

INSTITUTO TECNOLÓGICO DE AERONÁUTICA



Allan Machado da Silva

Extensão do Framework Esfinge QueryBuilder
para Bancos de Dados NoSQL

Trabalho de Graduação
2012

Computação

Allan Machado da Silva

Extensão do Framework Esfinge QueryBuilder para Bancos de Dados NoSQL

Orientador

Prof. Dr. Eduardo Martins Guerra (ITA)

Engenharia de Computação

SÃO JOSÉ DOS CAMPOS

INSTITUTO TECNOLÓGICO DE AERONÁUTICA

2012

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Silva, Allan Machado

Extensão do Framework Esfinge QueryBuilder para Bancos de Dados NoSQL / Allan Machado da Silva.
São José dos Campos, 2012.
78f.

Trabalho de Graduação – Engenharia de Computação – Instituto Tecnológico de Aeronáutica, 2012.
Orientador: Prof. Dr. Eduardo Martins Guerra.

1. Bancos de Dados 2. Engenharia de Sistemas 3. Computação
I. Instituto Tecnológico de Aeronáutica. II. Título

REFERÊNCIA BIBLIOGRÁFICA

SILVA, Allan Machado. **Extensão do Framework Esfinge QueryBuilder para Bancos de Dados NoSQL**. 2012. 78f. Trabalho de Conclusão de Curso. (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Allan Machado da Silva

TÍTULO DO TRABALHO: Extensão do Framework Esfinge QueryBuilder para Bancos de Dados NoSQL

TIPO DO TRABALHO/ANO: Graduação / 2012

É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias deste trabalho de graduação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desta monografia de graduação pode ser reproduzida sem a autorização do autor.




Allan Machado da Silva

Rua Santa Emília, 12 – Paciência

CEP 23580-025 – Rio de Janeiro – RJ

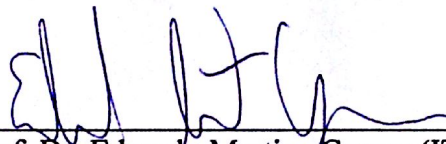
EXTENSÃO DO FRAMEWORK ESFINGE QUERYBUILDER PARA BANCOS DE DADOS NOSQL

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação



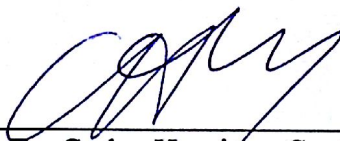
Allan Machado da Silva

Autor



Prof. Dr. Eduardo Martins Guerra (ITA)

Orientador



Prof. Dr. Carlos Henrique Costa Ribeiro
Coordenador do Curso de Engenharia de Computação

*Dedico esse trabalho aos meus pais,
que permitiram que eu chegasse até aqui.*

AGRADECIMENTOS

Aos amigos, que compensaram os momentos sofridos e desgastantes de estudante com situações engraçadas e únicas. Nunca serão esquecidos.

Aos meus pais, que sempre me deram apoio e me proporcionaram uma educação de qualidade, tão difícil nesse país.

À minha noiva Bárbara, que suportou meus momentos de estresse e me ajudou a seguir em frente nesses tantos anos de ITA.

“O único lugar em que o sucesso vem antes do trabalho é no dicionário”
Albert Einstein

RESUMO

Com o recente crescimento do uso de alternativas ao tradicional modelo relacional de bancos de dados, movimento conhecido como NoSQL, surge a necessidade de uma API que uniformize o acesso aos bancos sob um mesmo padrão – tanto os pertencentes ao novo paradigma quanto os já consagrados relacionais. Isso tem importância não só pela portabilidade que um tratamento uniforme proporciona, mas também pela simplificação do trabalho de quem desenvolve aplicações que fazem uso de bases de dados – o acesso heterogêneo exige conhecimentos específicos do desenvolvedor.

Esse trabalho é uma tentativa nesse sentido. Com as extensões desenvolvidas para o framework Esfinge QueryBuilder, seu escopo que antes era limitado a bancos relacionais passa, também, a contemplar os bancos MongoDB e Neo4J, provendo uma mesma interface de acesso para bancos pertencentes a paradigmas diferentes.

ABSTRACT

Considering the recent growth in the use of alternatives to the database relational model, movement that came to be known as NoSQL, the necessity of a database access API able to unify under the same standard both paradigms presents itself. This is justified not only by the obvious portability consequences, but also by the simplification of development and maintenance work – treating each database differently calls for specific knowledge.

This work is an attempt to provide such an API. The extensions developed for the Esfinge QueryBuilder framework extended its reach, that was limited to relational databases, to the NoSQL ones MongoDB and Neo4J. The same access interface is provided for databases belonging to different paradigms.

LISTA DE FIGURAS

Figura 1 – Rede social em um grafo (amigos são atingidos em percurso unitário)	22
Figura 2 – Relacionamento “é amigo de” em um banco relacional	33
Figura 3 – Mapeamento objeto-relacional.....	37
Figura 4 – Módulos do Framework Esfinge QueryBuilder.....	43
Figura 7 – Diagrama de classes referente à implementação da interface <i>EntityClassProvider</i>	52
Figura 8 - Diagrama de classes referente à implementação da interface <i>Repository</i>	54
Figura 9 - Diagrama de classes referente à implementação da interface <i>QueryExecutor</i>	55
Figura 10 – Diagrama de classes do mapeador objeto-relacional desenvolvido para o banco Neo4J.....	60
Figura 11 – Diagrama de classes referente à implementação da interface <i>QueryExecutor</i> para o banco <i>Neo4J</i>	61
Figura 12 – Tela da aplicação <i>Pet Shop Systems</i>	66

LISTA DE TABELAS

Tabela 1 – Exemplo de uma relação de cachorros	18
Tabela 2 – Relação dos donos dos cachorros	19
Tabela 3 – Tabela auxiliar de um relacionamento “muitos para muitos”	19
Tabela 4 – Testes realizados	64

LISTA DE CÓDIGOS FONTE

Listagem 1 – API exposta por bancos chave/valor.....	24
Listagem 2 – Exemplo de documento de um banco orientado a documentos.....	26
Listagem 3 – Exemplo de <i>traversal</i>	28
Listagem 4 – Coluna de um banco do tipo <i>Column Families</i>	29
Listagem 5 – Família de colunas de um banco do tipo <i>Column Families</i>	29
Listagem 6 – Exemplo de uso do JDBC.....	35
Listagem 7 – Extração de resultados de um <i>ResultSet</i>	35
Listagem 8 – Exemplo de mudança de banco usando JDBC	35
Listagem 9 – Exemplo de uso do JPA: anotando a classe representativa da entidade	38
Listagem 10 – Exemplo de uso do JPA: configurando a comunicação por meio do <i>EntityManager</i>	38
Listagem 11 – Exemplo de uso do JPA: arquivo xml de configuração.....	38
Listagem 12 – Exemplo de uso do JPA: realização da consulta	39
Listagem 13 – Exemplo de query no MongoDB fazendo uso do Morphia.....	40
Listagem 14 – Exemplo simplificado de query no Neo4J.....	40
Listagem 15 – Definição da interface <i>QueryExecutor</i>	45
Listagem 16 - Definição da interface <i>Repository</i>	45
Listagem 17 - Definição da interface <i>EntityClassProvider</i>	46
Listagem 18 – Exemplo de <i>QueryObject</i>	47
Listagem 19 – Exemplo de implementação da interface <i>DatastoreProvider</i> para o banco <i>MongoDB</i>	49
Listagem 20 – Exemplo de uso do framework Morphia	50
Listagem 21 – Exemplo de uso do framework Morphia: definição de uma entidade de persistência	51
Listagem 23 – Implementação do método <i>QueryByExample</i> da interface <i>Repository</i> para o banco MongoDB	53
Listagem 24 – Exemplo de implementação da interface <i>DatastoreProvider</i> para o banco <i>Neo4J</i>	57
Listagem 25 – Exemplo de criação de query “idade > 14 ou nome = João” no mapeador objeto-relacional criado	58

Listagem 26 – Exemplo de criação de query “idade = 14 ou (nome = João e sobrenome = Silva)” no mapeador objeto-relacional criado	59
Listagem 27 – Definição da interface <i>DatastoreProvider</i> para o banco MongoDB	67
Listagem 28 – Conteúdo do arquivo de configuração “META-INF/services/org.esfinge.querybuilder.mongodb.DatastoreProvider”	68
Listagem 29 – Definição da interface <i>DatastoreProvider</i> para o banco Neo4J	68
Listagem 30 – Conteúdo do arquivo de configuração “META-INF/services/org.esfinge.querybuilder.neo4j.DatastoreProvider”	68
Listagem 31 – Anotações na classe Cliente para JPA	69
Listagem 32 – Anotações na classe Pagamento para JPA.....	69
Listagem 33 – Anotações na classe Cachorro para JPA.....	69
Listagem 34 – Anotações na classe Cliente para o banco MongoDB	70
Listagem 35 – Anotações na classe Pagamento para o banco MongoDB.....	70
Listagem 36 – Anotações na classe Cachorro para o banco MongoDB.....	70
Listagem 37 – Anotações na classe Cliente para o banco Neo4J	71
Listagem 38 – Anotações na classe Pagamento para o banco Neo4J.....	71
Listagem 39 – Anotações na classe Cachorro para o banco Neo4J.....	71

LISTA DE ABREVIATURAS, SIGLAS E SÍMBOLOS

ACID – *Atomicity, Consistency, Isolation, Durability*

API – *Application Programming Interface*

CRUD – *Create, Read, Update, Delete*

JDBC – *Java Database Connectivity*

JPA – *Java Persistence Application Programing Interface*

JSON – *JavaScript Object Notation*

NoSQL – *Not Only SQL*

SQL – *Structured Query Language*

SUMÁRIO

1	INTRODUÇÃO	13
1.1	Contexto	13
1.2	Objetivo.....	14
1.3	Abordagem de Solução	14
1.4	Relevância e Originalidade	15
1.5	Avaliação.....	16
1.6	Organização do Trabalho	16
2	BANCOS DE DADOS	18
2.1	Bancos Relacionais	18
2.1.1	Espaço de Armazenamento	20
2.1.2	Desempenho	21
2.1.3	Escalabilidade.....	22
2.2	NoSQL	23
2.3	Tipos.....	23
2.3.1	Chave/valor.....	24
2.3.2	Orientados a Documentos.....	25
2.3.3	Baseados em Grafos	27
2.3.4	<i>Column Families</i>	28
2.4	Bancos NoSQL Utilizados no Trabalho.....	30
2.4.1	MongoDB	30
2.4.2	Neo4J.....	32
3	ABORDAGEM DE ACESSO A DADOS	34
3.1	Execução Direta de Comandos	34
3.2	Mapeamento Baseado em Metadados	36
3.3	Acesso a Bancos NoSQL	39
3.4	Esfinge QueryBuilder.....	40
4	EXTENSÃO DO QUERYBUILDER PARA NoSQL	44
4.1	As Interfaces de Comunicação	44
4.2	MongoDB.....	47
4.2.1	Configuração e Uso	48
4.2.2	Implementação	50
4.2.3	Limitações e Dificuldades	56
4.3	Neo4J.....	56
4.3.1	Configuração e Uso	57
4.3.2	Implementação	58
4.3.3	Limitações e Dificuldades	62
5	AVALIAÇÃO.....	63
5.1	Testes.....	63
5.2	Estudo de Caso	65
5.2.1	Configuração	66
5.2.2	Anotações	68
5.3	Análise do Estudo de Caso.....	71
6	CONCLUSÃO	73
6.1	Contribuições	74
6.2	Trabalhos Futuros.....	74

1 INTRODUÇÃO

1.1 Contexto

O uso de banco de dados para armazenar informações pertinentes a aplicações é algo extremamente comum e difundido. De fato, raras são as ocasiões em que um programa qualquer não tem necessidade de manter algum tipo de dado salvo para uso posterior. Desejando “persistir” os dados, termo que doravante utilizaremos para designar o ato de gravar em memória não volátil, o desenvolvedor conta com duas principais maneiras: o sistema de arquivos nativo do sistema operacional, adequado em muitas ocasiões, ou bancos de dados, mais indicados no caso de informações estruturadas, ou seja, informações que apresentam relações complexas entre si.

A fim de tornar mais clara a distinção feita acima, vamos exemplificar com dois casos. Primeiro, tome um editor de texto. Perceba que os dados inseridos pelo usuário, o texto em si, são apenas uma sequência de caracteres. É fácil guardá-los e recuperá-los com uma escrita ou leitura sequencial, respectivamente. Agora, imagine os dados de uma rede social. É difícil gerar uma sequência que caracterize toda a intrincada forma pela qual as pessoas se relacionam e que define sua rede de amigos, por exemplo. Enquanto os arquivos são úteis no primeiro caso, se tornariam um problema no segundo, que seria mais facilmente tratado por meio de um banco de dados.

Essa facilidade se justifica na estrutura que os bancos disponibilizam ao desenvolvedor. Não se limitam somente a receber uma massa de dados e guardá-la: associam a esses dados informações de como eles se relacionam. Isso pode ser feito de diversas maneiras, as quais dão origem a diferentes tipos de bancos.

Esses tipos se dividem em dois grandes grupos: os bancos de dados relacionais, que monopolizaram o cenário de persistência de dados durante vários anos, e os bancos NoSQL – *Not Only SQL* – que surgiram para tratar limitações do paradigma relacional que se tornaram especialmente críticas nos sistemas de hoje em dia – escalabilidade horizontal e alta disponibilidade.

A existência de variações, contudo, pode ser um problema para o desenvolvedor. Bancos diferentes implicam em maneiras de interagir com os mesmos diferentes, ou seja, não existe uma API uniforme – *Application Programming Interface*, interface exposta por um

software para que possa ser feito uso de seus serviços sem preocupações com detalhes internos de programação. Isso causa grande impacto em dois aspectos que são, de maneira geral, desejáveis: facilidade de manutenção e portabilidade das aplicações.

Caso a interface fosse única, uma aplicação que roda sobre um tipo de banco poderia facilmente rodar sobre outro, sem qualquer alteração de código – ou com poucas. Nesse caso, dizemos que não há dependência, ou seja, a camada de aplicação não precisa de qualquer informação sobre como de fato está sendo feita a persistência. Ela apenas se comunica com a camada do banco por meio de uma interface pré-definida e uniforme, confiando que o trabalho requisitado será feito.

Perceba que, além de permitir a troca livre do sistema de banco de dados utilizado ou, em outras palavras, a portabilidade, essa uniformidade também facilita o processo de manutenção do código. Ora, a manutenção é feita por humanos; uma interface padrão e bem definida é muito mais tratável do que um conjunto de regras variável.

Dessa forma, é altamente desejável que a interface de acesso aos diferentes bancos de dados seja uniformizada, de maneira a reduzir o trabalho do programador no próprio desenvolvimento das aplicações e permitir, em um segundo momento, sua portabilidade e manutenção.

1.2 Objetivo

Desenvolver uma solução que disponibilize uma API uniforme para diferentes tipos de bancos de dados a fim de prover portabilidade para as aplicações que a utilizem.

1.3 Abordagem de Solução

Esse Trabalho de Graduação visa estender o Framework Esfinge QueryBuilder a bancos de dados NoSQL. O Esfinge QueryBuilder consiste em um framework para consultas a bases de dados, inteligente o suficiente para inferir o que precisa ser feito partindo apenas dos nomes dos métodos utilizados. Basta que o usuário crie uma interface com os métodos de acesso desejados, respeitando uma sintaxe pré-definida em sua nomeação, para que a comunicação com o banco funcione.

No momento em que esse Trabalho de Graduação foi iniciado, o framework trabalhava apenas com bancos de dados relacionais, através das APIs JPA e JDBC. O objetivo do mesmo, então, é permitir seu funcionamento em conjunto com dois bancos que não

pertencem a essa categoria, pois seu armazenamento se baseia em paradigmas diferentes: o MongoDB, baseado em documentos, e o Neo4J, que armazena seus dados em uma estrutura de grafos.

O Framework Esfinge QueryBuilder foi, desde o princípio, pensado para ser facilmente extensível. Apresenta design modular, de forma que todo o código que lida com as particularidades de cada banco possa ser alterado sem qualquer efeito sobre o seu programa-base. Caso surja a necessidade de torná-lo compatível com qualquer outro tipo de base de dados que possa surgir, bastará criar o módulo específico para esse banco, observando as necessidades definidas por interfaces de comunicação especificadas.

Esse Trabalho de Graduação tem seu foco na extensão do Esfinge QueryBuilder a bancos NoSQL ou, mais especificamente, aos bancos MongoDB e Neo4J. Dessa forma, serão desenvolvidos os módulos referentes a cada para que possam trabalhar em conjunto com o kernel.

1.4 Relevância e Originalidade

Deve ser ressaltado o fato de que, apesar de essas novas bases de dados – que não seguem o paradigma relacional – estarem surgindo e crescendo em uso, não existe uma interface que unifique o acesso aos mesmos sob um mesmo padrão. Como já foi dito, essa realidade heterogênea em que cada banco define sua própria interface de comunicação é ruim tanto para o desenvolvimento de aplicações que utilizem tais bancos – pois exigirá do desenvolvedor conhecimentos específicos acerca daquela base de dados – como para a manutenção do código. Some-se a isso o fato de toda a camada de acesso ao banco da aplicação dever ser reescrita no caso da necessidade de alteração da base utilizada – as aplicações não são portáteis.

Ainda, diferentemente daquilo que é disponibilizado pelos frameworks de acesso a bancos existentes na linguagem Java atualmente – JDBC e JPA – o Esfinge QueryBuilder tem como proposta uma maneira de acesso única em sua originalidade: não é necessário se preocupar em de fato escrever as queries, elas são geradas em tempo de execução simplesmente a partir do nome do método. Isso certamente é algo que gerará avanço na produtividade de seus utilizadores.

1.5 Avaliação

A fim de averiguar o cumprimento do objetivo desse trabalho – que tem seu foco na portabilidade de aplicações entre bases de dados diversas – foram realizados diversos testes, tanto unitários como de integração dos módulos com a parte principal do programa.

Perceba que, como a interface provida pelo kernel do framework é única para todos os bancos de possível utilização, garantir a portabilidade é equivalente a garantir que todas as requisições que podem ser feitas pelo kernel aos módulos específicos de cada banco são respondidas de maneira correta.

Contudo, na prática é impossível definir um conjunto de testes que, de maneira exhaustiva, verifique a ausência de qualquer erro em uma aplicação. O que pode ser feito é definir um conjunto que garanta pelo menos um nível de confiança considerado adequado, processo que normalmente tem base nos casos de uso definidos para a aplicação. Como para os módulos desenvolvidos nesse Trabalho de Graduação os casos de uso são referentes às funcionalidades providas pelo kernel, considerou-se adequado testar individualmente cada uma dessas funcionalidades como forma de validação.

1.6 Organização do Trabalho

A fim de atingir o seu objetivo final, apresentar uma extensão funcional do Framework Esfinge QueryBuilder para os bancos NoSQL MongoDB e Neo4J, esse Trabalho de Graduação será organizado da maneira descrita nessa seção.

O capítulo 2 apresenta os diferentes tipos de bancos de dados. São comentadas as restrições impostas pelo teorema CAP (GILBERT e LYNCH, 2002) e como cada tipo de banco lida com as mesmas. Serão, também, introduzidos os bancos alvo desse Trabalho de Graduação: MongoDB e Neo4J.

O capítulo 3 tratará do que existe hoje em dia no tocante ao acesso a bancos, tanto relacionais como NoSQL. Serão apresentadas as APIs JPA e JDBC, que tratam dos bancos relacionais, assim como o que é disponibilizado para uso dos bancos NoSQL. Será apresentado o Framework Esfinge QueryBuilder e o que se propõe a fazer.

O capítulo 4 tratará com detalhes o que foi de fato desenvolvido, apresentando a maneira como foram feitas as extensões, como configurá-las e suas limitações.

O capítulo 5 apresentará testes realizados e o uso do Framework em uma aplicação real, fazendo uso tanto de bancos relacionais como do MongoDB e do Neo4J e apresentando o resultado obtido: independência e portabilidade.

Por fim, o capítulo 6 conterá, de forma resumida, uma descrição do problema original e das contribuições dadas por esse Trabalho de Graduação à sua resolução. Serão considerados possíveis trabalhos futuros de maneira a dar continuidade, melhorando e complementando, ao desenvolvido aqui.

2 BANCOS DE DADOS

Esse trabalho tem por objetivo uniformizar o acesso a bancos de dados, provendo uma interface única tanto para aqueles pertencentes ao grupo dos relacionais como para os bancos NoSQL. Todavia, ainda não foi discutido o que de fato são esses bancos, porque configuram um paradigma à parte do modelo relacional, quais são suas vantagens em relação a esse modelo e em quais casos seu uso é interessante. A fim de entender essas diferenças torna-se necessário falar, primeiro, sobre o modelo relacional.

2.1 Bancos Relacionais

Um banco de dados relacional (CODD, 1970) é um conjunto de dados formalmente organizados na forma de tabelas. Uma relação (uma tabela) é definida como um conjunto de tuplas com os mesmos atributos. Cada tupla contém informações acerca de um objeto, uma entidade de persistência, sendo seus valores referentes aos atributos dessa entidade. Essas tuplas devem, de maneira obrigatória, definir uma identificação única, chamada chave primária.

Tabela 1 – Exemplo de uma relação de cachorros

Nome (Chave Primária)	Raça (Um Atributo)	Cor (Um outro atributo)
Rayka	Pitbul	Malhada
Lalinha	Poodle	Branca
Rex	Labrador	Creme

A tabela acima exhibe os dados de três tuplas que obedecem à forma (nome, raça, cor), a saber: (Rayka, pitbul, malhada), (Lalinha, poodle, branca) e (Rex, labrador, creme). A chave primária, nesse caso, é definida pelo primeiro elemento de cada tupla. Perceba que, enquanto seria válido um banco com duas tuplas (Rex, labrador, creme) e (Rayka, labrador, creme), não havendo qualquer problema na repetição das características, as tuplas seguintes fugiriam à

restrição de unicidade de chave primária e não seriam possíveis: (Rex, labrador, creme) e (Rex, pitbul, malhada).

Relacionamentos entre entidades diferentes também são tratados nas tabelas, com a adição de atributos que se referem à chave primária da entidade associada. No nosso exemplo de cachorros, caso tivéssemos, também, uma outra tabela com os dados de seus donos e quiséssemos associá-los aos seus respectivos cães, isso poderia ser feito da seguinte maneira:

Tabela 2 – Relação dos donos dos cachorros

Nome	Endereço	Cachorro
Renan	Rua Benedito, 23	Rayka
Sandra	Rua Santa Bárbara, 47	Lalinha
Igor	Rua São Clemente, 10	Rex

Perceba que as chaves primárias da relação de cachorros entram na tabela de donos como um atributo. Essa construção permite que um cachorro tenha mais de um dono, mas não aceita que uma pessoa tenha mais de um cachorro. Nesse caso, dizemos que o relacionamento é “de um para muitos” – de um cachorro para muitos donos. Se o desejado fosse o inverso, ou seja, que cada cachorro tivesse apenas um dono e cada pessoa pudesse ter mais de um cão, a chave primária da relação de pessoas seria inserida na relação de cachorros na forma de um atributo, de maneira análoga. Também existe a possibilidade de armazenar relacionamentos “de muitos para muitos” com o uso de uma tabela auxiliar.

Tabela 3 – Tabela auxiliar de um relacionamento “muitos para muitos”

Identificação da pessoa (sua chave primária)	Identificação do cachorro (sua chave primária)
Renan	Rayka
Renan	Rex
Igor	Rex

Na tabela acima está mostrado um relacionamento “muitos para muitos”, no qual existe uma pessoa com mais de um cachorro (Renan é dono tanto de Rayka como de Rex) e um cachorro com mais de um dono (Rex é tanto de Renan como de Igor). Atente ao fato de que essa tabela auxiliar, assim como todas as outras, também deve definir uma chave primária. Todavia, desejamos que tanto a identificação das pessoas como dos cães seja repetível, pois caso contrário fugiríamos ao próprio propósito do relacionamento “de muitos para muitos”. Felizmente, chaves primárias não necessitam ser um atributo apenas, podendo ser um conjunto de atributos. É válido definir o conjunto {identificação da pessoa, identificação do cachorro} como chave primária, de maneira que a igualdade de chaves ocorreria apenas se ambos os atributos fossem iguais.

Os bancos de dados relacionais são, dessa forma, basicamente conjuntos de grandes tabelas onde os dados são armazenados, até mesmo os referentes a relacionamentos entre entidades. Nessas tabelas, cada linha (cada tupla) se refere a uma entidade de persistência, enquanto cada coluna representa um atributo dessa entidade (um atributo presente na tupla).

Nesse nosso exemplo simples, o paradigma relacional parece se adequar muito bem. Os dados são salvos de maneira clara e simples, com sua recuperação sendo igualmente fácil. Todavia, esse modelo apresenta suas limitações.

2.1.1 Espaço de Armazenamento

Considere um caso em que não existe um *schema* fixo, ou seja, em que nem todas as entidades apresentam os mesmos atributos, existindo atributos de determinadas entidades que não são aplicáveis a outras. Para que essas entidades possam ser armazenadas na tabela, será necessária uma coluna extra para guardar seu atributo exclusivo. Essa coluna, apesar de presente em todas as entidades da tabela, não é efetivamente utilizada por cada uma, provocando desperdício de espaço.

A fim de tornar mais claro o que foi dito, tomemos um exemplo com dados concretos. Seja um sistema hipotético que armazena informações acerca de um grupo de pessoas, incluindo seu emprego, a carga horária semanal a que estão submetidos, seu salário, etc. Acontece que, dentre os membros desse grupo, existem aqueles que estão desempregados. Para esses, os atributos referentes às características do emprego simplesmente não se aplicam.

Em termos de aproveitamento do espaço de armazenamento, o ideal seria que, dependendo da necessidade, o atributo estivesse ou não presente: pessoas com emprego seriam refletidas por entidades com os atributos necessários, enquanto desempregadas os

omitiriam. A tabela não é, contudo, alterável linha a linha. Todas as linhas devem, de maneira obrigatória, apresentar o mesmo conjunto de atributos, as mesmas colunas. Essa obrigação faz com que, mesmo nos casos em que nenhuma informação será armazenada, seja reservado espaço em memória, inutilmente.

Enquanto essa característica é claramente negativa no tocante à utilização de espaço, a restrição que impõe é uma garantia de consistência, umas das condições ACID (HAERDER e REUTER, 1983). É sabido de antemão exatamente o que se encontra em cada entidade persistida no banco, mesmo que algumas das informações contidas na mesma sejam desprovidas de significado. Isso não é verdade quando os atributos de cada entidade podem ser quaisquer, o que clama por um controle maior por parte da aplicação que faz uso do banco de dados. Afinal,

Informação é o resultado do processamento, manipulação e organização de dados, de tal forma que represente uma modificação (quantitativa ou qualitativa) no conhecimento do sistema (pessoa, animal ou máquina) que a recebe (SERRA, 2007)

de maneira que os dados nada representam se não soubermos organizá-los em informação. A escolha se resume em um *trade-off*: consistência por eficiência em armazenamento.

2.1.2 Desempenho

Outra questão que pode ser levantada como uma limitação do modelo relacional é a forma como são tratados os relacionamentos entre entidades. A estratégia de chaves estrangeiras – nome dado às chaves primárias de outras tabelas que aparecem como atributos – pode parecer simples e efetiva, mas não é a mais adequada para dados altamente conectados, como os de uma rede social, por exemplo.

Imagine que os dados de uma dessas redes estejam armazenados em um banco relacional. O relacionamento “é amigo de” é, naturalmente, do tipo “de muitos para muitos”, visto que cada pessoa é livre para se relacionar com quaisquer outras. Se desejarmos consultar o banco a fim de encontrar os amigos de alguém, seria necessário percorrer a tabela auxiliar do relacionamento – conforme a utilizada acima – em busca das ocorrências dessa pessoa para, em seguida, percorrer a relação de usuários em busca daqueles identificados como amigos na primeira busca. Esse processo é especialmente crítico quando as tabelas envolvidas se tornam muito grandes.

Agora pense em um banco que armazene seus dados em grafos ao invés de tabelas. Cada pessoa teria uma ligação direta com seus amigos, simplificando bastante o problema. Na

situação acima, ao invés de percorrer duas tabelas em busca dos usuários, bastaria um percurso de tamanho unitário pelo grafo.

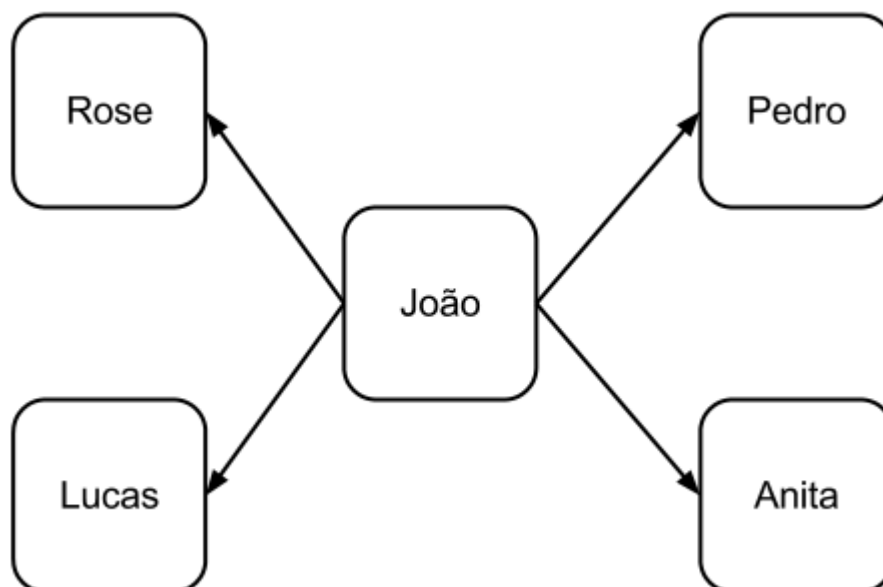


Figura 1 – Rede social em um grafo (amigos são atingidos em percurso unitário)

2.1.3 Escalabilidade

Essa seção é fortemente baseada na referência (EINI).

Bancos relacionais seguem rigidamente as condições ACID (HAERDER e REUTER, 1983), o que acaba gerando dificuldades quando a questão é escalabilidade horizontal. O problema é inerente aos requisitos básicos de um banco de dados relacional: a consistência é obrigatória de maneira a manipular corretamente relacionamentos e chaves estrangeiras. Ao tentar escalar o banco horizontalmente por um grande número de servidores, as limitações impostas pelo teorema CAP (GILBERT e LYNCH, 2002) se fazem presentes. De acordo com o teorema, apenas duas das seguintes três características podem ser obtidas em um sistema distribuído: consistência, disponibilidade e tolerância ao particionamento (no original: *Consistency, Availability e Partition Tolerance*). Como o modelo relacional dita a absoluta necessidade de consistência, uma das outras duas características deverá ser abandonada, inviabilizando a escalabilidade horizontal.

Uma maneira de burlar esse problema é a prática de *sharding*, que quebra o banco original em bancos independentes sendo executados em nós separados. Isso, contudo, elimina

algumas capacidades do modelo relacional, pois determinadas operações não são permitidas entre nós diferentes – JOIN, por exemplo.

A escalabilidade restante é a vertical, o que significa dizer que, para uma maior performance, é necessária a aquisição de servidores maiores e mais potentes ao invés de simplesmente aumentar o número de servidores e distribuir a carga, o que pode ser um problema no aspecto financeiro.

Bancos NoSQL, por outro lado, não têm o peso de garantir as condições ACID (HAERDER e REUTER, 1983). Trabalhando com o conceito de consistência eventual (VOGELS, 2009), pelo qual é garantido que todos os nós refletirão uma determinada mudança nos dados tendo decorrido tempo suficiente, torna-se mais fácil o crescimento do seu número. Isso será discutido adiante.

2.2 NoSQL

O termo NoSQL foi utilizado pela primeira vez em 1998 para denominar um projeto (STROZZI) de banco de dados que, apesar de seguir o paradigma relacional, deliberadamente não utilizava a linguagem SQL. Sua origem nada tem a ver com o movimento atual, cujos bancos participantes fogem completamente ao modelo relacional. Por esse motivo, Carlo Strozzi, seu cunhador, sugere que uma denominação mais adequada seria NoREL. Apesar disso, o termo continua em pleno uso.

NoSQL, hoje, se refere a uma classe abrangente de softwares gerenciadores de bancos de dados que se caracterizam por não seguirem o modelo relacional, ou seja, por não necessariamente basearem-se em tabelas e relacionamentos por meio de chaves estrangeiras. O termo é lido como *Not Only SQL*, fugindo do conceito inicial de Strozzi para se adequar a seu uso. A classe de bancos NoSQL apresenta vários integrantes com características distintas, cada um tratando o armazenamento à sua maneira e fazendo suas escolhas no que diz respeito ao teorema CAP (GILBERT e LYNCH, 2002).

2.3 Tipos

Os tipos que serão enumerados nesse Trabalho de Graduação são os principais existentes no momento em que é escrito. Nada limita a possibilidade de surgirem outros futuramente. Dessa forma, essa seção não pretende apresentá-los de maneira exaustiva.

2.3.1 Chave/valor

Baseia-se no conceito simples sugerido por seu nome: são armazenadas chaves e a cada uma corresponde um valor. De maneira geral, expõem a seguinte API, ou variações da mesma:

```
void put(String key, byte[] data);  
byte get(String key);  
void remove(String key);
```

Listagem 1 – API exposta por bancos chave/valor

Um banco chave/valor não se importa com o que é armazenado. Ele simplesmente guarda os dados e os associa à chave passada. Toda a tarefa de dar estrutura aos mesmos, de saber como lê-los e transformá-los em informação cabe à aplicação que os utiliza.

Por sua simplicidade, são capazes de atingir uma boa performance. De fato, estruturas como Árvores AVL (ADELSON-VELSKII e LANDIS, 1962) permitem que o acesso aos dados ocorra em ordem logarítmica (HARDY e WRIGHT, 1979).

Em um banco desse tipo, concorrência só se aplica no acesso individual a uma chave. Nesse cenário com baixo nível de conectividade entre os dados, a concorrência normalmente é tratada pela estratégia otimista (KUNG, 1981) ou pelo modelo de consistência eventual (VOGELS, 2009). Em ambientes distribuídos com um número grande de nós, contudo, a estratégia otimista pode se tornar lenta devido ao custo de verificar a alteração nos dados, que podem ter sido replicados a várias máquinas. Ainda, existem consequências para cada escolha impostas pelo teorema CAP (GILBERT e LYNCH, 2002), como será discutido adiante.

Por sua natureza, esses bancos são muito limitados no tocante a maneiras de fazer queries nos dados. Não existe outra maneira de busca-los a não ser pela chave correspondente. Funcionalidades mais complexas devem ser implementadas pelo usuário.

O fato de as queries serem feitas apenas por chave, sem qualquer operação possível sobre seus valores – se houver um valor “nome”, por exemplo, não é possível fazer uma query “nome” = “Paulo” –, pode parecer uma limitação desagradável. Todavia, é extremamente importante de maneira a ser possível obter uma alta escalabilidade. Caso fosse permitida uma operação sobre valores, como recuperar valores máximos, por exemplo, seria necessário que as informações de todos os nós estivessem disponíveis. Isso implica na necessidade de resposta de todos, comprometendo a tolerância ao particionamento, ou na centralização, derrotando o propósito.

O espaço de chaves pode, então, ser dividido entre vários nós: por exemplo, chaves que começam com A vão para um servidor, chaves que começam com B vão para outro, etc. Perceba que isso é, novamente, o *sharding* que foi comentado na seção sobre bancos de dados relacionais. Aqui, todavia, não existe qualquer perda de capacidades do banco por conta dessa prática, como lá ocorria.

É desejável que as chaves não se encontrem apenas em um servidor, pois isso tornaria o sistema vulnerável à perda de dados, além de torna-lo parcialmente indisponível no caso de particionamento da rede. Para combater esses efeitos indesejáveis, introduzimos a replicação, mas essa traz consigo um novo problema: a consistência. Permitir que uma mesma chave esteja em vários servidores abre margem à ocorrência de versões divergentes: um servidor guarda um valor para a chave enquanto outro servidor tem outro valor armazenado.

Perceba que, caso a estratégia adotada seja rejeitar absolutamente a inconsistência, ou seja, não permitir que uma operação de escrita se complete caso exista a possibilidade de a mesma produzir divergências entre servidores, isso implica em abrir mão da tolerância a particionamento. De fato, sem acesso a todos os servidores para garantir que os dados fiquem consistentes, uma operação de escrita não pode ser completada.

Nesse ponto ficam bem claras as restrições impostas pelo teorema CAP (GILBERT e LYNCH, 2002): sem replicação, temos consistência e tolerância a particionamento, mas não temos disponibilidade; com replicação e garantia de consistência, temos disponibilidade e consistência, mas não temos tolerância a particionamento; com replicação e sem garantia de consistência, temos disponibilidade e tolerância a particionamento, mas não temos consistência.

Aqui entra a consistência eventual (VOGELS, 2009), modelo pelo qual é permitido que o sistema se torne inconsistente em uma janela temporal. De maneira resumida, o que é feito é o seguinte: em uma operação de escrita, ao invés de exigir o *acknowledge* de todos os servidores antes de aplicar o *commit*, aceita-se que a operação se complete com uma fração desse número; essa mudança é, eventualmente, replicada aos servidores que não estavam disponíveis no momento da operação. O controle de última versão pode facilmente ser obtido em um modelo de banco chave/valor, pois é simples comparar versões diferentes.

2.3.2 Orientados a Documentos

São, em essência, bancos chave/valor. Aqui, contudo, o valor é imbuído de estrutura, não sendo mais apenas uma sequência de bytes sem significado. Um banco desse tipo exige

que os dados a serem armazenados sejam estruturados em algum formato que ele é capaz de entender, seja este XML, JSON, BSON, etc.

O que ganhamos com isso é a capacidade do banco de fazer queries sobre os dados, não mais apenas sobre as chaves. Um banco chave/valor apenas armazena, um banco orientado a documentos tem informações sobre o que está guardando, o que lhe dá a possibilidade de operar com esse dados. Agora, supondo que o banco armazena um documento no formato

```
{  
  "nome": "Olavo",  
  "email": "olavo@ita.br"  
}
```

Listagem 2 – Exemplo de documento de um banco orientado a documentos

é possível fazer uma query em que são buscados todos os documentos cujo campo “nome” tem o valor “Olavo”, por exemplo. Isso tem implicações sobre o *sharding*, como será discutido.

Relacionamentos, contudo, não são suportados. Cada documento existe de maneira individual e isolada. É possível fazer referências a outros documentos, mas não existe nenhum mecanismo que garanta a integridade dessa relação.

Quanto à concorrência, são válidos os mesmos comentários feitos acerca de bancos chave/valor (2.3.1), com a diferença de que operações concorrentes se aplicam a um documento inteiro, não mais apenas a um valor.

Além de seguirem um padrão pré-definido de documento, seja esse JSON ou qualquer outro, não existe restrição quanto à forma. Cada documento pode definir seus campos livremente.

Perceba, ainda, que há pouca dificuldade na tradução objeto/documento, de maneira que armazenar dados em um banco orientado a documentos é mais fácil do que modelá-los em um paradigma relacional. Em casos não triviais, a maneira como os dados são pensados na aplicação e a forma como são armazenados em um banco relacional apresentam pouca semelhança.

O que foi discutido para bancos chave/valor na seção 2.3.1 quanto à escalabilidade também se aplica a bancos orientados a documentos. Baseando-se no fato de que cada documento é independente, é feito *sharding* da base através de vários servidores.

Contudo, o modo restritivo de recuperar dados dos bancos chave/valor tinha um papel importante em sua capacidade de *sharding*, como foi exposto. Dessa forma, deve ser esperado

um *tradeoff* para a obtenção da maior capacidade de consulta dos bancos orientados a documentos. Isso de fato ocorre e será discutido na seção específica do banco MongoDB (2.4.1), pertencente a essa classe.

As restrições e escolhas impostas pelo teorema CAP (GILBERT e LYNCH, 2002), naturalmente, também se aplicam aqui.

2.3.3 Baseados em Grafos

Armazenam, como o nome sugere, seus dados em grafos. Cada nó do grafo apresenta informações associadas, de maneira análoga aos documentos, não existindo restrições quanto à forma dos dados – cada nó define seus atributos de maneira independente. Os relacionamentos, contudo, ganham destaque: não são simples ponteiros para outros nós, são dotados de tipo – um nó pode se relacionar com outro de várias formas diferentes – e carregam informação.

O exemplo mais natural para um banco desse tipo são os dados de uma rede social. Cada pessoa tem informações inerentes a ela, como seu nome, idade, telefone, etc. Existem, também, informações pertinentes à forma como se conhecem: podem ser relacionar como amigos ou família, por exemplo.

Grafos são estruturas de dados com alto nível de interconectividade. Dessa forma, uma abordagem quanto à concorrência como as anteriores, tanto otimista (KUNG, 1981) como eventualmente consistente (VOGELS, 2009) tende a não ser a mais adequada. A mesma necessidade de consistência presente no paradigma relacional se apresenta aqui, de maneira que a abordagem por meio de transações e garantias ACID (HAERDER e REUTER, 1983) é o caminho.

Nesse ponto, cabe o questionamento: se bancos baseados em grafos se submetem às mesmas restrições dos relacionais, isso não os tornará difíceis de escalar de forma horizontal? Como seus dados são altamente conectados, é muito difícil encontrar subgrafos independentes que possam ser divididos em servidores diferentes. Qual é o ganho em utilizar essa estrutura ao invés da já consagrada? Ganhamos uma nova maneira de acessar os dados, como descrito a seguir.

Como grafos que são, esses bancos nos dão opções de trabalhar com operações de grafos. Uma maneira muito útil de realizar queries complexas é o *traversal*, ou percurso. Na nossa rede social, suponha que eu gostaria de descobrir quais dos meus amigos estão por perto para convidá-los para uma bebida. Simples certo? Mesmo em um banco relacional, basta

buscar meus amigos cuja localização é igual à minha. Agora, inclua nessa busca os amigos indiretos, ou seja, amigos de amigos. O cenário começa a ficar mais complicado para o paradigma relacional. Todavia, de um grafo essas informações vêm de maneira fácil. Veja:

```
new GraphDatabaseQuery{
    SourceNode = me,
    MaxDepth = 3,
    RelationsToFollow = new[]{"Family", "Friend"},
    Where = node => node.Location == ayende.Location,
    SearchOrder = SearchOrder.BreadthFirst
}.Execute();
```

Listagem 3 – Exemplo de *traversal*

2.3.4 *Column Families*

Os dados são vistos de uma maneira que se assemelha às tabelas do modelo relacional, porém são agrupados no disco de um modo diferente. A forma como os dados são efetivamente escritos no disco é importante porque o tempo de busca – *seek time* – tem grande peso na leitura de um arquivo. É possível que uma grande quantidade de dados escrita de forma sequencial seja recuperada em menos tempo do que uma quantidade menor, fragmentada.

No modelo relacional, os dados de cada linha são armazenados em sequência, de maneira que o processo de recuperar todas as colunas referentes a essa linha – ou a essa entidade – tem bom desempenho. Contudo, existem situações em que o desejo não é recuperar a totalidade dos dados de uma entidade, mas parte dos dados de várias. Suponha uma base de dados que armazena informações referentes às pessoas de um país: nome, nascimento, nomes dos pais, endereço, etc. Uma consulta que visa descobrir os nomes mais registrados pelos pais para seus filhos em cada ano, por exemplo, precisa do nome e da data de nascimento de todas as pessoas, mas não tem qualquer uso para outros dados.

Caso a base seja relacional, essa consulta provocará várias buscas pela cabeça de leitura, degradando o desempenho, visto que os dados de interesse não estão de maneira sequencial no disco. Para contornar isso surgiram as *column families*. A fim de entender seu conceito, pense em uma tabela de um banco relacional. Agora, considere que os dados de cada coluna, não cada linha, são escritos em sequência. Evolua o pensamento, considerando que grupos de colunas podem ser formados – famílias de colunas – e que esses grupos são escritos sequencialmente.

A maneira de escrever os dados não é a única diferença em relação aos bancos relacionais. Aqui, o *schema* não é fixo: cada linha da nossa tabela pode definir suas colunas, quaisquer e quantas sejam elas. As únicas obrigações quanto à forma são: cada coluna é uma combinação chave/valor – e rótulo de tempo – e super colunas não podem conter outras super colunas, apenas colunas. A fim de entender melhor como ocorre toda a organização dos dados, vamos definir as estruturas fundamentais: colunas, super colunas e famílias de colunas.

A coluna é o arranjo fundamental de dados. É uma tupla de três elementos: nome, valor e rótulo de tempo. Por simplicidade de compreensão, podemos ignorar o rótulo de tempo e pensar apenas no par nome/valor. Em uma notação JSON, simples de entender, uma coluna ficaria assim:

```
{ // isso é uma coluna
  nome: "enderecoEmail",
  valor: "arin@exemplo.com",
  rótulo: 123456789
}
```

Listagem 4 – Coluna de um banco do tipo *Column Families*

Uma super coluna nada mais é que um grupo de colunas. É definida como uma tupla com dois valores: seu nome e seu valor, o qual é um mapa de colunas cujas chaves são seus nomes. Uma super coluna não apresenta rótulo de tempo.

Por fim, uma família de colunas é uma estrutura que contém um número não limitado de linhas – cada linha apresenta uma chave e um mapa de colunas.

```
Usuarios = { // isso é uma família de colunas
  ronaldo: { // essa é a chave para essa linha dentro da família
    // aqui temos um número não limitado de colunas dentro da linha
    username: " ronaldo ",
    email: " ronaldo @exemplo.com",
    celular: "9976-6666"
  }, // fim da linha
  adriano: { // isso é a chave para outra linha na família
    //novamente temos um número não limitado de colunas
    username: " adriano ",
    email: " adriano @example.com",
    celular: "8555-1212"
    idade: "66",
    sexo: "masculino"
  },
}
```

Listagem 5 – Família de colunas de um banco do tipo *Column Families*

Por simplicidade, foram representados apenas os valores das colunas, quando na verdade os valores no mapa são as colunas inteiras, incluindo seus respectivos rótulos de tempo.

O acesso aos dados é limitado a queries por chaves, a fim de obter maior escalabilidade horizontal. De maneira semelhante ao que ocorre nos bancos chave/valor, a baixa conectividade propicia um controle por meio de estratégia otimista (KUNG, 1981) ou consistência eventual (VOGELS, 2009). O software Apache Cassandra (THE APACHE SOFTWARE FOUNDATION), membro dessa classe de bancos NoSQL, vai mais longe e permite ao usuário definir níveis de consistência para leitura e escrita.

A restrição quanto ao modo de recuperar os dados, apenas por meio de chaves, permite que o *sharding* seja feito sem grandes dificuldades. O que foi discutido para bancos chave/valor (2.3.1) se aplica.

2.4 Bancos NoSQL Utilizados no Trabalho

A seguir, dois exemplos de softwares que pertencem à classe de bancos de dados NoSQL: MongoDB e Neo4J. Serão apresentados com algum detalhamento, visto que a implementação de seus módulos para o framework Esfinge QueryBuilder é a base desse trabalho de Graduação e serão utilizados, à frente, no estudo de caso que averiguará o sucesso desse objetivo.

2.4.1 MongoDB

É um banco NoSQL *opensource* de alta performance, escrito em C++, orientado a documentos. O nome deriva da palavra *humongous* – algo muito grande, enorme. Seus documentos seguem o formato JSON, que é simples de ser lido e compreendido tanto por uma pessoa como por uma máquina.

Suas propriedades básicas são aquelas descritas na seção referente aos bancos NoSQL orientados a documentos (2.3.2), de modo que essa seção servirá para expor algumas características específicas do MongoDB em si, sendo a mais importante a maneira como lida com escalabilidade horizontal.

Por que a escalabilidade é tão importante? Ora, o grande problema atrelado ao paradigma relacional é justamente a incapacidade de escalar de maneira horizontal. Não faria

sentido uma mudança radical na forma de persistir os dados, e todo o esforço provocado pela mudança, caso uma solução não acompanhasse.

Foi dito na seção 2.3.2 que a maior capacidade de consulta obtida através da estrutura de documento implicaria em um *tradeoff* com a escalabilidade horizontal. Vamos explicar melhor o que isso quer dizer.

Tome um banco chave/valor, sem qualquer capacidade de query além da busca por chave. Fazer *sharding* nessa situação é simples, pois cada query que chega ao banco distribuído pode ser facilmente direcionada ao servidor correto.

Imagine que o *sharding* é feito da seguinte maneira: chaves que começam com a letra “A” vão para um servidor, as que começam com “B” vão para outro, etc. Dessa forma, caso chegue ao banco uma query com a chave “Ana”, é sabido exatamente qual nó do sistema distribuído pode conter a informação referente àquela chave. A query não precisa rodar em todos os nós, apenas naqueles efetivamente relevantes para a obtenção do resultado.

Agora tomemos o cenário do MongoDB. É escolhida uma chave para o *sharding*, ou seja, definimos um campo do documento – que por razões óbvias deve estar presente em todos os documentos – cujo valor será utilizado na distribuição através dos diversos servidores que compõem o sistema distribuído do banco de dados, de forma análoga ao que foi descrito para bancos chave/valor. Se escolhermos o campo “nome” como chave do *sharding*, por exemplo, podemos ter uma divisão em que cada servidor armazena nomes que começam com determinada letra: nomes iniciados em “A” vão para um servidor, iniciados em “B” vão para outro, etc.

Quando a query é feita sobre esse campo escolhido, é sabido previamente qual nó conterá dados relevantes. De fato, se a query busca documentos cujo campo nome tem o valor “Olavo”, todos os possíveis resultados se encontram no servidor que armazena nomes iniciados na letra “O”, de maneira que a consulta pode ser executada apenas nesse servidor específico.

Documentos não são, todavia, compostos apenas de um campo. No uso real, queries vão se basear em campos que não necessariamente são chaves para o *sharding* – a chave pode ser um conjunto de campos, não apenas um – o que gerará a necessidade de executá-las em todos os nós, por não sabermos de antemão qual pode respondê-las.

O *tradeoff* que temos, portanto, é a perda de tolerância ao particionamento em situações específicas. Nos casos em que a consulta não pode ser limitada a um subconjunto de nós por não envolver campos escolhidos como chave para o *sharding*, a perda de comunicação com algum servidor pode provocar respostas parciais.

2.4.2 Neo4J

É um banco de dados NoSQL baseado em grafos. Suas características principais, assim como ocorreu na seção anterior e específica ao MongoDB, já foram descritas na seção 2.3.3, que trata sobre bancos desse tipo. Vamos comentar, então, aquilo que foi dito na seção mencionada sobre a capacidade de escalar de forma horizontal desse banco, assim como o que pode levar alguém a querer utilizá-lo em detrimento dos tradicionais bancos relacionais.

Grafos são estruturas que, de maneira geral, guardam dados com alto nível de conectividade. Foi dito, contudo, que a alta capacidade de escalar horizontalmente introduzida pelos bancos chave/valor é, em grande parte, devida ao isolamento dos dados que guarda. De fato, é intuitivo ver que quebrar algo pouco conectado em várias partes é uma tarefa muito mais simples do que subdividir uma intrincada rede de dependências. É natural, portanto, que um banco baseado numa estrutura altamente conectada não tenha como preocupação principal a capacidade de escalar horizontalmente – e com o Neo4J isso não é diferente.

Mas se o problema de escala não é resolvido, o que pode levar o desenvolvedor a basear sua aplicação em um banco de grafos ao invés de tabelas? A resposta pode ser resumida em duas palavras: facilidade e desempenho.

Voltemos ao exemplo da rede social – que é o mais óbvio para falar de armazenamento de informação em grafos. O modo como os dados são vistos pela aplicação – pessoas com características e relacionamentos diversos – não é semelhante ao que se pode guardar nas tabelas de um banco relacional. A maneira como relacionamentos são feitos – por meio de chaves estrangeiras – pode tornar o modelamento muito complexo, especialmente se os dados são altamente conectados, como no caso de exemplo. De fato, o projeto de bancos relacionais para aplicações de larga escala não é simples e envolve, muitas vezes, a necessidade do trabalho de um especialista.

Além da dificuldade de projeto em si, a maneira de recuperar esses dados e traduzi-los em informação útil para a aplicação também não é trivial. Queries complexas podem ser necessárias de maneira a obter informações que a aplicação usa rotineiramente. Isso tem implicações tanto na dificuldade de programá-las – exigindo novamente o especialista – como no custo de processamento para executá-las.

Uma operação *join*, através da qual dados de tabelas distintas – dados de um relacionamento – são agrupados de uma forma que faça sentido para a aplicação, pode ser muito intensa no processamento requerido. De fato, percorrer tabelas e uni-las atendendo a restrições dadas envolve a leitura de muitos dados do disco – o que é uma operação lenta.

Existem, sim, boas práticas no modo de programar queries de maneira a reduzir o impacto provocado por essas operações. Todavia, o modo como o banco baseado em grafos lida com isso tende a ser muito mais eficiente.

Como foi introduzido na seção 2.1.2, os dados são buscados de maneiras radicalmente diferentes em tabelas e grafos. Seja um relacionamento “é amigo de” entre duas pessoas. Em um banco relacional, as informações serão armazenadas da seguinte maneira:

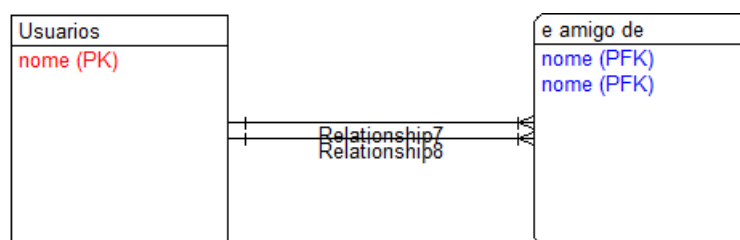


Figura 2 – Relacionamento “é amigo de” em um banco relacional

A tabela “Usuários” guarda todos os usuários cadastrados no sistema. O fato de uma pessoa ser amiga de outra é armazenado na tabela auxiliar “é amigo de”, por meio de suas identificações. Por exemplo, caso existam os usuários “Marcos” e “Vitor” e eles sejam amigos, então a tabela “Usuários” terá o registro de ambos, com uma linha para cada, e a tabela “é amigo de” terá uma linha com os nomes dos dois, caracterizando a relação de amizade.

O processo de obter todos os amigos de um usuário é feito da seguinte maneira: a tabela “é amigo de” é percorrida em busca das ocorrências do usuário cujos amigos desejamos descobrir. Para cada ocorrência, o outro nome envolvido – cada linha armazena dois nomes – é um amigo. De posse dos nomes dos amigos, então, a tabela de usuários é novamente percorrida em busca dos mesmos.

Perceba que, quando essas tabelas crescem muito, essas buscas tornam-se cada vez mais lentas. Tornar esse processo o mais rápido possível envolve cuidado com a ordem em que as restrições são aplicadas, assim como uso inteligente de índices.

Obter os amigos de um determinado usuário no Neo4J, contudo, é mais simples. Cada usuário está diretamente ligado a seus amigos através de arestas do grafo, podendo os mesmos serem atingidos em tempo de ordem constante (HARDY e WRIGHT, 1979). A dificuldade envolvida na recuperação dos dados também é reduzida. Nada de queries complexas, basta percorrer o grafo.

3 ABORDAGEM DE ACESSO A DADOS

Um framework é “*A skeleton of an application into which developers plug in their code and provides most of the common functionality*” (GAMMA, HELM, *et al.*, 1998). Em outras palavras, é algo que surge da semelhança entre problemas, é a solução da parte comum.

Um exemplo claro disso é facilmente obtido com relação a bancos de dados. Todas as aplicações que os usam devem acessá-los, de maneira que o problema de como lidar com o acesso se repete. Nesse contexto se insere o QueryBuilder, que propõe uma maneira diferente e fácil de recuperar os dados armazenados.

Antes de falar do framework desenvolvido, contudo, cabe descrever o que já existia antes de sua criação. As próximas seções tratarão do JDBC, do JPA e do acesso a bancos NoSQL, não atendidos pelos dois primeiros.

3.1 Execução Direta de Comandos

A primeira proposta para lidar com o acesso aos bancos por parte das aplicações é simples em seu conceito: prover a conexão com o banco, permitir o envio de instruções SQL ao mesmo e disponibilizar uma maneira de lidar com os resultados retornados. Um framework que segue essa filosofia é o JDBC.

O framework JDBC – *Java Database Connectivity* – proporciona conectividade entre a linguagem de programação Java e uma grande variedade de bancos de dados relacionais. Permite o estabelecimento de uma conexão com o banco a fim de enviar *statements* SQL e processar os resultados obtidos, encapsulando os mecanismos específicos de comunicação nos drivers de conexão desenvolvidos para cada banco. A camada de abstração que cria promove a portabilidade da aplicação entre várias bases de dados distintas, bastando, para tanto, que sejam substituídos os drivers.

De uma maneira geral, apenas permite que queries SQL no formato de *strings* possam ser invocadas programaticamente. Isso não exclui a possibilidade de cada banco ter suas particularidades no que tange à linguagem SQL utilizada. De fato, a portabilidade é obtida a nível de API, mas não a nível de query.

Para entender isso melhor, vamos lançar mão de um exemplo. Suponha que desejamos obter do banco de dados de uma empresa qualquer – que faz uso do software PostgreSQL – a

lista dos dez vendedores com melhores resultados acumulados no ano, para fins de pagamento de bônus. Isso seria feito, por meio de JDBC, da seguinte forma:

```
Class.forName("org.postgresql.Driver");
Connection con = DriverManager.getConnection("jdbc:postgresql://localhost/banco", "login", "senha");
Statement stm = con.createStatement();
ResultSet rs = stm.executeQuery("SELECT * FROM Vendedores ORDER BY valor_vendas DESC LIMIT 10");
```

Listagem 6 – Exemplo de uso do JDBC

Uma característica que pode provocar incômodo ao desenvolvedor é a forma como os dados são retornados: um *ResultSet*. Quaisquer que sejam as entidades buscadas, sejam os vendedores da nossa consulta-exemplo ou os produtos que os mesmos venderam, são retornadas dessa mesma maneira. Dessa forma, caso exista a necessidade de traduzi-los em objetos *Vendedor* ou *Produto*, possibilitando seu uso pela aplicação, ainda será necessário um outro passo: a extração dos valores contidos nesse *ResultSet* e sua transposição para os objetos de interesse.

```
List<Vendedor> listaVendedores = new LinkedList<Vendedor>();
while(rs.next()){
    String nome = rs.getString("nome");
    int valorVendido = rs.getInt("valor_vendas");
    Vendedor vendedor = new Vendedor();
    vendedor.setNome(nome);
    vendedor.setVendas(valorVendido);
    listaVendedores.add(vendedor);
}
```

Listagem 7 – Extração de resultados de um *ResultSet*

Agora, imagine que a administração da empresa resolveu, por motivos quaisquer, mudar o banco para o Microsoft SQL Server. Todo o código que trata da conexão ao banco se mantém, bastando alterar o driver de conexão para o referente ao novo banco utilizado. Todavia, mesmo com os ajustes realizados e conexão ao banco funcionando, toda a resposta que obteremos ao tentar executar essa mesma consulta no banco da Microsoft será uma mensagem de erro. Isso ocorre por um motivo simples: a query está errada! O correto seria:

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
Connection con = DriverManager.getConnection("jdbc:microsoft:sqlserver://localhost/ banco", "login", "senha");
Statement stm = con.createStatement();
ResultSet rs = stm.executeQuery("SELECT TOP 10 * FROM Vendedores ORDER BY valor_vendas DESC");
```

Listagem 8 – Exemplo de mudança de banco usando JDBC

Perceba que a portabilidade obtida, apesar de garantir comunicação com vários bancos diferentes por meio de drivers específicos, ainda se submete às possíveis diferenças na linguagem usada por cada um. A consulta, que é definida em uma *string* de forma separada ao código, não pode ser alterada automaticamente pelo JDBC de forma a se adequar ao banco alvo.

Outro aspecto que deixa a desejar é o fato de que o banco precisa ser previamente criado de forma manual – ou com auxílio de geradores de scripts – antes do uso pela aplicação, assim como suas tabelas, triggers, etc. Não existe automatização por parte da aplicação nesse sentido. Toda a dificuldade de traduzir objetos e suas relações – composição, agregação, herança, etc. – em dados armazenáveis em um banco relacional se renova a cada projeto.

3.2 Mapeamento Baseado em Metadados

Muito tempo era gasto pelos desenvolvedores no mapeamento do modelo orientado a objetos da aplicação no modelo entidade-relacionamento dos bancos de dados. A cada novo projeto, novas dificuldades para armazenar heranças, agregações e composições de maneira eficiente se mostravam presentes, consumindo um tempo considerável do processo de desenvolvimento. Recuperar esses dados também não era tarefa simples, exigindo escrita de várias queries.

Much of the code that deals with object-relational mapping describes how fields in the database correspond to fields in in-memory objects. The resulting code tends to be tedious and repetitive to write. A Metadata Mapping allows the developers to define the mappings in a simple tabular form, which can then be processed by generic code to carry out the details of reading, inserting, and updating the data. (FOWLER, RICE, *et al.*, 2002)

O padrão *Metadata Mapping* definido em (FOWLER, RICE, *et al.*, 2002) tem como objetivo aliviar o desenvolvedor da tarefa de reescrever toda a forma como atributos de objetos se relacionam com campos presentes no banco de dados. É proposta uma forma de fazer isso de maneira automática com uso de metadados – dados que explicam a relação entre os dados.

O primeiro framework *open-source* baseado nessas ideias a surgir foi o Hibernate. Seu objetivo era disponibilizar um mapeamento objeto-relacional, com base em metadados

presentes em arquivos xml, dos objetos da aplicação para as tabelas do banco de dados. Também contemplava relacionamentos e herança.

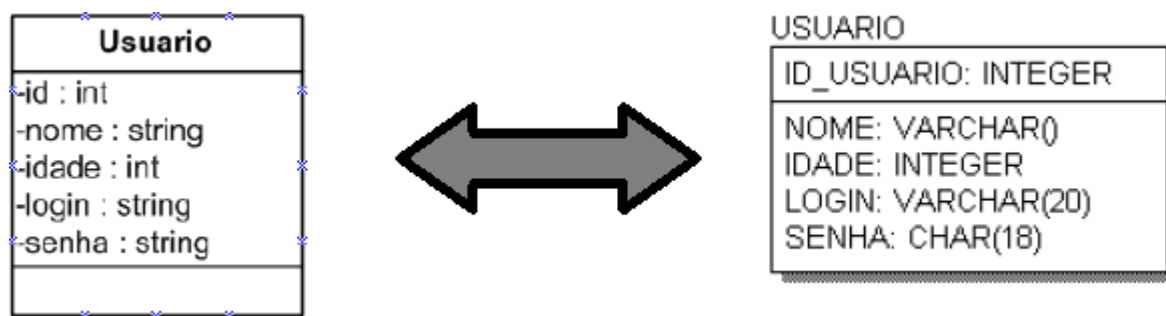


Figura 3 – Mapeamento objeto-relacional

Oferecia suporte a herança, relacionamentos com multiplicidade qualquer – um para um, um para muitos e muitos para muitos – tanto unidirecionais como bidirecionais e mecanismos para otimização da recuperação – estratégias *lazy* e *eager*.

A interface de persistência que proporcionou fazia com que os problemas de mapeamento mencionados, verdadeiros tomadores de tempo, pudessem ser resolvidos de maneira automática – um salto de produtividade. Por esse motivo, com a introdução das *annotations* no JDK 5.0, foi criado um padrão para a estratégia de mapeamento objeto-relacional: o JPA.

O *Java Persistence Application Programming Interface*, ou JPA, foi além dos arquivos xml e introduziu o uso das anotações para definir o novo padrão de persistência da linguagem Java. A facilidade proporcionada por essa forma de programar propiciou um ganho ainda maior de produtividade.

Ainda, foi definida uma linguagem comum para a criação de queries, o que elevou a portabilidade ao nível não só da API, mas também das consultas. Essa linguagem é traduzida nos dialetos específicos de cada banco de maneira automática por meio de implementações específicas a cada base.

Voltando ao exemplo anterior, fazer a consulta por meio de JPA envolveria os seguintes passos: anotar devidamente a classe representativa de um vendedor para que o mapeamento funcione, algumas configurações com relação à comunicação com a base de dados e a realização da consulta em si. Abaixo, suas execuções:


```

@Entity
@Table(name="Vendedores")
public class Vendedor{
    @Id
    String nome;
    int valor_vendas;
}

```

Listagem 9 – Exemplo de uso do JPA: anotando a classe representativa da entidade

```

public class JPAUtil {
    private static EntityManagerFactory emf;
    static{
        emf = Persistence.createEntityManagerFactory("vendedores");
    }
    public static EntityManager getEntityManager(){
        return emf.createEntityManager();
    }
}

```

Listagem 10 – Exemplo de uso do JPA: configurando a comunicação por meio do *EntityManager*

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
(...)
    <persistence-unit name="vendedores" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>Vendedor</class>
        <properties>
            <property name="hibernate.connection.url"
value="jdbc:postgresql://localhost/eleicoes" />
            <property name="hibernate.connection.driver_class"
value="org.postgresql.Driver" />
            <property name="hibernate.connection.password" value="senha" />
            <property name="hibernate.connection.username" value="usuario" />
            <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider" />
            <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
        </properties>
    </persistence-unit>
</persistence>

```

Listagem 11 – Exemplo de uso do JPA: arquivo xml de configuração

```
String sql = "SELECT ven FROM Vendedores ven ORDER BY ven.valor_vendas DESC";
EntityManager em = JPAUtil.getEntityManager();
Query query = em.createQuery(sql);
query.setMaxResults(10);
List<Vendedor> result = query.getResultList();
```

Listagem 12 – Exemplo de uso do JPA: realização da consulta

As únicas mudanças que seriam necessárias de maneira a fazer esse código funcionar no banco Microsoft SQL Server seriam algumas propriedades da conexão especificadas no arquivo xml, como o driver e o dialeto utilizados. A query " SELECT ven FROM Vendedores ven ORDER BY ven.valor_vendas DESC " não sofre qualquer tipo de alteração, pois a API se encarrega de traduzi-la ao dialeto especificado.

Outro ponto a se notar é o fato de a API já retornar uma lista de objetos Vendedor prontos para uso, sem a enfadonha necessidade de definir os seus campos individualmente como foi feito em JDBC.

Todavia, a padronização conseguida pelo JPA não se preocupou com bancos NoSQL, talvez pelo fato de os mesmos não terem muita visibilidade no momento de seu desenvolvimento. Agora, contudo, o movimento ganha cada vez mais força, de maneira que é desejável tal extensão.

3.3 Acesso a Bancos NoSQL

As seções anteriores apresentaram soluções para o problema do acesso aos bancos de dados por uma aplicação, de modo que o leitor desavisado poderia crer que o advento do JPA pôs um fim definitivo às dificuldades enfrentadas pelo desenvolvedor que precisa lidar com a persistência de informações. Todavia, nem o JDBC nem o JPA contemplam todos os bancos existentes. De fato, talvez por sua pouca visibilidade na época do desenvolvimento dessas interfaces, toda a classe dos bancos NoSQL foi deixada de lado.

O acesso aos dados baseia-se, então, apenas nas inúmeras – e heterogêneas – interfaces providas por cada banco. Um verdadeiro desastre tanto no tocante à portabilidade quanto à manutenção. Até mesmo o desenvolvimento é impactado, visto que cada banco demanda conhecimentos específicos do programador.

A fim de tornar isso evidente, vejamos como executar aquela mesma consulta quanto a vendedores tanto no MongoDB – fazendo uso do framework Morphia – quanto no Neo4J.

```

Morphia morphia = new Morphia();
morphia.map(Vendedor.class);
Mongo mongo = new Mongo(serverAddress, serverPort);
Datastore ds = morphia.createDatastore(mongo, "testDatabase");
Query query = ds.createQuery(Vendedor.class).order("-valor_vendas").limit(10);
List<Vendedor> = query.asList();

```

Listagem 13 – Exemplo de query no MongoDB fazendo uso do Morphia

```

GraphDatabaseService graphDB = new GraphDatabaseFactory().newEmbeddedDatabase(databasePath);
String queryString =
Iterator<Node> i = graphDB.index().forNodes(indexName).query(queryString).iterator();
List<Vendedor> vendedores = new LinkedList<Vendedor>();
while(i.hasNext()){
    Node node = i.next();
    Vendedor vendedor = new Vendedor();
    vendedor.setNome((String) node.getProperty("nome"));
    vendedor.setVendas((int) node.getProperty("valor_vendas"));
    vendedores.add(vendedor);
}

```

Listagem 14 – Exemplo simplificado de query no Neo4J

Perceba que o código das consultas não apresenta qualquer semelhança, tanto entre si como com os códigos apresentados para JDBC e JPA. Qualquer aplicação que se baseie em um desses bancos sofreria muito com a necessidade de mudança, visto que praticamente todo o código de persistência deveria ser refeito. Ainda, atente ao fato de que a consulta no Neo4J, diferentemente das demais, não retorna apenas os dez vendedores mais bem colocados em relação às suas vendas. De fato, a ordenação não é suportada a nível de banco, devendo ser tratada pela aplicação.

3.4 Esfinge QueryBuilder

O Esfinge QueryBuilder oferece uma solução para a criação de uma camada de persistência de forma simples e rápida. Através da filosofia “para um bom framework, o nome do método basta”, o QueryBuilder utiliza os nomes dos métodos de uma interface para inferir as consultas que precisam ser executadas na base de dados.

Para que isso ocorra, contudo, é necessário que exista previamente um mapeamento automático entre os objetos da linguagem Java e as entidades armazenadas nas bases de dados – sejam essas relacionais, orientadas a documentos, a grafos, etc. – de maneira que a comunicação entre a aplicação e o banco não encontre, na forma como descrevem seus dados, uma barreira.

Antes desse Trabalho de Graduação, suportava apenas bancos de dados relacionais, baseando-se tanto em JDBC como em JPA. Agora, contudo, passará a contemplar, também, os bancos NoSQL MongoDB, orientado a documentos, e Neo4J, baseado em grafos.

Funciona de uma maneira simples. Deve ser criada uma interface com todos os métodos de acesso que se deseja usar, os quais devem obrigatoriamente obedecer a uma sintaxe pré-definida.

O framework interpreta o nome em tempo de execução e cria uma query que é executada no banco. O método *getPersonByLastName()*, por exemplo, retorna instâncias da classe *Person* filtrando pela propriedade *lastName*, o método *getPersonByAddressCity()* faz a consulta filtrando pela propriedade *city* da propriedade *address*, que é também uma classe persistente, etc.

Para instanciar a interface definida, basta utilizar o método *create* da classe *QueryBuilder*, como segue:

```
PersonDAO dao = QueryBuilder.create(PersonDAO.class);
```

Foi dito que uma sintaxe deve, obviamente, ser obedecida a fim de que o programa seja capaz de interpretar corretamente as intenções do usuário. Abaixo são listadas as regras que a regem:

- Os métodos devem começar com *get* e serem seguidos do nome da entidade. Exemplo: *getPerson()*.
- Para passar parâmetros para a consulta, o nome da entidade deve ser seguido por *by* e por nomes de propriedades da classe. O parâmetro deve ser do mesmo tipo da propriedade. Exemplo: *getPersonByName(String name)* e *getPersonByLastName(String lastName)*.
- Os métodos podem retornar uma instância da entidade ou uma lista de instâncias da entidade, como os dois exemplos anteriores apresentados. Exemplo: *Person getPerson()* ou *List<Person> getPerson()*.
- O parâmetro passado pode navegar entre as dependências da classe e acessar propriedades delas. Exemplo: *getPersonByAddressCity(String city)*, que filtraria a propriedade *city* na propriedade *address*.
- Para passar mais de um parâmetro pode-se utilizar *and* ou *or* entre as propriedades. Os parâmetros serão considerados na mesma ordem definida no nome. Exemplo: *getPersonByNameAndLastName(String name, String lastname)* e *getPersonByNameOrLastName(String name, String lastname)*.

- Para comparações com relação que não seja de igualdade – maior, maior ou igual, menor, menor ou igual, diferente – existem duas abordagens possíveis. Pode ser inserida uma anotação antes do argumento do método ou o tipo de comparação pode ser inserido logo após a propriedade, no nome do método. Exemplos: `getPersonByAge(@Greater Integer age)`, `getPersonByAgeGreater(Integer age)`.
- Também é possível compor o nome dos métodos de forma que as consultas sejam ordenadas. Para isso é necessário no final do nome adicionar *OrderBy* seguido pelo nome da propriedade. Caso seja necessária a ordenação por dois campos, pode-se separá-los com o conector *and*. Também é possível definir o sentido da ordenação com a adição de “Asc” ou “Desc” após o nome da propriedade, sendo ascendente o valor padrão. Exemplos: `getPersonOrderByName()`, `getPersonOrderByNameAndLastName()`, `getPersonOrderByAgeDesc()`.

Quanto ao projeto do framework, possui design modular de maneira a facilitar sua extensão aos mais diversos bancos de dados que possam surgir. Observe a estrutura UML (OBJECT MANAGEMENT GROUP) simplificada na Figura 4.

Nessa abordagem o framework não é único. De fato, existe o Spring Data (SPRING) que também se baseia em design modular para prover uma API uniforme de acesso aos mais diversos bancos de dados, incluindo em seu conjunto de suporte bancos NoSQL. O diferencial do QueryBuilder é, no entanto, sua capacidade de gerar consultas automaticamente em tempo de execução, que não é compartilhada por nenhuma outra solução disponível.

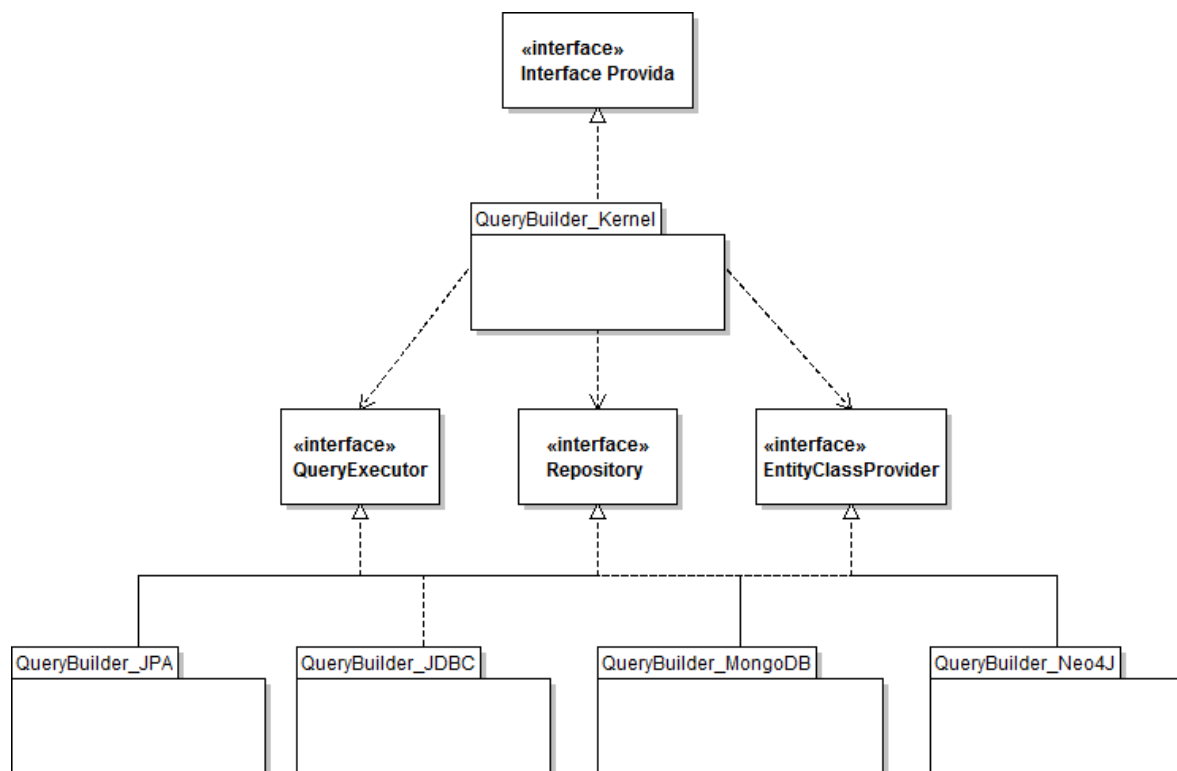


Figura 4 – Módulos do Framework Esfinge QueryBuilder

Perceba que toda a comunicação do QueryBuilder_Kernel com os módulos responsáveis pela comunicação com os bancos de dados é feita por meio de três interfaces pré-definidas. Dessa forma, caso surja a necessidade de torná-lo compatível com qualquer outro tipo de base de dados que possa surgir, bastará criar o módulo específico para esse banco, cuja única exigência é a implementação dessas três interfaces e seus métodos.

Todo o trabalho comum aos diferentes bancos, ou seja, tudo o relacionado ao processo de *parsing* dos nomes dos métodos a fim de definir o que deve ser efetivamente buscado é feito pelo Kernel, sobrando aos outros módulos apenas o comportamento específico a cada base.

Como foi dito anteriormente, esse Trabalho de Graduação tem seu foco na extensão do Esfinge QueryBuilder a bancos NoSQL ou, mais especificamente, aos bancos MongoDB e Neo4J. Dessa forma, serão desenvolvidos os módulos referentes a cada um deles, com a implementação das três interfaces citadas, de maneira a garantir o funcionamento em conjunto com o Kernel.

4 EXTENSÃO DO QUERYBUILDER PARA NoSQL

A presente seção visa o esclarecimento do leitor quanto ao modo como a extensão do framework Esfinge QueryBuilder aos bancos de dados NoSQL escolhidos – MongoDB e Neo4J – foi realizada, assim como instruí-lo quanto ao seu uso.

São explorados os aspectos relevantes de como a comunicação entre o kernel e os módulos se dá, com a devida descrição das interfaces que a definem – seus métodos e o que de fato devem realizar.

Além dessa análise, que é comum a ambos os módulos, também são expostas particularidades de cada implementação, inerentes ao fato de os bancos serem diferentes e precisarem ser tratados de maneira diferente. É discutida a pré-existência de um framework para mapeamento de objetos Java em documentos do MongoDB, tomado como base na implementação para esse banco, e a completa ausência de algo nesse sentido para o Neo4J, o que levou à necessidade de sua criação.

4.1 As Interfaces de Comunicação

O framework Esfinge QueryBuilder foi projetado de maneira a ser facilmente extensível. De fato, como a Figura 4 evidencia, sua estrutura é modular, com as funcionalidades básicas e comuns a todos os bancos sendo cumpridas pelo núcleo do programa, enquanto o acesso específico a cada base de dados, com suas idiossincrasias, é devidamente tratado por cada módulo.

Como a mesma figura mostra, a comunicação entre o núcleo e os módulos dos bancos é especificada por três interfaces, que englobam todas as funcionalidades que devem ser desempenhadas. São elas: *QueryExecutor*, *Repository* e *EntityClassProvider*. As extensões desenvolvidas, para os bancos NoSQL MongoDB e Neo4J, tiveram seu projeto baseado nas necessidades impostas por tais interfaces.

A interface *QueryExecutor* define um único método, *Object executeQuery(QueryInfo info, Object[] args)* que, recebendo do núcleo do programa um objeto *QueryInfo* – detentor das informações relevantes à montagem da query – e os valores desejados para cada atributo – presentes no *array* de objetos *args* – retorna um resultado, seja este uma entidade de

persistência propriamente dita ou uma lista delas, caso o resultado não seja único. As informações contidas no objeto *QueryInfo* e a forma de usá-las serão discutidas em breve.

```
public interface QueryExecutor {
    public Object executeQuery(QueryInfo info, Object[] args);
}
```

Listagem 15 – Definição da interface *QueryExecutor*

A interface *Repository* define uma interface padrão para o uso mais rotineiro de cada banco, caso o usuário do framework tenha seu interesse centrado em funcionalidades CRUD – Create, Read, Update, Delete – não se limitando, contudo, a apenas essas quatro operações básicas. A seguir, a definição da interface, com a assinatura de seus métodos:

```
public interface Repository<E> {
    public E save(E obj);
    public void delete(Object id);
    public List<E> list();
    public E getById(Object id);
    public List<E> queryByExample(E obj);
}
```

Listagem 16 - Definição da interface *Repository*

Perceba que existe um método nessa interface que vai além do simples CRUD: o *queryByExample*. Ele está disponibilizado para consultas que, por envolverem muitos parâmetros, são inviáveis de definir através de simples nomeação de métodos. Seu funcionamento é o que o nome sugere: passando um objeto cujas propriedades tenham nomes correspondentes àqueles que se deseja buscar, a consulta é realizada. Um exemplo de *QueryObject*, nome dado a um tal objeto, pode ser visto na Listagem 18, página 47.

Existem três maneiras diferentes de definir o tipo de comparação desejada para cada propriedade, estando todas elas presentes no exemplo dado. De fato, isso pode ser feito por meio de anotações na declaração da propriedade, anotações no método *getter* da propriedade ou, finalmente, adição ao próprio nome da propriedade – como *ageGreater* e *ageLesser*, do exemplo.

Um outro ponto a se notar é que ambas as funcionalidades *Create* e *Update* estão fundidas em um único método: o método *save*. De fato, seu funcionamento é tal que, caso o objeto que se intenciona salvar não exista, ele é criado; caso já esteja presente no banco, é atualizado com as propriedades passadas.

A interface *EntityClassProvider* existe para suprir a necessidade do framework de ser capaz de traduzir o nome de uma entidade fornecido nos métodos por uma classe Java que a represente. Sua definição é extremamente simples e engloba apenas um método, como pode ser visto abaixo:

```
public interface EntityClassProvider{  
    public Class<?> getEntityClass(String name);  
}
```

Listagem 17 - Definição da interface *EntityClassProvider*

Dessa forma, a tarefa se resume a implementar todos esses métodos requeridos pelas interfaces apresentadas de acordo com as características de cada banco em questão, tanto o MongoDB quanto o Neo4J.

```

public class ExampleQueryObject {
    private Integer ageGreater;
    private Integer ageLesser;
    @Contains
    private String name;
    private String lastName;

    public Integer getAgeGreater() {
        return ageGreater;
    }
    public void setAgeGreater(Integer ageGreater) {
        this.ageGreater = ageGreater;
    }
    public Integer getAgeLesser() {
        return ageLesser;
    }
    public void setAgeLesser(Integer ageLesser) {
        this.ageLesser = ageLesser;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Contains
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}

```

Listagem 18 – Exemplo de *QueryObject*

4.2 MongoDB

Apesar de terem os mesmos objetivos finais em mente, as implementações diferem em alguns aspectos, principalmente devido à pré-existência de um mapeamento objeto-relacional para o banco MongoDB, que possibilitou a redução do trabalho envolvido em sua criação. O módulo referente a esse banco teve seu desenvolvimento apoiado no projeto Morphia (A type-safe Java library for MongoDB), que já provia as funcionalidades de tradução de objetos Java de/para o banco.

4.2.1 Configuração e Uso

Existem, para o uso do QueryBuilder com qualquer dos seus bancos de dados suportados, aqueles passos comuns já mencionados – como criação de uma interface com os métodos de acesso desejados, devidamente nomeados. Essa seção, assim como sua equivalente para o banco Neo4J, objetiva instruir o usuário acerca dos pormenores da configuração específica ao módulo do banco MongoDB, não contemplando aqueles passos, portanto.

O usuário deverá implementar a interface *DatastoreProvider* e registrá-la para uso através de um arquivo de configuração. A implementação da interface é tarefa simples e serve para, basicamente, informar ao framework quais classes desejamos persistir – ou recuperar – da base de dados.

Um exemplo de implementação da interface *DatastoreProvider* é apresentado na Listagem 19. Suas partes que devem ser alteradas de acordo com a necessidade do usuário se resumem a:

- Comandos referentes ao mapeamento – ao invés das classes *Person* e *Address*, do exemplo, deverão ser utilizadas as classes para as quais se deseja a funcionalidade do framework.
- A localização do banco de dados – a string “localhost” deve ser substituída pela localização correta, caso o banco não esteja na máquina em que a aplicação é executada.
- A porta de comunicação com a aplicação banco de dados – a porta deve ser corrigida caso o banco seja configurado de maneira diferente e não use a porta padrão.
- O nome do banco que será acessado pela aplicação – no caso de exemplo, “testdb”.

```

@ServicePriority(1)
public class TestMongoDBDatastoreProvider extends DatastoreProvider{

    public TestMongoDBDatastoreProvider(){
        try {
            mongo = new Mongo("localhost", 27017);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (MongoException e) {
            e.printStackTrace();
        }

        getMorphia().map(Person.class);
        getMorphia().map(Address.class);
    }

    @Override
    public Datastore getDatastore() {
        return getMorphia().createDatastore(mongo, "testdb");
    }
}

```

Listagem 19 – Exemplo de implementação da interface *DatastoreProvider* para o banco *MongoDB*

Com a implementação da interface *DatastoreProvider* feita, resta uma configuração extra de maneira a permitir que o framework saiba qual classe utilizar para esse papel. Isso é tarefa simples e rápida. Para tanto, basta criar um arquivo com o nome “org.esfinge.querybuilder.mongodb.DatastoreProvider” em uma pasta META-INF/services localizada na pasta com o código-fonte do projeto. Esse arquivo deve conter apenas uma linha com o nome da classe criada que implementa a interface, por exemplo, “org.esfinge.querybuilder.mongodb.TestMongoDBDatastoreProvider”.

A anotação *ServicePriority* tem relação com esse processo de registro da classe para uso por parte do framework, descrito no parágrafo anterior. Várias classes podem ser definidas para tomar o papel de uma mesma interface, com prioridades diferentes. Aquela com prioridade mais alta será a efetivamente utilizada.

As classes que representam entidades de persistência, ou seja, aquelas que desejamos persistir e recuperar do banco de dados, as mesmas que acabamos de informar na implementação da interface *DatastoreProvider*, devem ser devidamente anotadas. Como o framework Morphia é a base do mapeamento objeto-relacional, as anotações utilizadas são as que ele define, as quais podem variar de versão em versão, com possível adição de algumas e

exclusão de outras. Uma visita à documentação atualizada é recomendada: <http://code.google.com/p/morphia/wiki/AllAnnotations>.

Faz-se necessário, ainda, adicionar os arquivos jar referentes tanto ao framework Morphia como ao software MongoDB, assim como aquele que contém o kernel do framework QueryBuilder.

Com os passos descritos acima devidamente realizados, o software está pronto para ser utilizado.

4.2.2 Implementação

A implementação das interfaces requeridas pelo núcleo do framework QueryBuilder foi, para o MongoDB, baseada nesse projeto pré-existente denominado Morphia. Cabe então, antes de expor o que foi feito, apresentar o que o mesmo disponibiliza, mesmo que de maneira resumida e simples, de forma a dar base ao seguinte.

Seu mapeamento é puramente baseado em anotações, sem necessidade de arquivos extras – como xml, por exemplo. Cada classe que será uma entidade de persistência e, portanto, terá existência no banco condicionada à sua tradução a uma linguagem que o mesmo compreenda, deverá ser registrada na classe Morphia antes de seu uso.

A maneira de utilizar esse framework pode ser resumida através do exemplo simples ilustrado nas figuras a seguir. Perceba que a classe de acesso ao banco de dados, tanto para fazer inserções como recuperações, é a classe *Datastore*.

```
public void MorphiaUse(){

    Morphia morphia = new Morphia();
    Datastore ds = morphia.createDatastore(new Mongo("localhost", 27017), "testDB");
    morphia.map(Employee.class);

    ds.save(new Employee("John"));

    Employee boss = ds.find(Employee.class).field("manager").equal(null).get();

}
```

Listagem 20 – Exemplo de uso do framework Morphia

```

@Entity("employees")
class Employee {

    @Id ObjectId id; // auto-gerado se não for especificado
    String firstName, lastName; // salvos automaticamente
    Long salary = null; // apenas valores não nulos são salvos

    Address address; // por default, é inserido como campo do documento, não como referência a
    outro documento

    Key<Employee> manager; //referências podem ser mantidas sem carregamento automático
    @Reference List<Employee> underlings = new ArrayList<Employee>(); //referências salvas e
    carregadas automaticamente

    @Property("started") Date startDate; //campos podem ter nomes especificados
    @Property("left") Date endDate;

    @Indexed boolean active = false; //podem ser criados índices para melhor performance
    @NotSaved String readButNotStored; //campos podem ser carregados, mas não salvos
    @Transient int notStored; //campos podem ser ignorados, nem carregados nem salvos

}

```

Listagem 21 – Exemplo de uso do framework Morphia: definição de uma entidade de persistência

A maneira mais clara de tratar do que foi implementado é apresentar o trabalho em função das interfaces requeridas desenvolvidas. Dessa forma, essa será a abordagem tomada, tanto nessa seção como na equivalente referente ao banco Neo4J.

A interface *EntityClassProvider* tem como responsabilidade implementar apenas um método que traduz nomes de entidades em classes Java. Felizmente, o framework Morphia mantém um mapa de todas as classes que são registradas nele para uso com o banco, utilizando como chave justamente o nome das classes. Dessa forma, a implementação dessa interface é simples e se limita a buscar no Morphia a informação desejada.

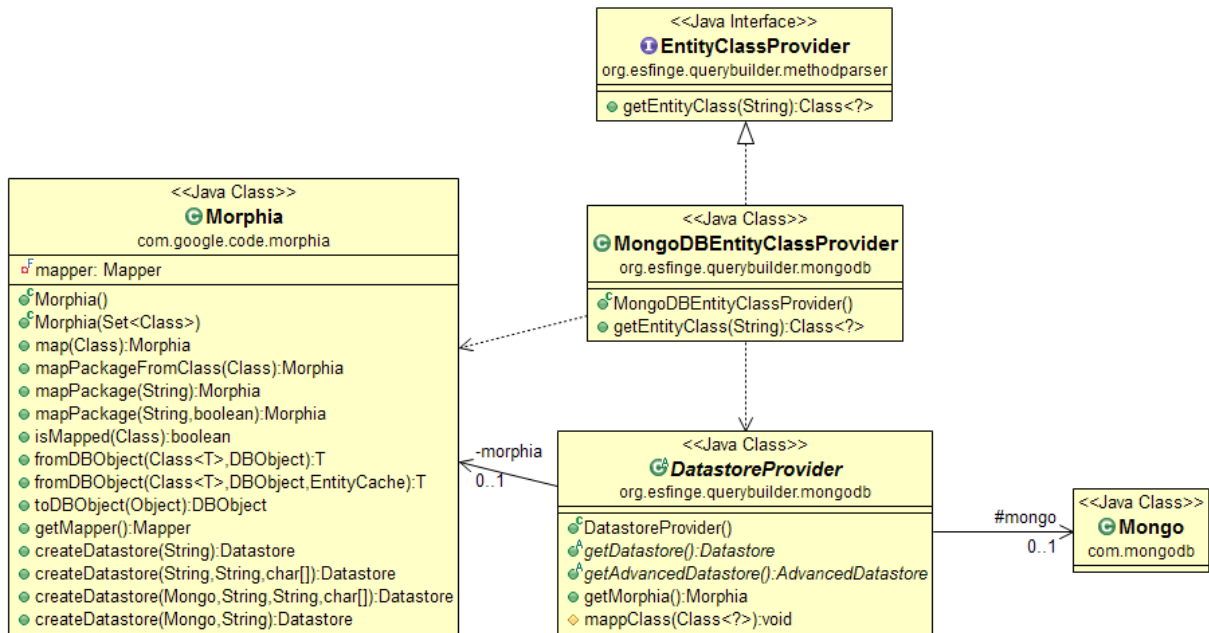


Figura 5 – Diagrama de classes referente à implementação da interface *EntityClassProvider*

A dependência expressa no diagrama acima quanto à interface *DatastoreProvider* se justifica pelo fato de ser essa a responsável pelo acesso da aplicação à instância do Morphia utilizada. De fato, foi feita a opção por manter uma única instância Morphia para toda a aplicação, tanto para evitar a necessidade de refazer as configurações de mapeamento para cada nova instância como para não provocar perdas de performance desnecessárias.

A interface *Repository* também não exige muito trabalho em cima do que é fornecido pelo framework Morphia. Os métodos CRUD são atendidos quase sem necessidade de mudança, fazendo uso da classe de acesso *Datastore*. O que merece maior atenção nesse ponto é a implementação do método *QueryByExample*, que necessitou do uso de *Java Reflection Language*. Veja:

```

@Override
public List<E> queryByExample(E obj) {
    Class<?> clazz = obj.getClass();
    Query query = ds.createQuery(clazz);

    try {
        for (Method m : clazz.getMethods()) {
            if (!m.getName().equals("getClass")
                && MongoDBDAOUtils.isGetterWhichIsNotTransient(m, clazz)) {
                Object value = m.invoke(obj);
                if (value != null && !value.toString().trim().equals("")) {
                    String prop = m.getName().substring(3, 4).toLowerCase() +
                        m.getName().substring(4);

                    query.field(prop).equal(value);
                }
            }
        }
    } catch (Exception e) {
        throw new InvalidPropertyException("Error building query", e);
    }

    return query.asList();
}

```

Listagem 22 – Implementação do método *QueryByExample* da interface *Repository* para o banco MongoDB

A lógica é simples: o objeto de exemplo recebido tem seus métodos percorridos à procura de *getters*. Encontrando-os, são invocados de maneira a obter os valores dos atributos contidos no objeto passado como parâmetro. O nome dos campos aos quais se referem também são extraídos. Em seguida, a condição “atributo = valor obtido” é adicionada à query que é retornada no fim da execução.

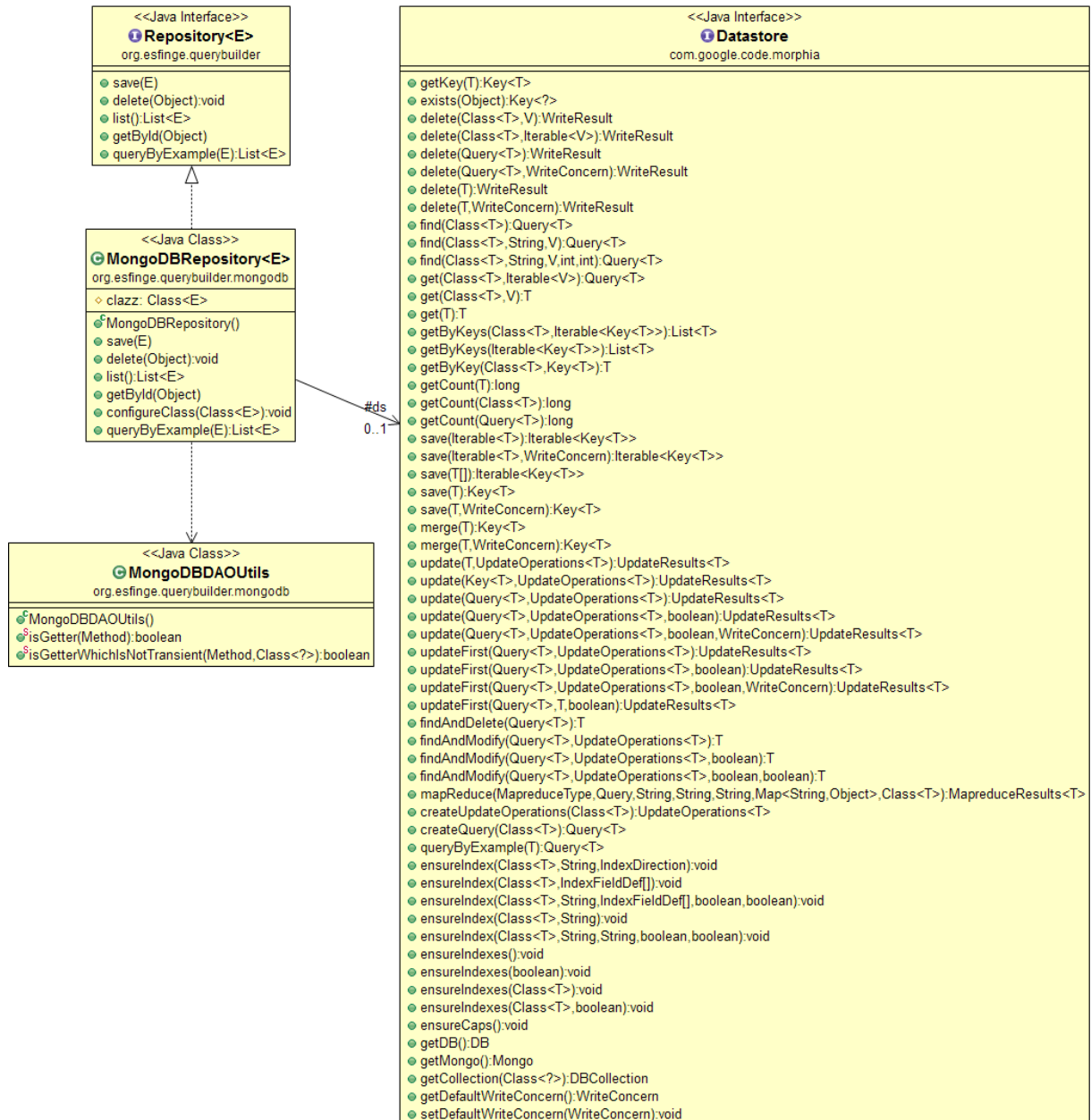


Figura 6 - Diagrama de classes referente à implementação da interface *Repository*

A interface *QueryExecutor* foi onde se concentrou a maior parte do trabalho de implementação. Isso era de fato esperado, visto que é a espinha dorsal do funcionamento do framework. Vamos começar observando o diagrama de classes que evidencia suas dependências:

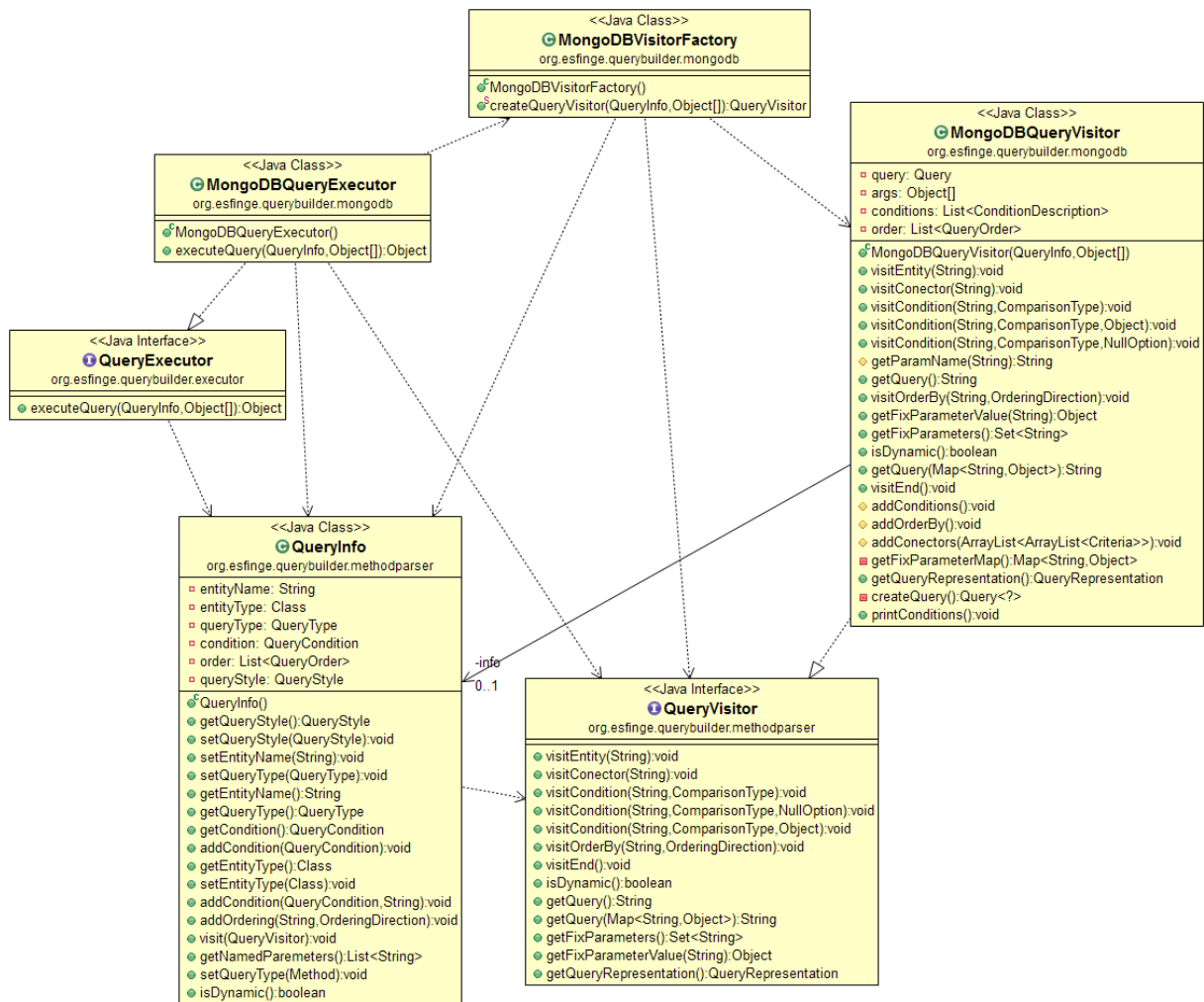


Figura 7 - Diagrama de classes referente à implementação da interface *QueryExecutor*

Podemos ver que a classe *MongoDBQueryExecutor* depende tanto das classes *QueryInfo* quanto *QueryVisitor*. A dependência em relação à primeira se justifica pelo óbvio: *QueryInfo* é a classe detentora de todas as informações necessárias à montagem a query, como a entidade a ser buscada, valores de atributos, etc. A figura do *QueryVisitor*, contudo, não tem razão tão evidente. Observando o diagrama com cuidado, podemos ver que a presença da interface *QueryVisitor* se faz presente no kernel do framework, não sendo particularidade do módulo específico ao MongoDB. De fato, a maneira como as queries são montadas é amarrada pelo kernel, que faz uso do padrão de projeto *visitor* (GAMMA, HELM, *et al.*, 1998).

Esse padrão de projeto permite que as operações relacionadas à montagem da query, que são particulares para cada banco, possam ser alteradas sem que exista a necessidade de modificar as classes que definem sua estrutura. Todas essas operações são concentradas no objeto *Visitor*, que se torna o responsável pela criação da query de maneira adequada para cada banco de dados distinto.

O uso desse padrão que permite o desacoplamento se mostrou necessário para que o framework seja facilmente extensível a várias bases de dados, pois o kernel deve ser capaz de montar queries para todos os bancos da mesma maneira, ou seja, o processo de interpretação dos nomes dos métodos deve ser desacoplado da geração da query em si.

Dessa forma, a implementação da interface *QueryVisitor*, *MongoDBQueryVisitor*, concentra a maior parte do esforço de desenvolvimento, pois deve ser feita de forma a prover as operações definidas na interface de uma maneira aceita pelo banco.

4.2.3 Limitações e Dificuldades

O fato de a implementação se basear no framework Morphia impõe uma restrição quanto à possibilidade de criar um cache com as queries recém-realizadas. Seria interessante que uma consulta que difere unicamente nos valores buscados de uma outra já realizada pudesse reutilizá-la, de maneira a evitar todo o processo de interpretação e geração e promover um tempo de resposta menor.

Imagine um banco de dados que armazena informações dos clientes de uma empresa. Suponha que, para uma finalidade qualquer, é necessário obter uma lista de todos aqueles que moram em determinado bairro, são maiores de sessenta e dois anos e fizeram uma compra no último mês. Perceba que, enquanto o bairro em questão pode variar, com queries possivelmente abrangendo todos os bairros da cidade, a parte referente à idade e à compra é a mesma. A estrutura da query se mantém. Um cache que armazenasse essa estrutura para que o bairro em questão fosse especificado de forma subsequente pouparia o processo de construir toda a query novamente.

Todavia, a API exposta pelo framework Morphia não dá margem para a manutenção de tal cache. Não é como o caso dos bancos relacionais, em que a query é representada por uma string e é fácil alterar apenas os valores dos atributos sem ser necessária qualquer mudança na query em si.

4.3 Neo4J

De forma diferente ao que ocorreu no desenvolvimento do módulo para o banco MongoDB, não existia qualquer trabalho acerca de um mapeamento objeto-relacional para o banco Neo4J. Apenas a API exposta pelo banco em si se fazia disponível, com um tratamento

dos dados em baixo nível. Foi necessário criar, do zero, um mapeador baseado em *Java Reflection Language*. Isso será comentado na seção 4.3.2.

4.3.1 Configuração e Uso

De maneira semelhante ao que foi feito na configuração do banco MongoDB, aqui também se faz necessário implementar a interface *DatastoreProvider*, sendo nessa implementação que são especificadas quais classes deverão ser tratadas como entidades de persistência, ou seja, as classes para as quais desejamos o mapeamento objeto-relacional. Um exemplo de tal implementação segue:

```
@ServicePriority(1)
public class TestDatastoreProvider implements DatastoreProvider{
    private static final Neo4J neo4j = new Neo4J(dataBasePath);
    public TestDatastoreProvider(){
        neo4j.map(Person.class);
    }
    @Override
    public Neo4J getDatastore() {
        return neo4j;
    }
}
```

Listagem 23 – Exemplo de implementação da interface *DatastoreProvider* para o banco Neo4J

Também de forma análoga ao que já foi descrito para o MongoDB – as semelhanças não são mera coincidência –, deve ser criado um arquivo com o nome da interface implementada “org.esfinge.querybuilder.neo4j.DatastoreProvider” em uma pasta META-INF/services localizada no diretório em que se encontra o código-fonte. Esse arquivo deve conter apenas a classe que implementa a interface: por exemplo, “TestDatastoreProvider”.

Todos os arquivos jar referentes ao banco Neo4J, ao kernel do framework QueryBuilder e ao módulo do banco Neo4J – que inclui o mapeador objeto-relacional – devem ser adicionados às dependências do projeto.

As classes que se deseja utilizar como entidades de persistência devem receber as anotações devidas, de maneira que o mapeamento possa ser realizado corretamente. Não são muitas as anotações definidas pelo mapeamento objeto-relacional criado, visto que esse não tinha a pretensão de ser nada muito elaborado, mas apenas o necessário para o funcionamento em conjunto com o framework. São elas:

- @Id – marca um atributo como id.
- @Indexed – informa o mapeador que se deseja fazer consultas baseadas no valor desse atributo e, portanto, deve ser mantido um índice para ele.
- @NodeEntity – usado para anotar uma classe, informa que a mesma é uma entidade de persistência.
- @RelatedTo – indica um relacionamento com uma outra entidade de persistência; recebe um parâmetro *targetClass* que informa a classe dessa outra entidade. Aceita que o atributo anotado seja do tipo *Set*, a fim de permitir relacionamentos com multiplicidade diferente de um.

4.3.2 Implementação

De maneira diferente ao que ocorreu com o banco MongoDB, aqui foi necessário desenvolver do zero um mapeamento objeto-relacional para ser usado em conjunto com o framework. O mapeamento não pretende, contudo, ser muito elaborado: apenas provê o necessário às funcionalidades do QueryBuilder. Vejamos sua estrutura através de um diagrama de classe, exibido na Figura 8.

As classes com as quais o usuário efetivamente tem contado são: *Neo4J*, *Query<E>* e *Condition*. De fato, a primeira é aquela responsável por expor todas as funcionalidades básicas, como a possibilidade de mapear uma classe qualquer para uso com o banco, funcionalidades CRUD e a montagem de queries. A segunda é responsável por agrupar as diferentes condições que o usuário deseje impor à sua consulta, assim como efetivamente executá-la. Não deve ser instanciada através do seu construtor; o método *query* da classe *Neo4J* deve ser utilizado a fim de gerar uma instância com todas as informações corretas e necessárias, que só essa classe contém. A terceira representa a condição em si, o atributo e o valor desejado, por exemplo. Também pode ser utilizada para representar condições agrupadas por conectores lógicos.

A interface foi desenvolvida para que a escrita de uma query seja de fácil confecção e compreensão. Veja alguns exemplos:

```
Query<Person> q = neo.query(Person.class);
q.or(new Condition("idade", 14, ComparisonType.GREATER), new Condition("nome", "João"));
List<Person> results = q.asList();
```

Listagem 24 – Exemplo de criação de query “idade > 14 ou nome = João” no mapeador objeto-relacional criado

```

Query<Person> q = neo.query(Person.class);
q.or(new Condition("idade", 14), new Condition(ConditionType.AND, new Condition("nome", "João"),
Condition("sobrenome", "Silva")));
Person result = q.getSingle();

```

Listagem 25 – Exemplo de criação de query “idade = 14 ou (nome = João e sobrenome = Silva)” no mapeador objeto-relacional criado

Perceba como condições compostas que fazem uso dos conectores lógicos *and* e *or* podem ser criadas fazendo uso tanto dos métodos presentes na classe *Query* como por meio do próprio construtor da classe *Condition*, bastando para isso fornecer-lhe o tipo da condição desejada nos argumentos.

Os resultados da execução da consulta podem ser obtidos de duas formas diferentes. Caso seja uma query que, previamente, sabemos retornar um resultado único ou que, por particularidades da aplicação que a usa, tenha relevância apenas para o primeiro resultado, o método *getSingle()* pode ser utilizado, retornando apenas uma única instância da entidade buscada. No caso de todos os resultados serem necessários, deve ser feito uso do método *asList()*, que retorna uma lista das entidades.

Os tipos de comparação possíveis definidos por *ComparisonType* são todos os que utilizamos regularmente: *greater*, *greater or equals*, *lesser*, *lesser or equals*, *equals*, *not equals*, *contains*, *starts* e *ends* – sendo essas últimas para uso com expressões regulares. Dessa forma, a classe *Condition* é capaz de representar adequadamente quaisquer tipos de comparações que possam ser exigidas pelo kernel.

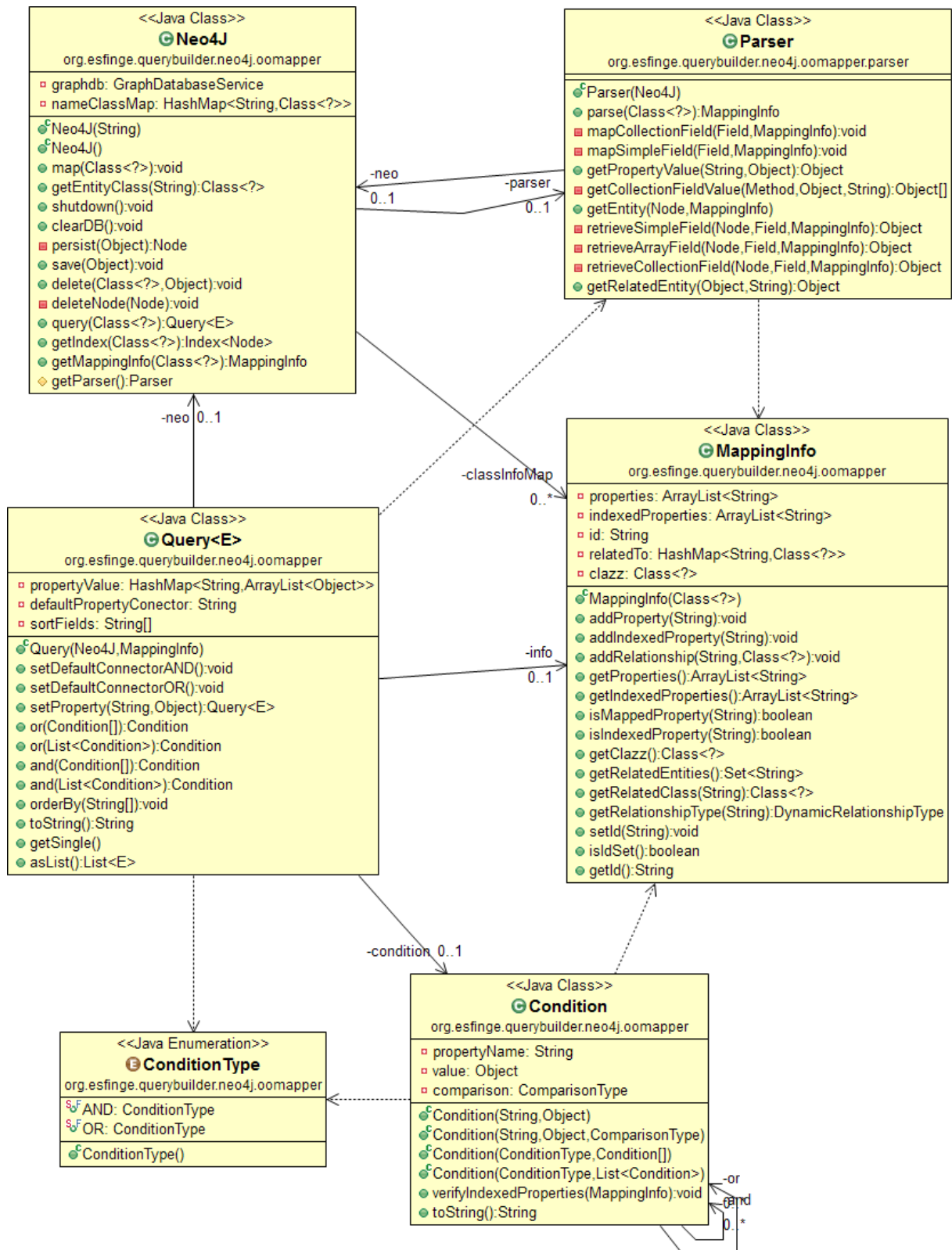


Figura 8 – Diagrama de classes do mapeador objeto-relacional desenvolvido para o banco Neo4J

A implementação do módulo do framework para o banco Neo4J não apresenta estrutura muito diferente em relação ao que foi mostrado para o banco MongoDB. Tudo o que foi dito em relação à utilização do padrão *visitor* a fim de desacoplar a lógica por trás da

montagem da query, comum a todos os bancos, das operações de criação da query em si, particular para cada base de dados, ainda se aplica. De fato, observe o diagrama de classes referente à implementação da interface *QueryExecutor*:

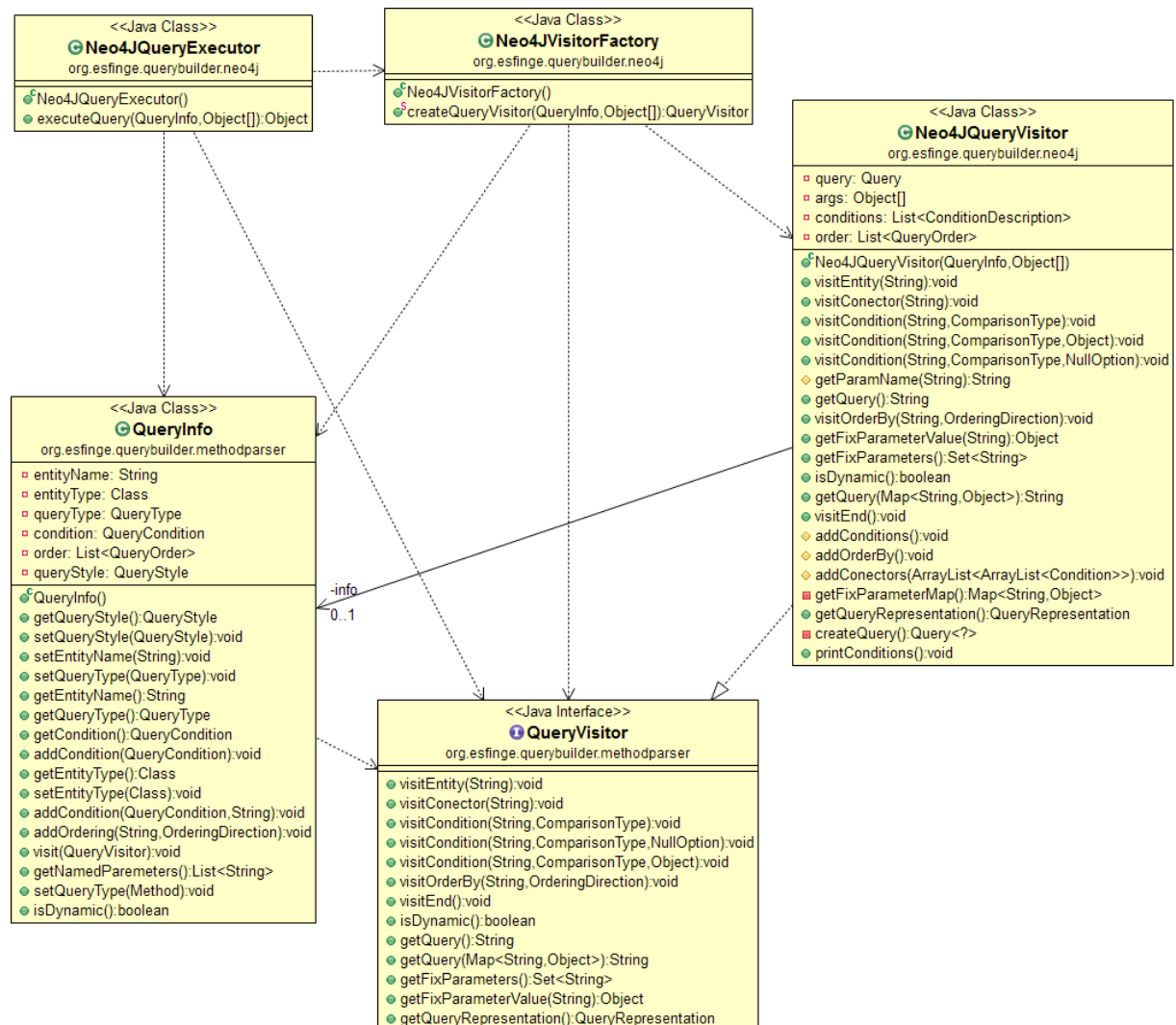


Figura 9 – Diagrama de classes referente à implementação da interface *QueryExecutor* para o banco Neo4J

Perceba que o diagrama exposto na Figura 9 é o mesmo presente na Figura 7. De fato, como já foi dito, as mudanças de um módulo para outro se concentram nas operações desempenhadas pelo *Visitor*.

Quanto às outras duas interfaces requeridas – *Repository* e *EntityClassProvider* – se assemelham bastante ao que foi desenvolvido para o banco MongoDB. Para a *EntityClassProvider*, o mapeador mantém, como o Morphia mantinha, um mapa de todas as classes registradas para uso, utilizando seus nomes como chave. A informação é obtida facilmente, então. Quanto à interface *Repository*, como o mapeador já provê funcionalidades CRUD básicas, sua implementação faz forte uso do mesmo, sem necessidade de grandes

alterações. O método *QueryByExample*, que vai além do simples CRUD, é feito de maneira análoga ao que foi feito para o banco MongoDB.

4.3.3 Limitações e Dificuldades

Quando os bancos NoSQL baseados em grafos foram apresentados, foi dito que um dos motivos que podem levar algum desenvolvedor a utilizá-los em detrimento dos já sedimentados bancos relacionais é a possibilidade de realizar operações de grafos – percursos, por exemplo – como forma de recuperar informações.

Todavia, a camada de abstração para o processo de persistência criada pelo framework QueryBuilder, pela sua própria intenção de uniformizar o acesso, acaba por suprimir as individualidades de cada base de dados, incluindo seus pontos positivos.

É fato que a implementação desenvolvida tenta fazer uso máximo das capacidades individuais, mas não é difícil compreender que as restrições impostas pelo desejo de uma maneira uniforme de tratar coisas diferentes se opõe a isso.

Para fazer uso máximo de suas capacidades, o desenvolvedor deverá usar a interface provida pelo banco, com funcionalidades de mais baixo nível.

Uma limitação conhecida do banco Neo4J, não do framework QueryBuilder e seu módulo, é a incapacidade de lidar com consultas baseadas em *regular expressions* iniciadas com a *wildcard* * quando o banco cresce. É recomendado cuidado ao criar queries assim – e mesmo evitá-las por completo.

O mapeamento objeto-relacional não contemplou, por falta de tempo hábil para tanto, relacionamentos com multiplicidade “muitos para muitos”.

5 AVALIAÇÃO

Nesse capítulo o trabalho desenvolvido será avaliado quanto à sua capacidade de cumprir o objetivo especificado para o mesmo na seção 1.2, ou seja, averiguaremos se o framework Esfinge QueryBuilder, munido de suas novas adições, é capaz de prover portabilidade fácil à aplicações que o utilizam.

Isso será feito de duas formas: primeiro, uma bateria de testes será realizada; em seguida, uma aplicação real será desenvolvida fazendo uso do framework, devendo ser capaz de funcionar com os paradigmas relacional e NoSQL sem a necessidade de grandes alterações de código.

5.1 Testes

O procedimento de testes será feito conforme aquilo que foi dito na seção 1.5, ou seja, os testes serão tais que todas as funcionalidades providas pelo kernel do framework serão avaliadas quanto ao seu correto funcionamento. Pode ser separado em dois grupos.

O primeiro é formado pelos testes de unidade realizados em cada módulo, a fim de garantir que suas operações fazem o esperado e geram resultados corretos – que as queries são montadas de acordo com o que o kernel passou no objeto *QueryInfo*, por exemplo. Nesse ponto, não são as funcionalidades do framework em si o objeto de atenção, mas a garantia de que as operações básicas ocorram. Por sua característica, são específicos para cada banco.

O segundo visa aferir se o conjunto como um todo funciona, ou seja, é constituído por testes de integração que fazem uso do framework da mesma maneira que uma aplicação faria. Os testes desse grupo não têm – e não precisam ter – qualquer conhecimento acerca de qual banco está sendo utilizado. Comportando-se como uma aplicação que faz uso da API exposta pelo framework, eles também são portáteis, de forma que se mantêm inalterados para todos os bancos testados. São eles os responsáveis por, de fato, verificar as funcionalidades providas.

Esse é um ponto importante a ser ressaltado: os módulos são submetidos a um mesmo conjunto de testes em sua validação, o que empresta um caráter de uniformidade à mesma. Como o objetivo principal desse trabalho é a portabilidade, é essencial não haver diferenças entre as maneiras como os diferentes bancos são utilizados.

A aplicação de testes é baseada na biblioteca Java JUnit, própria para esse fim. Os testes efetuados, assim como suas respectivas finalidades, são listados a seguir.

Tabela 4 – Testes realizados

Teste	Finalidade
domainQuery	Verificam se a funcionalidade <i>DomainTerm</i> apresenta o comportamento correto: checa a definição de um <i>DomainTerm</i> ; averigua se uma consulta que envolve um ou mais <i>DomainTerms</i> associados por meio de conectores lógicos retorna os resultados esperados, com presença ou ausência de condições que não sejam <i>DomainTerms</i> .
onDomainTermsQuery	
<u>twoDomainTerms</u>	
domainTermWithTwoWords	
domainTermWithParameter	
simpleCustomMethod	Verificam o funcionamento correto da configuração por meio dos arquivos da pasta META_INF/services, ou seja, garantem que as interfaces especificadas nos nomes dos arquivos são implementadas pelas classes cujos nomes estão presentes nos respectivos arquivos.
genericCustomInterface	
twoGenericCustomInterface	
compareToNullQuery	Queries que recebem parâmetros nulos podem ser definidas de maneira a buscar esse valor no banco ou simplesmente ignorar a consulta. Esses testes garantem que a definição do comportamento faz o desejado.
compareToNullQueryWithOtherParams	
ignoreWhenNull	
ignoreWhenNullWithTwoParams	
simpleQuery	Verificam se as funcionalidades básicas respondem adequadamente, como criação de uma query com único parâmetro, capacidade de retornar um único resultado ou lista de resultados.
simpleParameterQuery	
listParameterQuery	
queryWithTwoOrParameters	Testam queries envolvendo todos os operadores lógicos que são disponibilizados
queryWithGreaterThan	

queryWithLesserThan	para a montagem das consultas, como <i>and</i> , <i>or</i> , <i>not</i> , etc. Também verifica o funcionamento de queries baseadas em <i>regular expressions</i> .
queryWithNotEquals	
queryWithStringStarted	
queryWithTwoParametersWithComparisonTypes	
orderByQuery	Verificam o funcionamento da ordenação de resultados.
orderByWithParameter	
queryWithOtherTable	Verificam que a sintaxe “AddressState” faz uma busca no campo “State” da propriedade “Address”, que também é uma entidade de persistência.
compositeQueryWithOtherTable	
list	Verificam se os métodos CRUD – e o <code>queryByExample</code> – da interface <code>Repository</code> funcionam.
queryByExample	
getId	
delete	
save	
update	
simpleQueryObject	Verificam o funcionamento de queries que fazem uso de <i>QueryObjects</i> .
queryObjectWithComparisonType	
queryObjectWithDomainTerm	
queryObjectWithContainsDomainTerm	
queryObjectWithNullComparison	
queryObjectIgnoreNull	
queryObjectWithOrderBy	

5.2 Estudo de Caso

Nessa seção será desenvolvida uma aplicação pequena que fará uso do framework Esfinge QueryBuilder para sua comunicação com o banco de dados. A fim de que o objetivo de portabilidade seja aferido, essa aplicação será executada tanto em um banco relacional como no MongoDB e no Neo4J. As alterações de código necessárias para isso serão postas em evidência.

O conceito da aplicação é muito simples: um sistema de gerenciamento de um *pet shop* que conta apenas com um registro de clientes, seus respectivos cachorros e, como não pode deixar de ser em uma empresa, os pagamentos por eles feitos. Apresenta uma tela única, como pode ser visto abaixo:

Clientes:

Matrícula	Nome
0	Maria
1	Júlia
2	Pedro

Pedro

Pagamentos:

ID	Data	Valor
5	07/12/2012	50.0
4	04/11/2012	50.0
3	05/10/2012	50.0

07/12/2012
50.00

Cachorros:

Nome	Raça
Lalinha	Poodle

Lalinha
Poodle

Figura 10 – Tela da aplicação *Pet Shop Systems*

Ao clicar nos nomes de cada cliente, as tabelas de cachorros e pagamentos são preenchidas com os respectivos dados do cliente selecionado. Funcionalidades básicas de inserção e remoção são providas.

O grande ponto a ser notar aqui é que a aplicação não necessita de mudanças significativas para rodar em qualquer dos bancos, tanto relacional como os NoSQL abordados. De fato, apenas alguns passos de configuração e as anotações utilizadas no mapeamento precisam ser refeitos, conforme será apresentado nas seções seguintes. Todo o resto do código não precisa de qualquer tipo de mudança. A aplicação tem portabilidade fácil, como era o objetivo desse trabalho.

5.2.1 Configuração

Existem passos que envolvem aspectos fundamentais de configuração, como a localização na estrutura de arquivos dos dados armazenados pelo banco, as classes Java que deverão ser utilizadas como entidades de persistência, qual classe implementa a interface

DatastoreProvider – para os bancos NoSQL desenvolvidos – ou *EntityManagerProvider* – no uso do módulo JPA –, entre outras coisas. Tudo aquilo que deve ser feito para que a aplicação rode com qualquer dos bancos será apresentado a seguir.

Para bancos relacionais as configurações são um pouco diferentes daquilo exposto anteriormente para os módulos desenvolvidos nesse Trabalho de Graduação. De fato, como faz uso de JPA para comunicar-se com o banco, as configurações referentes ao funcionamento da API são necessárias – o arquivo *persistence.xml*, com o qual já estamos tão acostumados.

Além disso, de maneira análoga ao que foi feito para especificar a implementação da interface *DatastoreProvider*, é necessário definir por meio de um arquivo qual classe implementa *EntityManagerProvider*. Deve ser criado o arquivo “META-INF/services/org.esfinge.querybuilder.jpa1.EntityManagerProvider”, contendo apenas o nome de tal classe.

Para o MongoDB, conforme foi dito em 4.2.1, deve ser configurada e especificada a classe que implementa a interface *DatastoreProvider*.

```
@ServicePriority(1)
public class MongoDBDatastoreProvider extends DatastoreProvider{

    public MongoDBDatastoreProvider(){

        try {
            mongo = new Mongo("localhost", 27017);
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (MongoException e) {
            e.printStackTrace();
        }

        getMorphia().map(Cachorro.class);
        getMorphia().map(Cliente.class);
        getMorphia().map(Pagamento.class);
    }

    @Override
    public Datastore getDatastore() {
        return getMorphia().createDatastore(mongo, "testdb");
    }

}
```

Listagem 26 – Definição da interface *DatastoreProvider* para o banco MongoDB

```
Application.ip.MongoDBDatastoreProvider
```

Listagem 27 – Conteúdo do arquivo de configuração “META-INF/services/org.esfinge.querybuilder.mongodb.DatastoreProvider”

A configuração referente ao Neo4J é análoga. Detalhes podem ser obtidos em 4.3.1.

```
@ServicePriority(1)
public class Neo4JDatastoreProvider implements DatastoreProvider{

    private static final Neo4J neo4j = new Neo4J("/database");
    public Neo4JDatastoreProvider(){
        neo4j.map(Cliente.class);
        neo4j.map(Pagamento.class);
        neo4j.map(Cachorro.class);
    }
    @Override
    public Neo4J getDatastore() {
        return neo4j;
    }
}
```

Listagem 28 – Definição da interface *DatastoreProvider* para o banco Neo4J

```
Application.ip.Neo4JDatastoreProvider
```

Listagem 29 – Conteúdo do arquivo de configuração “META-INF/services/org.esfinge.querybuilder.neo4j.DatastoreProvider”

5.2.2 Anotações

Devido ao fato de o mapeamento de objetos Java para entidades possíveis de serem armazenadas nos bancos não ser uniforme, as anotações utilizadas pelos diferentes módulos do framework para realizar tal operação também apresentam suas diferenças. A seguir, serão apresentadas as anotações utilizadas para os diferentes bancos. Por não representarem algo de interesse aqui, os métodos das regras de negócio de cada classe serão omitidos.

```
@Entity
public class Cliente {

    @Id
    private int matricula;
    private String nome;

    @OneToMany
    private Set<Cachorro> cachorros = new HashSet<Cachorro>();
    @OneToMany
    private Set<Pagamento> pagamentos = new HashSet<Pagamento>();

}
```

Listagem 30 – Anotações na classe Cliente para JPA

```
@Entity
public class Pagamento {

    @Id
    private int id;
    private float valor;
    private String data;

}
```

Listagem 31 – Anotações na classe Pagamento para JPA

```
public class Cachorro {

    @Id
    private String nome;
    private String raca;

}
```

Listagem 32 – Anotações na classe Cachorro para JPA


```
public class Cliente {

    @Id
    private int matricula;
    private String nome;

    @Reference
    private Set<Cachorro> cachorros = new HashSet<Cachorro>();
    @Reference
    private Set<Pagamento> pagamentos = new HashSet<Pagamento>();

}
```

Listagem 33 – Anotações na classe Cliente para o banco MongoDB

```
public class Pagamento {

    @Id
    private int id;
    private float valor;
    private String data;

}
```

Listagem 34 – Anotações na classe Pagamento para o banco MongoDB

```
public class Cachorro {

    @Id
    private String nome;
    private String raça;

}
```

Listagem 35 – Anotações na classe Cachorro para o banco MongoDB

```

@Entity
public class Cliente {

    @Id
    private int matricula;
    private String nome;

    @RelatedTo(targetClass = Cachorro.class)
    private Set<Cachorro> cachorros = new HashSet<Cachorro>();
    @RelatedTo(targetClass = Pagamento.class)
    private Set<Pagamento> pagamentos = new HashSet<Pagamento>();

}

```

Listagem 36 – Anotações na classe Cliente para o banco Neo4J

```

@Entity
public class Pagamento {

    @Id
    private int id;
    private float valor;
    private String data;

}

```

Listagem 37 – Anotações na classe Pagamento para o banco Neo4J

```

@Entity
public class Cachorro {

    @Id
    private String nome;
    private String raça;

}

```

Listagem 38 – Anotações na classe Cachorro para o banco Neo4J

5.3 Análise do Estudo de Caso

É importante frisar aqui que, entre todas as classes que compõem a aplicação desenvolvida para esse estudo de caso, apenas aquelas que representam as entidades de persistência precisaram ser alteradas para tornar possível o funcionamento em outro banco – e a classe de configuração que implementa *DatastoreProvider*, é claro.

Para uma aplicação simples como a apresentada, isso já representa um ganho considerável – é preciso alterar três classes de entidade e uma de configuração, quatro classes, de um total de dez. Todavia, o ganho é muito subestimado nesse exemplo. De fato, no mundo real as classes que representam entidades serão um número muito pequeno diante do total presente na aplicação – até mesmo no nosso exemplo, uma aplicação que existe apenas para acessar o banco, seu número já é menor do que a metade do total.

Some-se a isso o fato de que mesmo as classes que devem ser alteradas não precisarem de mudanças significativas – basta uma pequena readequação de anotações, nada trabalhoso – e podemos concluir aqui que o objetivo de portabilidade foi alcançado com sucesso.

6 CONCLUSÃO

O problema inicial – disponibilizar uma API de acesso uniforme para bancos de dados de diferentes paradigmas – está, agora, mais próximo de uma solução. Com a contribuição dada por esse Trabalho de Graduação, é possível que aplicações baseadas no framework Esfinge QueryBuilder sejam executadas com base nos bancos NoSQL MongoDB e Neo4J, estendendo suas capacidades originais, que antes limitavam-se ao escopo relacional.

Esse alargamento do escopo do framework foi atingido através da implementação de dois módulos novos, referentes aos dois bancos citados. Tal extensão só foi possível devido ao design modular do Esfinge QueryBuilder, pensado desde o início para ser facilmente complementado.

A validade da solução apresentada foi posta em xeque tanto por uma extensa bateria de testes – que avaliam se cada funcionalidade do framework responde corretamente – como por um estudo de caso. O mesmo consistiu em uma aplicação real que foi desenvolvida com base na API do QueryBuilder e, em seguida, executada em três bancos de dados diferentes, com alterações mínimas de código.

Nos dias de hoje, em que cada vez mais o uso de bancos de dados NoSQL se difunde, qualquer tentativa no sentido de unificar o acesso aos mesmos sob o padrão de uma API deve ser vista com importância. De fato, uma interface uniforme não apenas promove um ambiente de portabilidade tranquila, como também reduz os esforços necessários para o desenvolvimento de aplicações que façam uso de tais bancos – não são necessários conhecimentos específicos – e a energia dispendida em sua manutenção.

Apesar da grande vantagem de obter a uniformização da interface de persistência buscada por esse Trabalho de Graduação, com suas implicações positivas em desenvolvimento, portabilidade e manutenção, esse paradigma trás, também, consequências negativas. Afinal, o surgimento de bancos diferentes, com estruturas diferentes, não é apenas amor pelo novo: cada estrutura é pensada de maneira otimizada para algum tipo de aplicação. Tratá-las da mesma forma, desprezando suas idiossincrasias que as tornam ótimas para algum cenário de uso, acaba suprimindo essas características.

É claro que cada módulo é feito com foco no banco em questão, buscando maximizar o uso de seu potencial. Mas uma interface padrão não leva em conta as capacidades únicas de cada banco, de maneira que o resultado final acaba subutilizando-os.

6.1 Contribuições

Foram desenvolvidos dois novos módulos para o framework Esfinge QueryBuilder: um para o banco MongoDB e outro para o Neo4J, ambos pertencentes ao paradigma NoSQL. Dessa forma, foi demonstrada a possibilidade de prover uma API uniforme de acesso que contempla tanto bancos do paradigma relacional como os mais diversos bancos NoSQL.

Ainda, foi criado um mapeador de objetos Java para entidades de persistência do banco baseado em grafos Neo4J. Esse mapeador faz o uso de metadados – anotações – de forma a efetuar a tradução de forma automática.

6.2 Trabalhos Futuros

O trabalho conseguiu atingir o objetivo almejado, mas ainda existem melhorias a serem implementadas. Um possível trabalho futuro, de maneira a tornar ainda mais fácil a portabilidade, seria uniformizar a maneira de anotar os objetos – o que implicaria em abrir mão do uso do framework Morphia e desenvolver, assim como feito para o Neo4J, um mapeador próprio que siga as especificações JPA.

A extensão do framework a outros bancos, além dos dois contemplados por esse trabalho, também deve ser considerada.

REFERÊNCIAS BIBLIOGRÁFICAS

A type-safe Java library for MongoDB. **Morphia**. Disponível em:

<<http://code.google.com/p/morphia/>>. Acesso em: 06 nov. 2012.

ADELSON-VELSKII, G. M.; LANDIS, E. M. An algorithm for the organization of information. **Proceedings of the USSR Academy of Sciences** **146**, 1962. 263-266.

CODD, E. F. A relational model of data for large shared data banks. **Communications of the ACM**, v. 13, n. 6, p. 377-387, 1970.

EINI, O. RavenDB Mythology Documentation. **RavenDB**. Disponível em:

<<https://s3.amazonaws.com/daily-builds/RavenDBMythology-11.pdf>>. Acesso em: 10 nov. 2012.

FOWLER, M. et al. **Patterns of Enterprise Application Architecture**. 2002.

GAMMA, E. et al. **Design Patterns**. 1998.

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **ACM SIGACT News**, v. 33, n. 2, p. 51-59, 2002.

HAERDER, T.; REUTER, A. Principles of transaction-oriented database recovery. **ACM Computing Surveys**, v. 15, n. 4, p. 287-317, 1983.

HARDY, G. H.; WRIGHT, E. M. Some Notations. In: HARDY, G. H.; WRIGHT, E. M. **An Introduction to the Theory of Numbers**. Oxford: Clarendon Press, 1979. p. 7-8.

KUNG, H. T. On Optimistic Methods for Concurrency Control. **ACM Transactions on Database Systems**, New York, v. 6, n. 2, p. 213 - 226, 1981.

OBJECT MANAGEMENT GROUP. Documents Associated With Unified Modeling Language (UML), V2.4.1. **Object Management Group**. Disponível em:

<<http://www.omg.org/spec/UML/2.4.1/>>. Acesso em: 14 out. 2012.

SERRA, J. P. Manual de Teoria da Comunicação. In: SERRA, J. P. **Manual de Teoria da Comunicação**. Covilhã: Livros Labcom, 2007. p. 93-101.

SPRING. Spring. **Spring Data**. Disponível em: <<http://www.springsource.org/spring-data>>. Acesso em: 22 nov. 2012.

STROZZI, C. NoSQL. **NoSQL: a non-SQL RDBMS**. Disponível em:

<http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home%20Page>. Acesso em: 18 out. 2012.

THE APACHE SOFTWARE FOUNDATION. Cassandra. **The Apache Cassandra Project**. Disponível em: <<http://cassandra.apache.org/>>. Acesso em: 24 out. 2012.

VOGELS, W. Eventually Consistent. **Communications of the ACM**, New York, v. 52, n. 1, 2009.

FOLHA DE REGISTRO DO DOCUMENTO			
1. CLASSIFICAÇÃO/TIPO TC	2. DATA 23 de novembro de 2012	3. REGISTRO N° DCTA/ITA/TC-128/2012	4. N° DE PÁGINAS 77
5. TÍTULO E SUBTÍTULO: Extensão do framework esfinge QueryBuilder para bancos de dados NoSQL.			
6. AUTOR(ES): Allan Machado da Silva			
7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES): Instituto Tecnológico de Aeronáutica – ITA			
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: Bancos de Dados, NoSQL, API uniforme			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Banco de dados; Estruturas (processamento de dados); Programas de aplicação (computadores); Engenharia de sistemas; Computação.			
10. APRESENTAÇÃO:		X Nacional	Internacional
ITA, São José dos Campos. Curso de Graduação em Engenharia de Computação. Orientador: Eduardo Martins Guerra. Publicado em 2012.			
11. RESUMO: Com o recente crescimento do uso de alternativas ao tradicional modelo relacional de bancos de dados, movimento conhecido como NoSQL, surge a necessidade de uma API que uniformize o acesso aos bancos sob um mesmo padrão – tanto os pertencentes ao novo paradigma quanto os já consagrados relacionais. Isso tem importância não só pela portabilidade que um tratamento uniforme proporciona, mas também pela simplificação do trabalho de quem desenvolve aplicações que fazem uso de bases de dados – o acesso heterogêneo exige conhecimentos específicos do desenvolvedor. Esse trabalho é uma tentativa nesse sentido. Com as extensões desenvolvidas para o framework Esfinge QueryBuilder, seu escopo que antes era limitado a bancos relacionais passa, também, a contemplar os bancos MongoDB e Neo4J, provendo uma mesma interface de acesso para bancos pertencentes a paradigmas diferentes.			
12. GRAU DE SIGILO: (X) OSTENSIVO () RESERVADO () CONFIDENCIAL () SECRETO			