dans le développement d'une solution simple qui leur permettrait, grâce à Archipelago, d'accéder directement à la BDOG sans passer par l'apprentissage des bibliothèques spécifiques.

SERIEZ-VOUS INTÉRESSÉS PAR LE DÉVELOPPEMENT D'UN FRAMEWORK PERMETTANT D'UTILISER SIMPLEMENT UNE BDOG DANS UN CODE JAVA ?

25 réponses

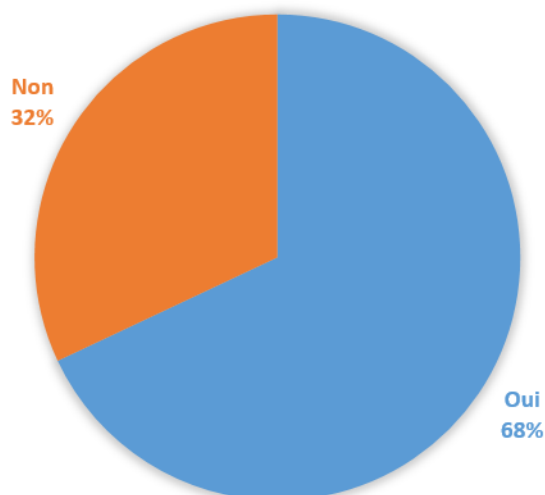


FIGURE 6.14 – Question 9) [Données filtrées] Cette réponse nous conforte à nouveau dans l'idée qu'une place est à prendre pour le développement d'Archipelago.

DANS LE CAS OÙ CE FRAMEWORK EXISTE ET EST PARFAITEMENT
FONCTIONNEL, ÊTES-VOUS PRÊT À ABANDONNER L'UTILISATION DES BASES
DE DONNÉES RELATIONNELLES ?

25 réponses

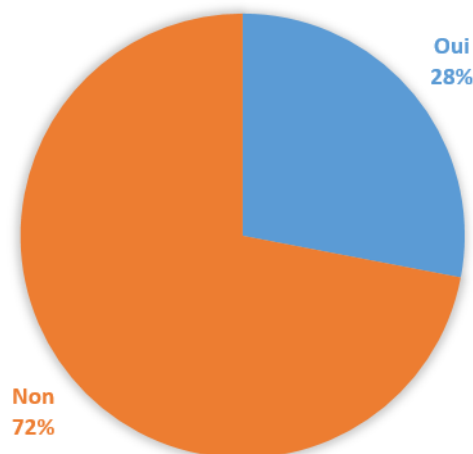


FIGURE 6.15 – Malgré la promesse d’un outil fiable, cette réponse témoigne essentiellement de la difficulté de changer les habitudes de développement.

6.3.2 CONTENU QUALITATIF

Nous avons décidé d’analyser la manière dont les répondants comprennent les bouts de codes utilisant le framework Archipelago. Nous ne leur avons fourni aucune documentation préalable. En légende des bouts de code ci-après, nous indiquerons le pourcentage de répondants ayant parfaitement compris la fonctionnalité que nous avons voulu préconiser (BR⁶), le pourcentage de personnes qui n’en ont pas saisi toutes les subtilités (MR⁷), et enfin le pourcentage de celles qui n’ont pas compris du tout (PC⁸).

En règle générale, nous avons été agréablement surpris par les bonnes réponses des répondants, nous pouvons supposer que l’utilisation du framework est simple car nous obtenons une moyenne de bonnes réponses de l’ordre de 78%. Le questionnaire se terminait par une possibilité de nous fournir un commentaire, nous permettant d’entrevoir plusieurs pistes d’amélioration.

6. **B**onne **R**éponse

7. **M**oyenne **R**éponse

8. **P**as **C**ompris

Comprenez-vous cette partie de code ? Si vous le pouvez, décrivez brièvement ce qu'il fait.

```
Archipelago a = Archipelago.getInstance();  
  
City winterfell = new City( name: "Winterfell");  
  
a.persist(winterfell);
```

FIGURE 6.16 – Bout de code permettant d’instancier un objet de type City et de le persister dans la BDOG :
BR : 80% MR : 4% PC : 16%

Comprenez-vous cette partie de code ? Si vous le pouvez, décrivez brièvement ce qu'il fait.

```
Archipelago a = Archipelago.getInstance();  
  
City winterfell = new City( name: "Winterfell");  
City kingsLanding = new City( name: "King's Landing");  
  
Road road= new Road();  
road.setDistance(5001);  
road.setUnit("leagues");  
  
a.link(kingsLanding,winterfell,road, biDirectional: true);
```

FIGURE 6.17 – Bout de code permettant de lier deux nœuds à l’aide d’un objet de type Road dans la BDOG :
BR : 84% MR : 0% PC : 16%

Comprenez-vous cette partie de code ? Si vous le pouvez, décrivez brièvement ce qu'il fait.

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

ArchipelagoQuery aq = a.getQueryBuilder()
    .of(City.class)
    .where(of("name", "King's Landing"), ConditionQualifier.EQUAL)
    .build();
List<Object> nodes = a.execute(aq);
```

FIGURE 6.18 – Bout de code permettant de faire une requête dans la bases de données en recherchant toutes les instances de villes se nommant « King's Landing » (Nous avons accepté les réponses désignant « La ville de King's Landing » comme étant une bonne réponse) :

BR : 76% MR : 4% PC : 20%

Comprenez-vous cette partie de code ? Si vous le pouvez, décrivez brièvement ce qu'il fait.

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

ArchipelagoQuery aq = a.getQueryBuilder()
    .of(City.class)
    .where(of("name", "King's Landing"), ConditionQualifier.EQUAL)
    .or(of("name", "Winterfell"), ConditionQualifier.EQUAL)
    .and(of("name", "Dragonstone"), ConditionQualifier.NOT_EQUAL)
    .build();
List<Object> nodes = a.execute(aq);
```

FIGURE 6.19 – Bout de code permettant de faire une requête dans la bases de données en recherchant toutes les instances de villes se nommant « King's Landing » OU « Winterfell » ET NON « Dragonstone » :

BR : 72% MR : 4% PC : 24%

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 peut être fortement interconnectées 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'ajouter ou d'ôter aisément de nouveaux paramètres, 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 utilisé pour développer ce framework offre une maintenance aisée car il permet de concentrer la recherche de problèmes dans un espace défini et bien cadré.

Les points à améliorer 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 de 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 faibles a 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.

Perspectives

Il nous semblait plus pertinent de scinder les perspectives d’améliorations de ce mémoire, des conclusions. A ce propos, nous dressons donc un ensemble de pistes auxquelles nous avons pensés mais que, pour diverses raisons, nous n’avons pu mettre en place actuellement. Ces éléments pourront contribuer au « cahier des charges » ou être une piste de réflexion pour une éventuellement deuxième version de ce framework.

8.1 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[25].

8.2 AMÉLIORATION DU SYSTÈME DE CONDITIONS

Pour améliorer le système de conditions, il est primordial de retravailler 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 questionnaire¹ que nous avons soumis, une idée de modification est apparue dans les suggestions et nous a particulièrement convaincue.

Le système pourrait être revu de la manière suivante :

```

1 // Instanciation du framework
2 Archipelago arch = Archipelago.getInstance();
3 // Création de la requête
4 ArchipelagoQuery aq = arch.getQueryBuilder()
5 // Spécification du type d'élément
6     .of(City.class)
7 // Création de condition

```

1. Mettre en place un système tel que réalisé par « Hibernate » avec l’implémentation de leur **Criterion**.


```

8      .where(
9  // Début de _1_ET_2_
10         and(
11 // Première partie du ET(1) et début de _1_OU_2_
12         or(
13 // Condition d'équivalence "name" = "King's Landing" dans OU(1)
14         eq("name", "King's Landing"),
15 // Condition d'équivalence "name" = "Winterfell" dans OU(2)
16         eq("name", "Winterfell")
17         ),
18 // Condition de non-équivalence "name" <> "Dragonstone" dans ET(2)
19         neq("name", "Dragonstone")
20         ),
21 // Condition de positionnement : Nœud de départ
22         ArchipelagoPosition.START
23     )
24 // Création de requête
25     .where(
26 // Début de _1_ET_2_
27         and(
28 // Condition de grandeur "distance" >= 300 dans ET(1)
29         gt("distance", 300),
30 // Condition d'équivalence "unit" = leagues" dans ET(2)
31         eq("unit", "leagues")
32         ),
33 // Condition de positionnement : relation
34         ArchipelagoPosition.EDGE
35     )
36 // Création de requête
37     .where(
38 // Condition d'équivalence "name" = Winterfell"
39         eq("name", "Winterfell"),
40 // Condition de positionnement : relation
41         ArchipelagoPosition.END
42     )
43 // Génération de la requête
44     .build();
45 // Exécution de la requête
46 List<Object> nodes = arch.execute(aq);

```

Avec le procédé décrit par les commentaires, 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.

Si le framework rendait possible l'utilisation d'un langage de condition simple avec une analyse de requête (parsing), cette deuxième solution permettrait quant à elle de simplifier grandement les conditions courantes. Une grammaire BNF² comme suit serait intéressante à implémenter :

2. <http://matt.might.net/articles/grammars-bnf-ebnf/>

```

1 condition ::=
2     token
3     | '(' token ('AND' | 'OR') token ')';
4 token ::= expr symbol value 'ON' position ;
5 position ::= ('START' | 'EDGE' | 'END') ;
6 symbol ::= ('=' | '!=' | '>' | '>=' | '<' | '<=') ;
7 expr ::= '"' [a-zA-Z0-9]+ '"';
8 value ::= '"' [a-zA-Z0-9]+ '"' | [0-9]+ ;

```

8.3 AMÉLIORATION DE L'EXPLOITATION DES RELATIONS

Un objet générique "Wrapper" tel que présenté dans la section 2.1, représentant un lien entre deux objets, peut être une piste d'amélioration de l'exploitation des relations. Comme la librairie de Neo4J 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 L'objet 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 LA THÉORIE DES GRAPHERS

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 »[10].

Exemple :

```

1 Archipelago arch = Archipelago.getInstance();
2
3 // Initialisation de la requête
4 ArchipelagoQuery aq = arch.getQueryBuilder()
5 // Spécification du type de départ
6     .from(City.class)
7 // Conditions voir exemples précédents
8     .where(
9         ... ,
10        ArchipelagoPosition.START
11    )

```

```

12 // Spécification du type d'arrivée
13     .to(City.class)
14 // Conditions voir exemples précédents
15     .where(
16         ...,
17         ArchipelagoPosition.END
18     )
19 // Spécification de l'algorithme de recherche de parcours
20     .with(ArchipelagoAlgo.DIJKSTRA)
21 // Génération de la requête
22     .build();
23 List<Object> nodes = arch.execute(aq);

```

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[11] peut être une fonctionnalité intéressante pour une seconde version de ce framework. En effet, si l'utilisateur a la possibilité d'employer ce genre d'algorithme pour résoudre ce type de problème, Archipelago répondra plus facilement aux attentes d'applications logistiques.

Bibliographie

- [1] JanusGraph Authors. Janusgraph official website. <http://janusgraph.org/>, 2017.
- [2] CCM Benchmark. Dictionnaire français. <http://www.linternaute.com/dictionnaire/fr/definition/archipel/>, 2017.
- [3] Rudi Bruchez. Les bases de données nosql et le big data. Eyrolles, 2016.
- [4] Roderic Geoffrey Galton Cattell, Douglas K Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. The object database standard : ODMG 2.0, volume 131. Morgan Kaufmann Publishers San Mateo, 1997.
- [5] Anthony Cleve, Anne-France Brogneaux, and Jean-Luc Hainaut. A conceptual approach to database applications evolution. In International Conference on Conceptual Modeling, pages 132–145. Springer, 2010.
- [6] Thierry Coupaye and Christine Collet. Modèles de comportement des sgbd actifs : caractérisation et comparaison. TSI. Technique et science informatiques, 17(3) :299–328, 1998.
- [7] 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.
- [8] Carlo A Curino, Letizia Tanca, Hyun J Moon, and Carlo Zaniolo. Schema evolution in wikipedia : toward a web information system benchmark. In In International Conference on Enterprise Information Systems (ICEIS). Citeseer, 2008.
- [9] Matteo Di Maglie. Adoption d’une solution NoSQL dans l’entreprise. PhD thesis, Haute école de gestion de Genève, 2012.
- [10] Edsger W Dijkstra. A note on two problems in connexion with graphs. Numerische mathematik, 1(1) :269–271, 1959.
- [11] 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.
- [12] Roy T Fielding and Richard N Taylor. Architectural styles and the design of network-based software architectures. University of California, Irvine Doctoral dissertation, 2000.
- [13] The Apache Software Foundation. Apache tinkerpops official website. <http://tinkerpops.apache.org/>, 2017.
- [14] Dmitrij Frishman, Klaus Heumann, Arthur Lesk, and Hans-Werner Mewes. Comprehensive, comprehensible, distributed and intelligent databases : current status. Bioinformatics (Oxford, England), 14(7) :551–561, 1998.
- [15] ArangoDB GmbH. Arrangodb official website. <https://www.arangodb.com/>, 2017.

- [16] Steve Lohr. The age of big data. New York Times, 11(2012), 2012.
- [17] OrientDB LTD. Orientdb official website. <https://orientdb.com/>, 2017.
- [18] Justin J Miller. Graph database applications and concepts with neo4j. In Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA, volume 2324, page 36, 2013.
- [19] Renée J Miller, Laura M Haas, and Mauricio A Hernández. Schema mapping as query discovery. In VLDB, volume 2000, pages 77–88, 2000.
- [20] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In vldb, volume 98, pages 24–27, 1998.
- [21] Cory Nance, Travis Losser, Reenu Iype, and Gary Harmon. Nosql vs rdbms-why there is room for both. 2013.
- [22] Inc Neo Technology. Neo4j official website. <https://neo4j.com/>, 2017.
- [23] LHRMB Niswonger, M Tork Roth, PM Schwarz, and EL Wimmers. Transforming heterogeneous data with database middleware : Beyond integration. Data Engineering, 31, 1999.
- [24] solid IT gmbh. Knowledge base of relational and nosql database management systems. <https://db-engines.com>, 2017.
- [25] Christine Solnon. Théorie des graphes et optimisation dans les graphes. INSA de Lyon.
- [26] Mohamed Tarroumi. Nosql (not only sql). <https://prezi.com/4flswlgipwbo/nosql-not-only-sql/>, 10 Décembre 2014.
- [27] OrientDB Team. Orientdb - getting started. <https://www.udemy.com/orientdb-getting-started>, 2017.
- [28] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database : a data provenance perspective. In Proceedings of the 48th annual Southeast regional conference, page 42. ACM, 2010.
- [29] Directeur France et Europe du Sud de MongoDB Yann Aubry. Pourquoi nosql s'impose face aux sgbd traditionnelles (avis d'expert). <http://www.silicon.fr/base-donnees-nosql-impose-sgbd-93305.html>, 19 mars 2014.

Troisième partie

Annexes

1. DIAGRAMME DE CLASSE « L'ÉCOLE »

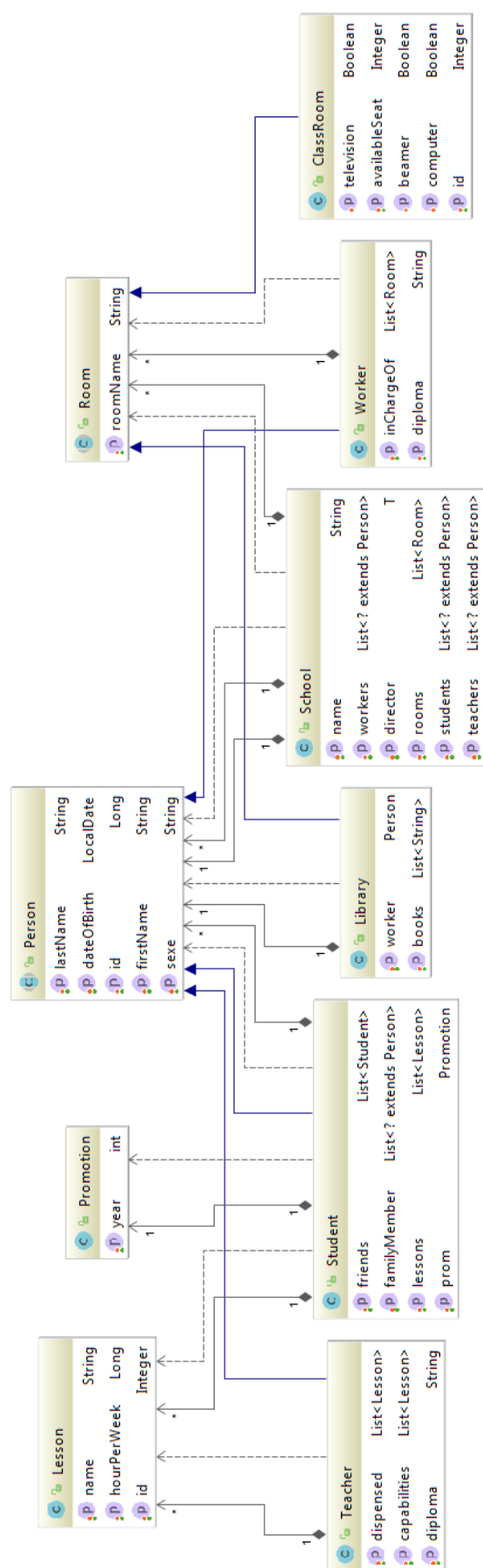


FIGURE 8.1 – Diagramme de Classe du modèle École.

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    Promotion p2011 = new Promotion(2011);
20    Promotion p2002 = new Promotion(2002);
21    Promotion p2010 = new Promotion(2010);
22    Teacher gys = new Teacher("Hans", "Gys", null, "M",
        Lists.newArrayList(dutch2, dutch4, english2, english4),
        Lists.newArrayList(dutch2, dutch4, english2, english4), "Master");
23    Teacher goffin = new Teacher("Michel", "Goffin", null, "M",
        Lists.newArrayList(math8, math6, math4), Lists.newArrayList(math8,
        math6, math4), "Master");
24    Teacher massart = new Teacher("Gabriel", "Massart", null, "M",
        Lists.newArrayList(math8, math6, math4), Lists.newArrayList(math8,
        math6, math4), "Master");
25    Teacher gouthers = new Teacher("Yves", "Gouthers", null, "M",
        Lists.newArrayList(frans5, frans6, religion),
        Lists.newArrayList(frans5, frans6, religion), "Master");
26    Teacher jacques = new Teacher("Christian", "Jacques", null, "M",
        Lists.newArrayList(geography), Lists.newArrayList(geography),
        "Master");
27    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);
28    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);
29    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);
30    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);
31 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);
32 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);
33 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);
34 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);
35 Worker cassart = new Worker("", "Cassart", null, "M", null, null);
36 List<Room> rooms;
37 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);
38 cassart.setInChargeOf(Lists.newArrayList(library));
39 Room l003 = new Classroom("L003", 30, true, false, true);
40 Room c203 = new Classroom("C203", 30, true, false, true);
41 Room l306 = new Classroom("L306", 50, true, true, false);
42 gilles.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,
    martin, benjamin, antoineBo));
43 gilles.setFamilyMember(Lists.newArrayList(antoineBo));
44 thomasB.setFriends(Lists.newArrayList(gilles, thomasR, charly, antoine,
    martin, benjamin));
45 thomasR.setFriends(Lists.newArrayList(thomasB, gilles, charly, antoine,
    martin, benjamin));
46 charly.setFriends(Lists.newArrayList(thomasB, thomasR, gilles, antoine,
    martin, benjamin));
47 antoine.setFriends(Lists.newArrayList(thomasB, thomasR, charly, gilles,
    martin, benjamin));
48 martin.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,
    gilles, benjamin));
49 benjamin.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,
    martin, gilles));
50 School school = new School();
51 school.setName("Saint Louis Namur");
52 school.setDirector(gys);
53 school.setTeachers(Lists.newArrayList(goffin, gouthers, jacques, gys,
    massart));
54 school.setStudents(Lists.newArrayList(gilles, thomasB, thomasR, charly,
    antoine, martin, benjamin, antoineBo));
55 school.setRooms(Lists.newArrayList(library, l003, l306, c203));
56 school.setWorkers(Lists.newArrayList(cassart));
57 return school;
58 }

```

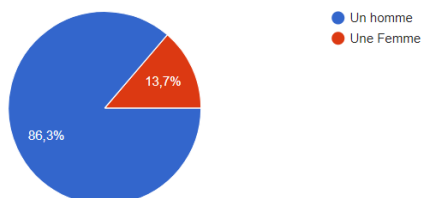
LES RÉSULTATS DU QUESTIONNAIRE

Les résultats non filtrés du questionnaire se trouvent dans un fichier CSV (Archipelago Questionnaire.csv), inclus dans la sauvegarde numérique de ce mémoire.

Voici un export brut des images d'analyses générées par Google© Form :

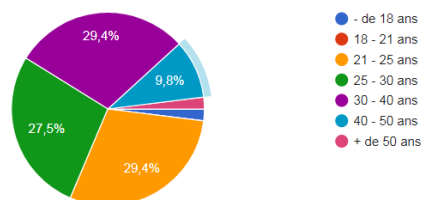
Etes-vous ?

51 réponses



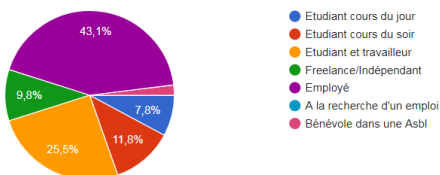
Dans quelle tranche d'âge vous situez-vous ?

51 réponses



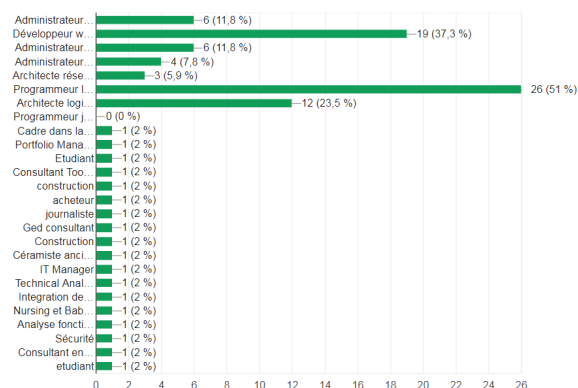
Etes-vous ?

51 réponses



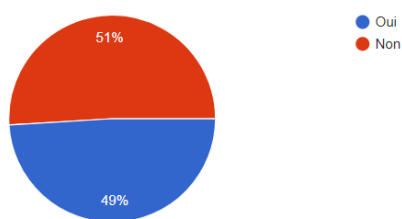
Dans quel(s) domaine(s) travaillez-vous ?

51 réponses



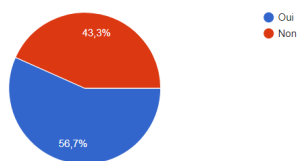
Connaissez-vous l'existence des BDOG ?

51 réponses



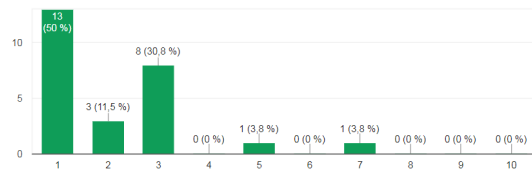
- Si vous avez répondu "Non" à la question précédente : Avez-vous l'intention de vous y intéresser ?

30 réponses



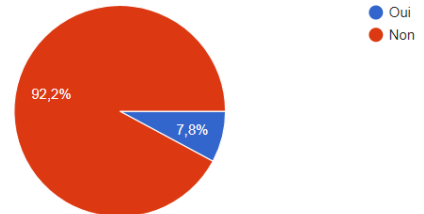
- Si vous aviez répondu "Oui" : Combien de BDOG connaissez-vous ?

26 réponses



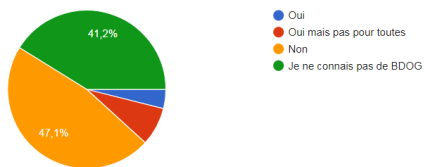
Avez-vous déjà travaillé avec une BDOG ?

51 réponses



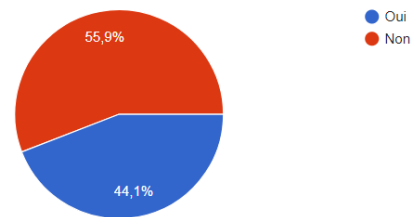
Connaissez-vous la manière de faire interagir du code Java et cette/ces BDOG ?

51 réponses



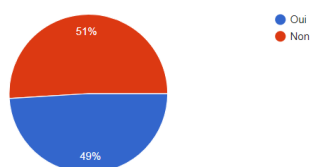
Avez-vous l'habitude de développer en Java ?

34 réponses



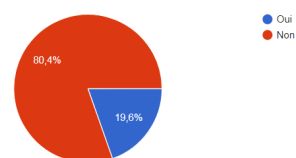
Seriez-vous intéressés par le développement d'un framework permettant d'utiliser simplement une BDOG dans un code Java ?

51 réponses



Dans le cas où ce framework existe et est parfaitement fonctionnel, êtes-vous prêt à abandonner l'utilisation des bases de données relationnelles ?

51 réponses



Code Sources

Par souci d'écologie et par praticité, notre code est disponible sur le dépôt GitHub (cf. lien ci-dessous) ainsi que sur la copie numérique de ce mémoire.

<https://github.com/GillesBodart/Archipelago>.