Université de Namur

Faculté d’informatique

Année académique 2017–2018

**Archipelago : Un framework de persistance**

**pour bases de données orientées graphe.**

Gilles Bodart



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

CLEVE Anthony

Co-promoteur : LAMBIOTTE Renaud

Mémoire présenté en vue de l’obtention du grade de

Master en Sciences Informatiques.

**Table des matières**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **I** | **État de l’art** | | | **5** |
| **1** | **L’évolution du NoSql** | | | **6** |
|  | 1.1 | Questionnement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 7 |
| **2** | **Backgroud technique** | | | **9** |
|  | 2.1 | Neo4J . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 9 |
|  |  | 2.1.1 | Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 9 |
|  |  | 2.1.2 | Langage de requête . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 9 |
|  |  | 2.1.3 | Communication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 10 |
|  | 2.2 | OrientDB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 10 |
|  |  | 2.2.1 | Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 10 |
|  |  | 2.2.2 | Langage de requête . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 11 |
|  |  | 2.2.3 | Communication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 12 |
| **3** | **Solutions existantes** | | | **13** |
|  | 3.1 | Les librairies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 13 |
|  |  | 3.1.1 | Hibernate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 13 |
|  |  | 3.1.2 | Spring Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 13 |
|  |  | 3.1.3 | Librairie Neo4J . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 14 |
|  |  | 3.1.4 | Librairie OrientDB . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 15 |
|  | 3.2 | Les frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 15 |
|  |  | 3.2.1 | Apache TinkerPop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 15 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **II** | **Contribution** | | | **17** |
| **4 Applications possibles des BDOG** | | | | **18** |
|  | 4.1 | L’exemple parfait . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 18 |
|  | 4.2 | Critères de comparaison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 18 |
|  | 4.3 | Comparaison des plus grandes BDOG . . . . . . . . . . . . . . . . . . . . . . . . | | 19 |
|  | 4.4 | Piste de normalisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 19 |
| **5** | **Le framework** | | | **20** |
|  | 5.1 | Utilisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 20 |
|  |  | 5.1.1 | La configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 20 |
|  |  | 5.1.2 | La persistance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 20 |
|  |  | 5.1.3 | La liaison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 23 |
|  |  | 5.1.4 | La récupération . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 24 |
|  | 5.2 | Schéma conceptuel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 25 |
|  |  | 5.2.1 | Archipelago . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 27 |

2

|  |  |  |  |
| --- | --- | --- | --- |
| **6 Evaluation** | |  | **29** |
| 6.1 | Points forts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 29 |
|  | 6.1.1 | La facilité . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 29 |
|  | 6.1.2 | La configurabilité . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 29 |
|  | 6.1.3 | L’abstraction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 29 |
|  | 6.1.4 | La simplification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 30 |
|  | 6.1.5 | La modularité . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 30 |
| 6.2 | Points faibles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 30 |
|  | 6.2.1 | La Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 30 |
|  | 6.2.2 | Les conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 30 |
|  | 6.2.3 | La récupération d’informations . . . . . . . . . . . . . . . . . . . . . . . | 32 |
|  | 6.2.4 | La propreté du code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 33 |

|  |  |  |
| --- | --- | --- |
| **7 Conclusion** |  | **35** |
| 7.1 Piste de réflexions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 35 |
| 7.1.1 | Amélioration du système de conditions . . . . . . . . . . . . . . . . . . . | 35 |
| 7.1.2 | Amélioration des performances . . . . . . . . . . . . . . . . . . . . . . . | 35 |
| 7.1.3 | Amélioration de l’exploitation des relations . . . . . . . . . . . . . . . . . | 35 |
| 7.1.4 | Ajout d’algorithmes célèbres de théorie des graphes . . . . . . . . . . . . | 35 |
| 7.1.5 | Standardisation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 35 |

|  |  |  |
| --- | --- | --- |
| 7.2 Archipelago en résumé . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 35 |
| **III** | **Annexes** | **36** |
| **IV** | **Glossaire** | **41** |
| **V** | **Code Sources** | **43** |
| **VI** | **Bibliographie** | **44** |

3

**Introduction**

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

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

A contrario, le mouvement NoSql pour "Not Only SQL" ne se base pas sur un fondement théorique commun. Que ce soient la théorie des graphes, les systèmes de hash key/value avec partitions, ... chaque système choisit ce sur quoi il va se baser. Mais s’il existait une sorte de contrat liant les SGBD NoSql entre eux afin d’assurer une cohérence d’utilisation, ce mémoire va se concentrer sur les implémentations de la théorie des graphe, en essayant de mettre en place un système de comparaison, nous allons tenter de spécifier certains objectifs communs et développer un framework Java permettant d’utiliser de manière agnostique l’une ou l’autre base de donnée.

4

**Première partie**

**État de l’art**

5

**Chapitre 1**

**L’évolution du NoSql**

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

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

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

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

Le NoSql découle de ce genre de problèmes, ces modèles arrivent avec des approches optimisées pour des secteurs spécifiques.

Comme ces modèles représentent ce qui n’est pas relationnel, par souci de classification, nous allons distinguer 4 usages principaux :

— Performances : L’objectif du SGBD sera d’augmenter au maximum les performances de la manipulation des données.

— Structures simples : Pour s’aﬀranchir de la rigidité du modèle relationnel, la structure sera généralement simplifiée, en utilisant une représentation plus souple comme le JSON par exemple.

— Structures spécifiques : Certains moteurs NoSql sont liés à des besoins spécifiques. La

structure de représentation de données sera dès lors focalisée sur un cas d’utilisation.

— Volumétries : Un des principaux aspects importants des SGBD NoSql est leur capacité de gérer la montée en charge de données. La distribution des traitements au travers de plusieurs clusters est un facteur essentiel dans la plupart des applications BigData.

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

6

— Document : L’utilisation de formats spécifiques tels que le très répandu "JSON" permet de stocker les données sur base de fichier.

— Clé / Valeur : Le système le plus simple, il manipule des paires de clé/valeur, ou accède à un élément en fonction d’une table de hachage.

— Colonne : Inspiré de Google BigTable, la structure ressemble à la table relationnelle. On

peut la comparer à une table de hachage qui va référencer une ou plusieurs colonnes.

— Graphe : La famille Graphe se distingue du fait que les entités ne sont pas considéré comme des entités indépendantes, mais que la relation entre ces objets est tout aussi importante que le contenu.

Les implémentations de bases de données de types graphe sont de plus en plus nombreuses, les relations entre les éléments permettent de parcourir le graphe de manière très performante les rendant de plus en plus intéressantes pour les entreprises possédant des millions de données. L’utilisation de ce genre de SGBD est dès lors tout à fait recommandée pour des entreprises intéressées entre les relations de ces données telles que des profils sociaux, des liens de cause à eﬀet, des liens géographiques et bien d’autres encore.[4][13][3]

1.1 Questionnement

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

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

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

En relationnel, il existe de nombreux moyens d’historiser les données, cependant, lorsque le schéma comprend des relations cycliques, cette opération devient plus complexe. L’utilisation d’un BDOG rend la démarche plus simple, il suﬃt de créer un nouveau nœud avec les anciennes données, lier ce nœud avec une relation historique en y spécifiant la date de mise à jour, et ensuite de changer les valeurs du nœud référencé par le cycle. Ce procédé peut être mis au point dans n’importe quelle représentation et ne nécessite aucune refonte globale du modèle de données.[15]

7

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

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

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

8

**Chapitre 2**

**Backgroud technique**

2.1 Neo4J

2.1.1 Description

Créé par Neo Technologie, une société américaine et suédoise, elle est actuellement la base de donnée orientée graphe, la plus utilisée dans le monde[12]. Développé en java sous licence GPL V3, AGPL ou licence commerciale, Neo4J représente les données sous forme de "Nœuds" et de "Relations". Chacun de ces éléments peuvent contenir une ou plusieurs propriétés. Les propriétés sont des couples clés/valeurs de type simple, comme des chaines de caractères ou des valeurs numériques, des coordonnées spatiales, ... [11]

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

2.1.2 Langage de requête

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

* CREATE (mamours:Person {name:"Mamours"})

2 CREATE (mamyco:Person {name:"Mamyco"})

3 CREATE (gilles:Person {name:"Gilles"})

4 CREATE (marie:Person {name:"Marie"})

5 CREATE (bebe:Person {name:"Enfant"})

6 CREATE (mamours)-[:PARENT\_OF]->(gilles)

7 CREATE (mamyco)-[:PARENT\_OF]->(marie)

8 CREATE (marie)-[:PARENT\_OF]->(bebe)

9 CREATE (gilles)-[:PARENT\_OF]->(bebe)

Ces requêtes vont créer 5 nœuds et 4 relations PARENT\_OF, nous pouvons aisément com-prendre que "Mamyco" est parente de "Marie"

9

* MATCH (p:Person)-[:PARENT\_OF]->(c:Person)

2 RETURN DISTINCT (p)

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

* MATCH (gp:Person)-[:PARENT\_OF\*2]->(c:Person)

2 RETURN DISTINCT (gp)

Celle-ci quant à elle va retourner toutes les personnes qui sont parent de parent et donc grand-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 né-cessiteraient une double jointure sur la Table "Person".

2.1.3 Communication

Neo4J peut être utilisé sous plusieurs formes.

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

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

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

2.2 OrientDB

2.2.1 Description

OrientDB est un SGBD initialement développé en C++ (Orient ODBMS), repris ensuite en 2010 en Java par Luca Garulli dans une version multi-modèle sous licence Apache 2.0, GPL et AGPL. Actuellement 3ème mondial (selon db-engines.com) il oﬀre de nombreuses fonctionna-lités intéressantes.[10]

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

Ce SGBD est composé de trois grands éléments

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

10

relationnelles.

— Links & Edge : Une arrête orientée reliant deux éléments non nécessairement distincts.

— Property : Typée ou embarquée dans un document JSON, ceci va représenter le contenu de l’information. Ces propriétés sont bien entendu primordiales pour ordonner, rechercher,

...

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

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

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

2.2.2 Langage de requête

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

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

* 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")

* ou
* 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.

* SELECT FROM V

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

11

Nous pouvons observer qu’OrientDB se charge de qualifier les documents de plusieurs métadata en leur octroyant par exemple un identifiant unique (no cluster :position), un numéro de version pour permettre une gestion de transaction "full optimistic" 1 et le class représente la structure du document.

* 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.[14]

2.2.3 Communication

OrientDB embarque une API Rest complète[6], toute action pouvant être faite sur les interfaces web et consoles, peuvent être reproduites au travers de cette API.

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

12

**Chapitre 3**

**Solutions existantes**

3.1 Les librairies

TODO livre

3.1.1 Hibernate

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

— @OneToOne : Relation 1-1

— @ManyToOne : Relation n-1

— @OneToMany : Relation 1-n

— @ManyToMany : Relation n-n

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

3.1.2 Spring Data

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

Exemple :

* @NodeEntity(label="Film")

2 public class Movie {

3

1. Object Graph Mapper

13

* @GraphId Long id;

5

* @Property(name="title")

7private String name;

* }

9

1. @NodeEntity
2. public class Actor extends DomainObject {

12

1. @GraphId
2. private Long id;

15

1. @Property(name="name")
2. private String fullName;

18

1. @Relationship(type="ACTED\_IN", direction=Relationship.OUTGOING)
2. private List<Role> filmography;

21

1. }

23

1. @RelationshipEntity(type="ACTED\_IN")
2. public class Role {
3. @GraphId
4. private Long relationshipId;
5. @Property
6. private String title;
7. @StartNode
8. private Actor actor;
9. @EndNode
10. private Movie movie;
11. }

3.1.3 Librairie Neo4J

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

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

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

5 driver.close();

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

14

3.1.4 Librairie OrientDB

La librairie OrientDB pour java est composée de trois éléments :

— graph API

— document API

— object API

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

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

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

3.2 Les frameworks

3.2.1 Apache TinkerPop

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

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

Les développeurs de ce framework ont mis au point un langage de travail sur les graphes appelé "Gremlin". Le langage dont nous avons parlé en section 2.1.2 s’est grandement inspiré de ce dernier.[7]

Exemple d’utilisation

* 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]

* 1. 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"

15

* gremlin> g.V(1) /\* récupère le nœud d’Id 1 du graphe\*/

9 ==>v[1]

1. gremlin> g.V(1).values(’name’) /\* récupère le nœud d’Id 1 du graphe et renvoie sa propriété ’name’ \*/
2. ==>marko
3. gremlin> g.V(1).outE(’knows’) /\* récupère les relations ’knows’ partant du noeud d’Id 1 \*/
4. ==>e[7][1-knows->2]
5. ==>e[8][1-knows->4]
6. gremlin> g.V(1).outE(’knows’).inV().values(’name’) /\* retourne le nom des personnes que le nœud 1 connait \*/
7. ==>vadas
8. ==>josh
9. gremlin> g.V(1).out(’knows’).values(’name’) /\* même résultat \*/
10. ==>vadas
11. ==>josh
12. gremlin> g.V(1).out(’knows’).has(’age’, gt(30)).values(’name’) /\* retourne le nom des personnes âgées de plus de 30 ans que le nœud 1 connait \*/
13. ==>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.

16

**Deuxième partie**

**Contribution**

17

**Chapitre 4**

**Applications possibles des BDOG**

4.1 L’exemple parfait

Pour la suite de ce mémoire, il est primordial d’avoir un exemple concret de modèle habituelle-ment 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 (7.1).

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

4.2 Critères de comparaison

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

18

4.3 Comparaison des plus grandes BDOG

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

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

4.4 Piste de normalisation

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

— Utilisation d’un langage unique

— Système de transaction

— Opération de base de persistance CRUD

— Possibilité de créer des méthodes stockées

— Contrôle sur les propriétés.

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

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

19

**Chapitre 5**

**Le framework**

5.1 Utilisation

5.1.1 La configuration

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

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

Voici le canevas de ce fichier :

* {
* "database": {
* "type": "", // Choix entre NEO4J et ORIENT\_DB

4"username": "",

5"password": "",

6"url": "",

7"name": "", // Nom de la base de donnée

8"embedded": false, // boolean Optionnel

9"port": 1234 //

1. },
2. "deepness": 3, // Profondeur de persistance voir ci-après
3. "domainRootPackage": "org.archipelago.test.domain.school"
4. // package où se trouvent les classes du domaine
5. }

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

5.1.2 La persistance

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

20

orientée objet.

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

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

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

* Archipelago arch = Archipelago.getInstance();

2

* Student gilles = new Student();

4 Student antoine = new Student();

5

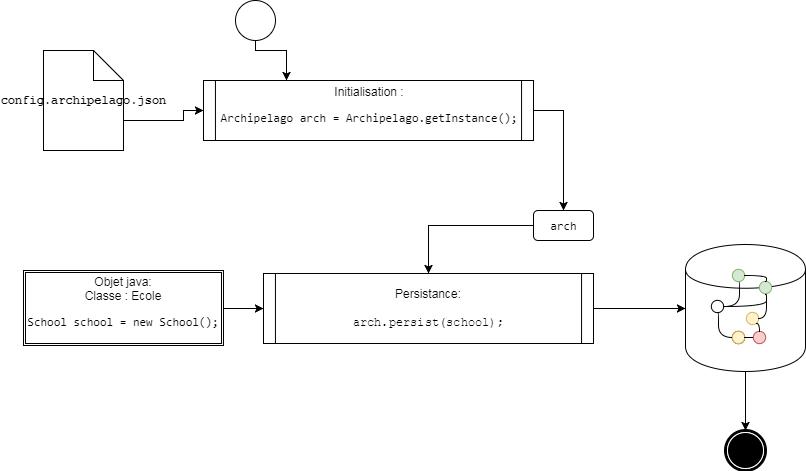
* gilles.setFriend(antoine);

7 antoine.setFriend(gilles);

8

* arch.persist(gilles);

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



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

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

Exemple avec la Classe *Student.java*

21

* public class Student extends Person {

2

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

5

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

8

* @Bridge(descriptor = "FamilyMember", biDirectionnal = true)

1. private List<? extends Person> familyMember = new ArrayList<>();

11

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

14

1. ...

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

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

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

* Archipelago arch = Archipelago.getInstance();

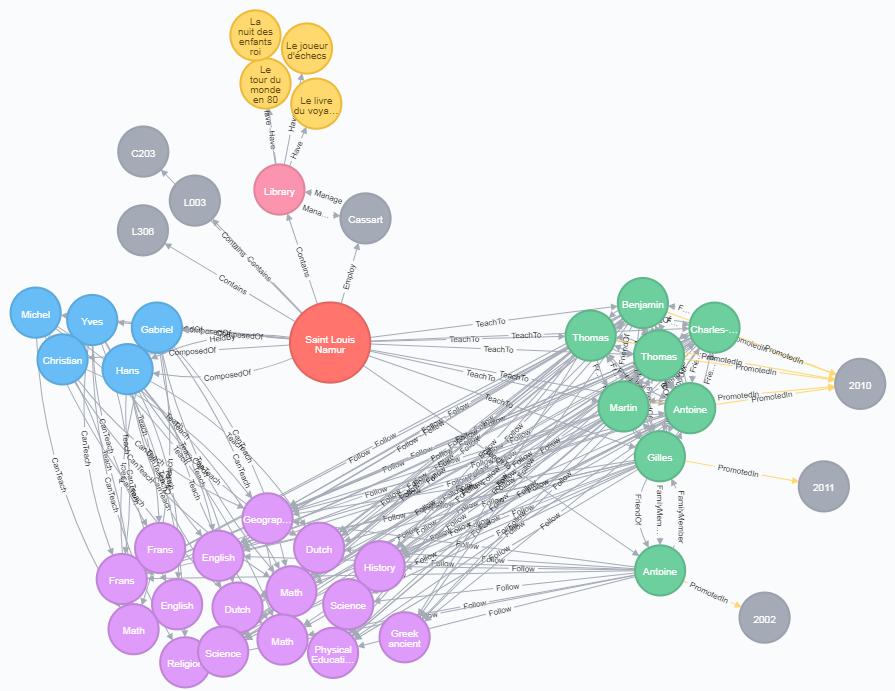
2

* School school = genSchool()

4

* arch.persist(gilles);

22



5.1.3 La liaison

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

Voici un exemple :

1

* Archipelago arch = Archipelago.getInstance();

3

* City winterfell = new City("Winterfell");
* City kingsLanding = new City("King’s Landing");

6

* Road road= new Road();

8 road.setDistance(500l);

9 road.setUnit("leagues");

10

1. arch.link(winterfell, kingsLanding, road);

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



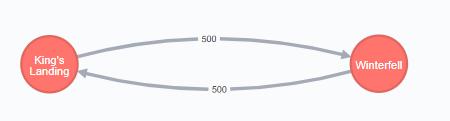
Toutefois si nous le spécifions à l’aide de la valeur "true" , le quatrième attribut de la méthode

23

nous permet de générer automatiquement une relation réciproque ou bidirectionnelle.

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

3 ...



5.1.4 La récupération

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

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

Voici un exemple :

* Archipelago arch = Archipelago.getInstance();

2

* ArchipelagoQuery aq = a.getQueryBuilder()

|  |  |
| --- | --- |
| 4 | .of(Student.class) |
| 5 | .build(); |

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

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

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

* Archipelago arch = Archipelago.getInstance();

2

* ArchipelagoQuery aq = a.getQueryBuilder()

|  |  |
| --- | --- |
| 4 | .of(Student.class) |
| 5 | .where(of("firstName","Gilles"), ConditionQualifier.Equal) |
| 6 | .build(); |

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

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

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

* Archipelago arch = Archipelago.getInstance();

2

* ArchipelagoQuery aq = a.getQueryBuilder()

4 .of(Student.class)

24

|  |  |
| --- | --- |
| 5 | .where(of("lastName", "Bodart"), ConditionQualifier.EQUAL) |
| 6 | .and(of("firstName", "Gilles"), ConditionQualifier.EQUAL) |
| 7 | .or(of("firstName", "Thomas"), ConditionQualifier.EQUAL) |
| 8 | .build(); |

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

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

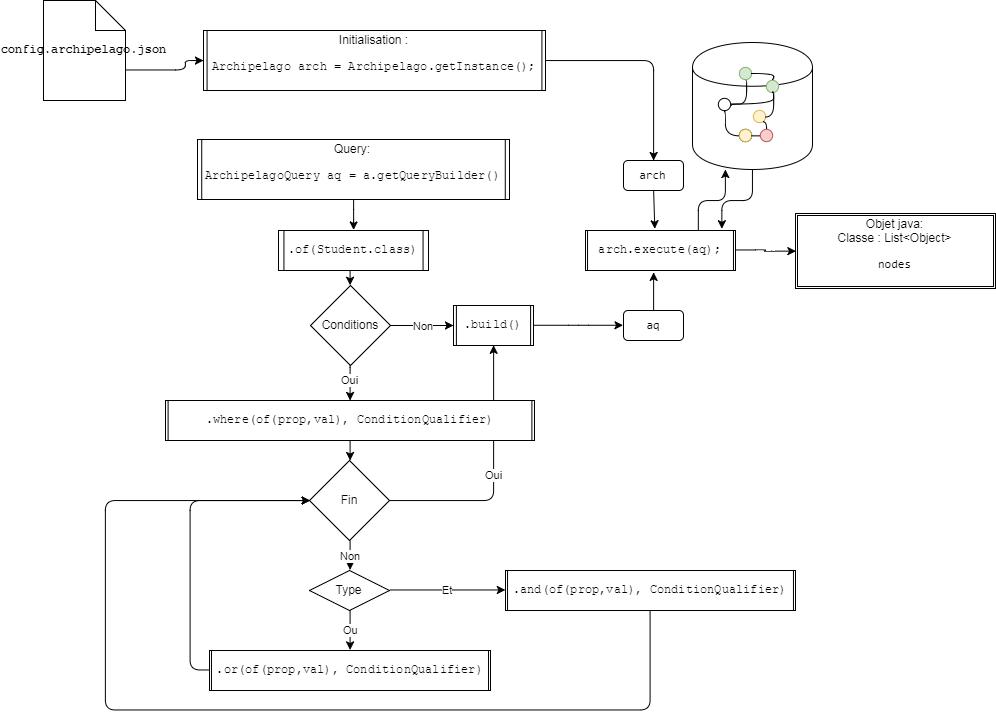
Pour l’exemple précédent, nous aurons donc :

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

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

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

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



5.2 Schéma conceptuel

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

25

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

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

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

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

Voici la démarche poursuivie :

26

5.2.1 Archipelago

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

— La persistance

— La liaison

— La recherche

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

Pour ces opérations, nous allons avoir besoin d’utilitaires nous permettant de générer des re-quêtes de persistance, de liaison et de recherche, pour chaque BDOG. Ces diﬀérentes implé-mentations 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 diﬀérentiation entre les requêtes qui créent de l’infor-mation et celles qui en récupèrent.

ScriptBuilders



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

**Nom de méthode Description**

Cette méthode doit générer une requête permettant de créer un

makeCreate

nœud.

Cette méthode doit générer une requête permettant de récupérer

makeMatch

l’identifiant correspondant à un objet.

Cette méthode doit générer une requête permettant de lier deux

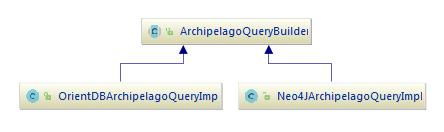
makeRelation

nœuds.

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

27

QueryBuilders



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

En voici le détail :

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

28

**Chapitre 6**

**Evaluation**

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

6.1 Points forts

6.1.1 La facilité

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

6.1.2 La configurabilité

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

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

6.1.3 L’abstraction

L’utilisateur ne doit se soucier d’aucune mise en oeuvre spécifique à la base de données ci-blée. Archipelago, grâce à ces implémentations concrètes cachées derrière les classes abstraites

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

29

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 les bases de données relationnelles 2 ; les frameworks de communication avec ces dernières étant plus courantes et mieux connues dans l’univers de la programmation.

6.1.5 La modularité

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

Ce genre d’architecture oﬀre une maintenance aisée car elle permet de concentrer la recherche de problèmes dans un espace défini et bien cadré. De plus, si deux équipes sont aﬀectées simultanément à ce framework, elles peuvent y ajouter de nouvelles destinations sans se gêner, telles que par exemple ArangoDB[8] ou encore JanusGraph[2].

6.2 Points faibles

6.2.1 La Performance

Bien que nous avons 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 arrêtes. Dans notre exemple de l’école, la partie reprenant les étudiants est fortement inter-connectée. En conséquence nous devons passer à de nombreuses reprises sur les mêmes arrêtes du graphe pour vérifier si cette relation existe dans la base de données au lieu de retenir l’information et d’éviter la démarche. Ces nombreux passages ralentissent le processus 3.

6.2.2 Les conditions

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

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

30

**Sous-conditions**

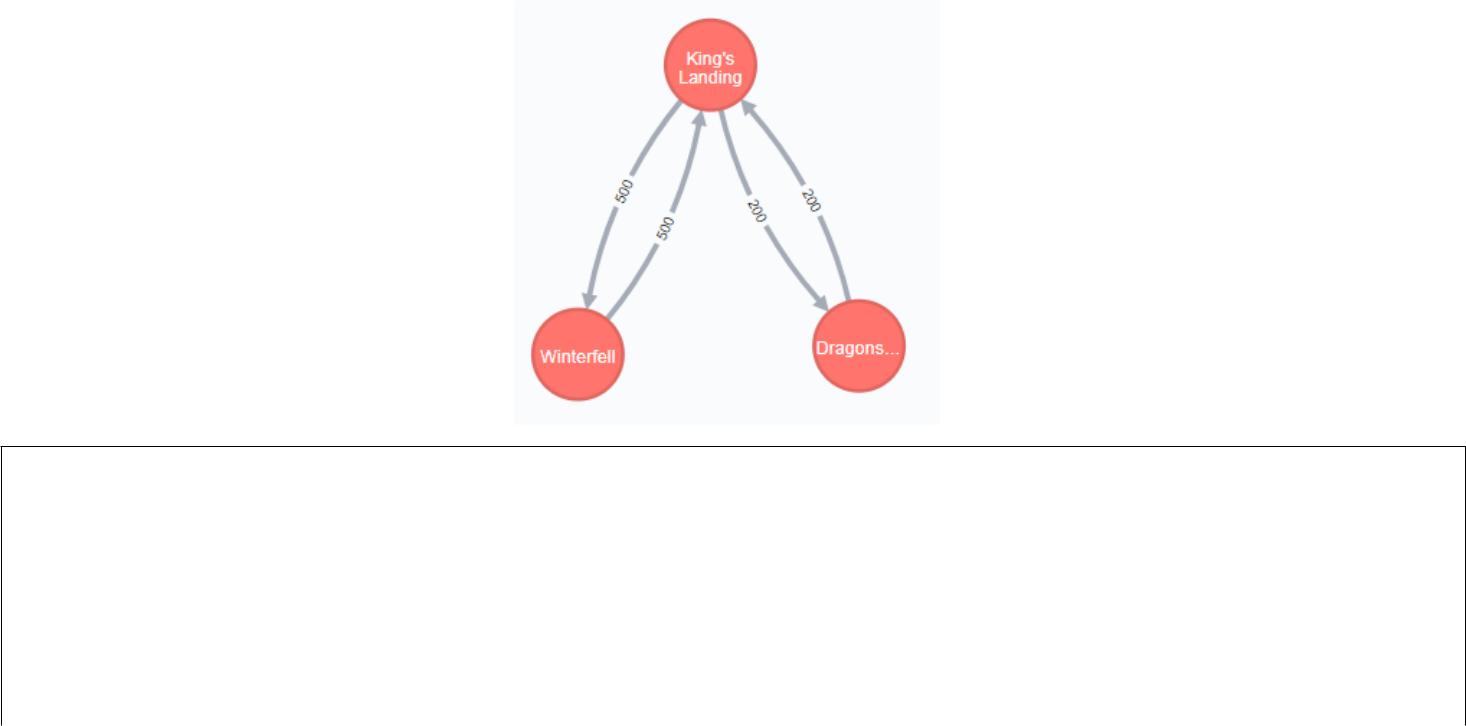
Comme nous l’avons évoqué dans le point 5.1.4, nous pouvons actuellement générer des condi-tions simples. Pour créer des conditions plus complexes, il faudrait eﬀectuer une refonte du système d’imbrication afin permettre à l’utilisateur de créer des sous-conditions.

Exemple de condition impossible à générer actuellement :

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

**Conditions sur les relations**

De plus il n’est pas possible de faire une condition sur une relation entre deux nœuds. Par exemple nous pourrions avoir grâce à la requête Cypher suivante :



* MATCH (d:City)-[rel:Road]->(a:City)

2 WHERE

|  |  |
| --- | --- |
| 3 | rel.distance > 300 |

* 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 pourrons pas nous limiter à celles situées à plus de 300 lieues.

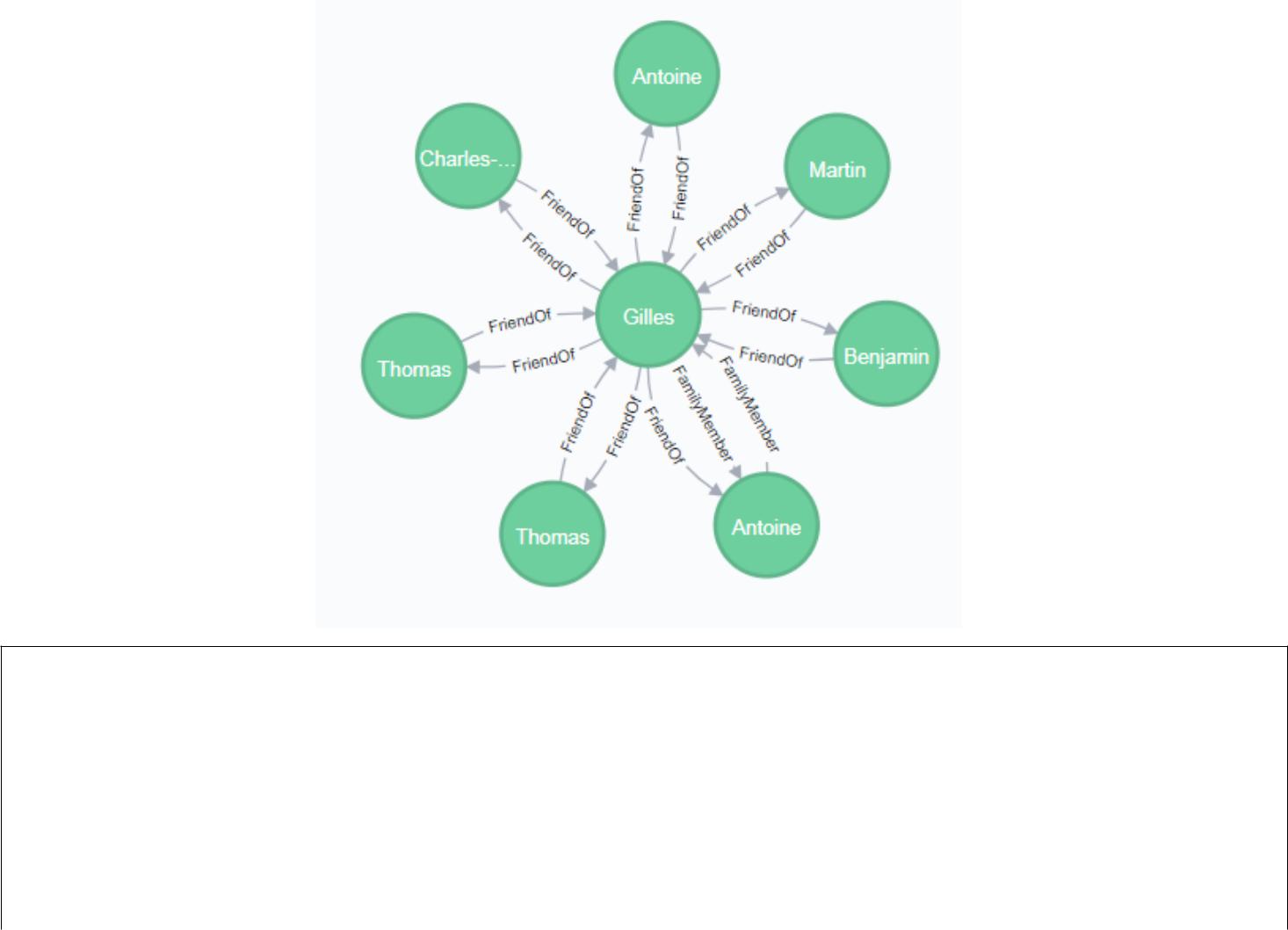
**Conditions sur les éléments liés**

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

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

31

Voici un exemple en Cypher :



* MATCH (me:Student)-[:FriendOf]->(f:Student)

2 WHERE

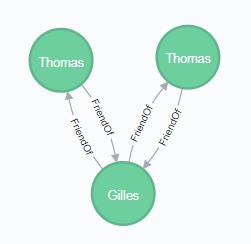
|  |  |
| --- | --- |
| 3 | me.firstName = "Gilles" |

* AND me.lastName = "Bodart"

5AND f.firstName = "Thomas"

6 RETURN (me)-[]-(f)

7 // lie-moi sont mes amis qui se prénomment Thomas?



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

6.2.3 La récupération d’informations

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

32

24

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

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

6.2.4 La propreté du code

Cette implémentation de ce framework pourrait être améliorée si l’on disposait de temps sup-plémentaire 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.

33

21

22

23

* public class City {

2

* private String name;

4

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

7

* ...
* 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

* Road road= new Road();

7 road.setDistance(500l);

8 road.setUnit("leagues");

9

1. Road road2= new Road();
2. road2.setDistance(200l);
3. road2.setUnit("leagues");

13

1. a.persist(winterfell);
2. a.persist(kingsLanding);
3. a.persist(dragonStone);
4. a.link(kingsLanding,winterfell,road, true);
5. a.link(kingsLanding,dragonStone,road2, true);

19

1. ArchipelagoQuery aq = a.getQueryBuilder()

.of(City.class)

.where(of("name","King’s Landing"), ConditionQualifier.EQUAL)

.build();

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

1. https ://github.com/GillesBodart/Archipelago
2. Nous n’en avons pas encore reçu.

34

**Chapitre 7**

**Conclusion**

7.1 Piste de réflexions

7.1.1 Amélioration du système de conditions

7.1.2 Amélioration des performances

7.1.3 Amélioration de l’exploitation des relations

7.1.4 Ajout d’algorithmes célèbres de théorie des graphes

7.1.5 Standardisation

bd orienté réseaux

7.2 Archipelago en résumé

Le mémoire sera conclu avec une explication transversale et complète du framework, permettant au lecteur de garder une bonne impression sur ce nouvel outil.

35

**Troisième partie**

**Annexes**

36

1. diagramme de classe "l’école"
2. Création des objets composant l’école

* private static School genSchool() throws ClassNotFoundException, IOException {

2

* Lesson math8 = new Lesson("Math", 8l);

4Lesson math6 = new Lesson("Math", 6l);

5Lesson math4 = new Lesson("Math", 4l);

6Lesson science6 = new Lesson("Science", 6l);

7Lesson science3 = new Lesson("Science", 3l);

8Lesson frans5 = new Lesson("Frans", 5l);

9Lesson frans6 = new Lesson("Frans", 6l);

1. Lesson dutch2 = new Lesson("Dutch", 2l);
2. Lesson dutch4 = new Lesson("Dutch", 4l);
3. Lesson english2 = new Lesson("English", 2l);
4. Lesson english4 = new Lesson("English", 4l);
5. Lesson history = new Lesson("History", 2l);
6. Lesson geography = new Lesson("Geography", 2l);
7. Lesson religion = new Lesson("Religion", 2l);
8. Lesson pE = new Lesson("Physical Education", 2l);
9. Lesson greek = new Lesson("Greek ancient", 4l);

19

1. Promotion p2011 = new Promotion(2011);
2. Promotion p2002 = new Promotion(2002);
3. Promotion p2010 = new Promotion(2010);

23

1. Teacher gys = new Teacher("Hans", "Gys", null, "M", Lists.newArrayList(dutch2, dutch4, english2, english4), Lists.newArrayList(dutch2, dutch4,english2, english4), "Master");
2. Teacher goffin = new Teacher("Michel", "Goffin", null, "M", Lists.newArrayList(math8, math6, math4), Lists.newArrayList(math8, math6, math4), "Master");
3. Teacher massart = new Teacher("Gabriel", "Massart", null, "M", Lists.newArrayList(math8, math6, math4), Lists.newArrayList(math8, math6, math4), "Master");
4. Teacher gouthers = new Teacher("Yves", "Gouthers", null, "M", Lists.newArrayList(frans5, frans6, religion), Lists.newArrayList(frans5, frans6, religion), "Master");
5. Teacher jacques = new Teacher("Christian", "Jacques", null, "M",

Lists.newArrayList(geography), Lists.newArrayList(geography), "Master");

29

1. 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);
2. 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);

37

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

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);

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);

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);

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);

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);

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);

Worker cassart = new Worker("", "Cassart", null, "M", null, null); List<Room> rooms;

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);

cassart.setInChargeOf(Lists.newArrayList(library));

Room l003 = new ClassRoom("L003", 30, true, false, true);

Room c203 = new ClassRoom("C203", 30, true, false, true);

Room l306 = new ClassRoom("L306", 50, true, true, false);

gilles.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine, martin, benjamin, antoineBo));

gilles.setFamilyMember(Lists.newArrayList(antoineBo)); thomasB.setFriends(Lists.newArrayList(gilles, thomasR, charly, antoine,

martin, benjamin));

thomasR.setFriends(Lists.newArrayList(thomasB, gilles, charly, antoine,

martin, benjamin));

charly.setFriends(Lists.newArrayList(thomasB, thomasR, gilles, antoine,

martin, benjamin));

antoine.setFriends(Lists.newArrayList(thomasB, thomasR, charly, gilles,

martin, benjamin));

martin.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,

gilles, benjamin));

benjamin.setFriends(Lists.newArrayList(thomasB, thomasR, charly, antoine,

martin, gilles));

School school = new School();

school.setName("Saint Louis Namur");

school.setDirector(gys);

school.setTeachers(Lists.newArrayList(goffin, gouthers, jacques, gys,

massart));

38

59

60

61

62

63

64

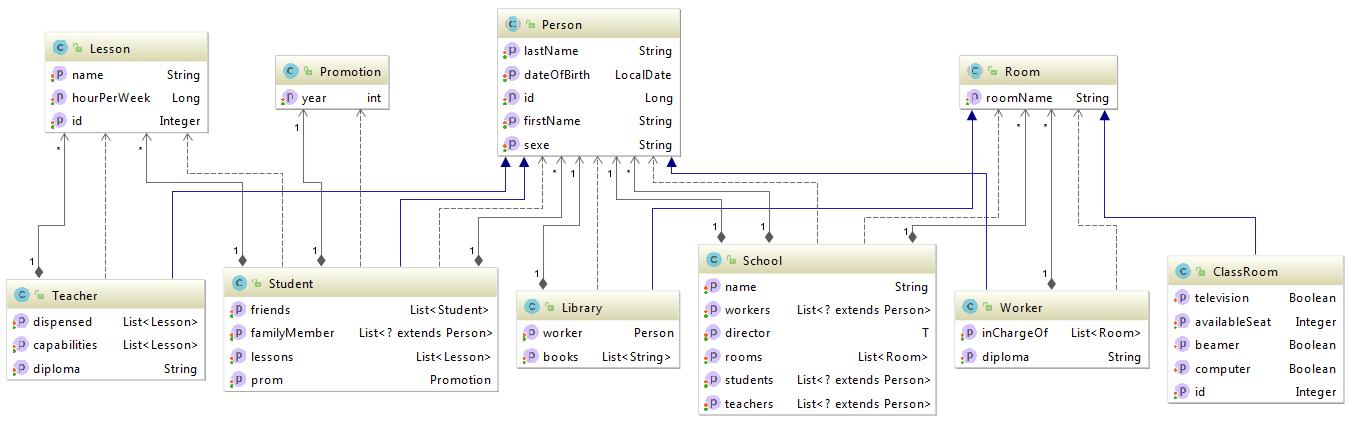
school.setStudents(Lists.newArrayList(gilles, thomasB, thomasR, charly, antoine, martin, benjamin, antoineBo));

school.setRooms(Lists.newArrayList(library, l003, l306, c203)); school.setWorkers(Lists.newArrayList(cassart));

return school;

}

39



|  |
| --- |
| 40 |

Figure 7.1 – Diagramme de Classe du modèle École

**Quatrième partie**

**Glossaire**

41

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

42

**Cinquième partie**

**Code Sources**

43

**Sixième partie**

**Bibliographie**

44

**Bibliographie**

1. Rohit Aggarwal. Networkx to neo4j’s documentation. https://neonx.readthedocs.io/ en/latest/, 2013.
2. JanusGraph Authors. Janusgraph oﬃcial website. http://janusgraph.org/, 2017.
3. Rudi Bruchez. Les bases de données nosql et le big data. Eyrolles,, 2016.
4. traduit par Simon Baslé Écrit par Peter Neubauer. Les bases orientées graphes, nosql et neo4j. https://www.infoq.com/fr/articles/graph-nosql-neo4j, 20 mai 2010.
5. NetworkX developers. Software for complex networks. https://networkx.github.io/, 2014-2017.
6. Roy T Fielding and Richard N Taylor. Architectural styles and the design of network-based software architectures. University of California, Irvine Doctoral dissertation, 2000.
7. The Apache Software Foundation. Apache tinkerpop oﬃcial website. http://tinkerpop. apache.org/, 2017.
8. ArangoDB GmbH. Arrangdb oﬃcial website. https://www.arangodb.com/, 2017.
9. Jure Leskovec. Stanford large network dataset collection. https://snap.stanford.edu/ data/, 2017.
10. OrientDB LTD. Orientdb oﬃcial website. https://orientdb.com/, 2017.
11. Inc Neo Technology. Neo4j oﬃcial website. https://neo4j.com/, 2017.
12. solid IT gmbh. Knowledge base of relational and nosql database management systems. https://db-engines.com, 2017.
13. Mohamed Tarroumi. Nosql (not only sql). https://prezi.com/4flswlgipwbo/ nosql-not-only-sql/, 10 Décembre 2014.
14. OrientDB Team. Orientdb - getting started. https://www.udemy.com/ orientdb-getting-started, 2017.
15. Directeur France et Europe du Sud de MongoDB Yann Aubry. Pourquoi nosql s’impose face aux sgbdr traditionnelles (avis d’expert). http://www.silicon.fr/ base-donnees-nosql-impose-sgbdr-93305.html, 19 mars 2014.

45