Neo4j na prática

Como entrar no mundo dos bancos de dados de grafo

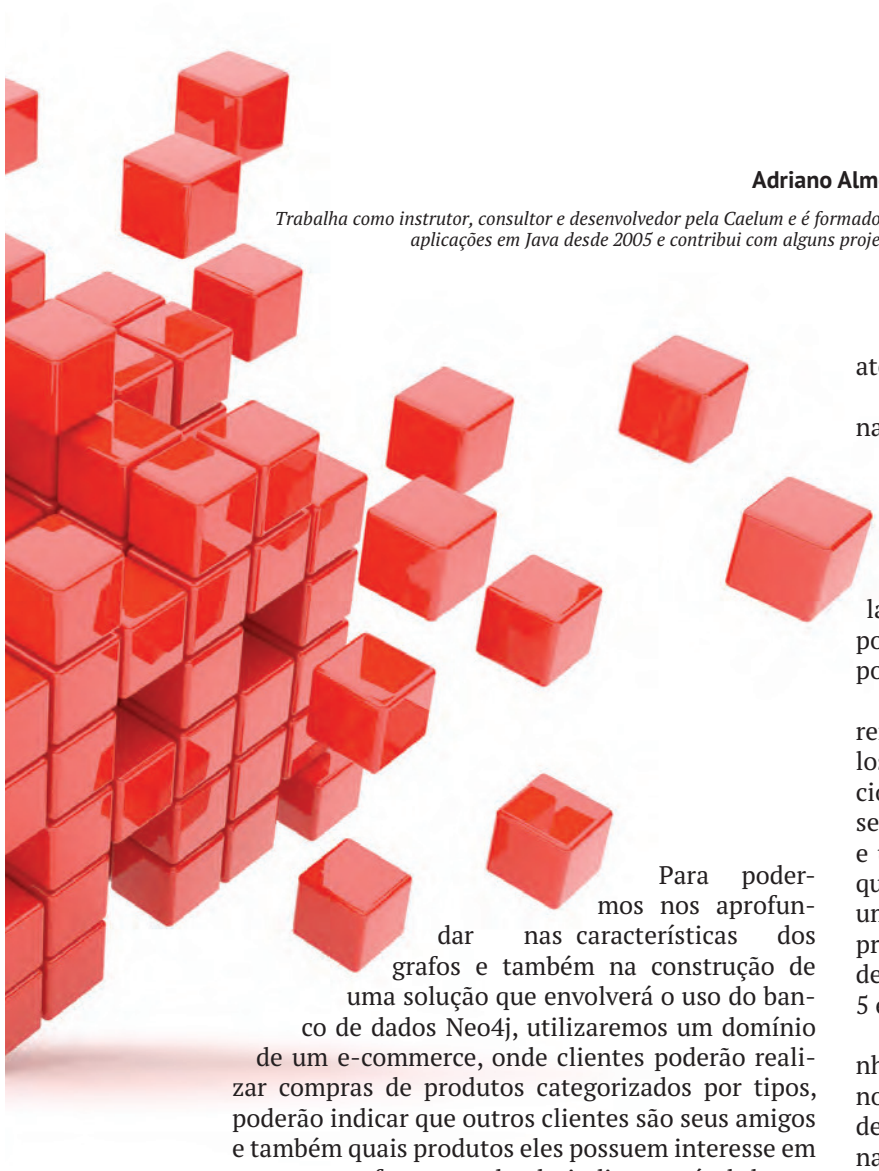
O mundo dos bancos de dados não-relacionais (NoSQL) cada vez mais tem ganhado reconhecimento do mercado, inclusive com grandes empresas adotando alguma das soluções disponíveis ou patrocinando o desenvolvimento das já existentes. Nesse cenário, ganhando grande atenção no mercado internacional e baseado em uma das mais antigas teorias matemáticas, o banco de dados Neo4j, implementado em Java, traz os grafos ao cenário da persistência de dados, de uma maneira que pode simplificar a solução de problemas não-triviais em bancos de dados relacionais ou até mesmo como uma boa alternativa à tradicional modelagem relacional.

A adoção de um banco de dados não-relacional geralmente passa por alguns fatores bastante comuns, como necessidade de alta escalabilidade e disponibilidade, flexibilidade de schema e ganhos de performance. O banco de dados Neo4j pode prover todos esses recursos às aplicações, aliando a elas um modelo rico de dados, baseado em grafos, permitindo a criação de modelos de dados extremamente complexos e, ainda assim, mantendo uma considerável facilidade em pesquisar dados e adicionando a tudo isso características que tornaram os bancos de dados relacionais ferramentas tão reconhecidas no mercado, como transações ACID e uma linguagem simples para realizar pesquisa de dados.

Neste artigo o leitor aprenderá como modelar suas aplicações dentro desse novo paradigma que é a modelagem via grafos, e será introduzido aos conceitos desse novo paradigma através de uma adaptação do modelo de dados de um e-commerce. Também será mostrado como utilizar a API do Neo4j para persistir e pesquisar as informações no banco de dados, utilizando desde a API Java até a poderosa linguagem de pesquisas Cypher.

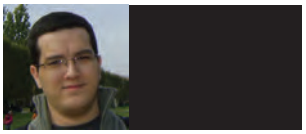
A modelagem através de grafos

Muitos desenvolvedores estão acostumados com o processo de modelagem relacional, onde definimos quais tabelas e colunas farão parte do nosso modelo de dados. Definimos também as tabelas envolvidas e seus respectivos relacionamentos pensando em suas cardinalidades, por exemplo. A modelagem nos consegue mostrar como as informações são interligadas em nosso modelo de negócio e como os dados estão relacionados entre si.



Adriano Almeida | adriano.almeida@caelum.com.br

Trabalha como instrutor, consultor e desenvolvedor pela Caelum e é formado em Sistemas de Informação pela FIAP. Desenvolve aplicações em Java desde 2005 e contribui com alguns projetos open sources, inclusive o banco de dados Neo4j



Para podermos nos aprofundar nas características dos grafos e também na construção de uma solução que envolverá o uso do banco de dados Neo4j, utilizaremos um domínio de um e-commerce, onde clientes poderão realizar compras de produtos categorizados por tipos, poderão indicar que outros clientes são seus amigos e também quais produtos eles possuem interesse em comprar no futuro, podendo indicar o nível desses interesses em uma escala de 1 a 5 (pouco interesse até muito interesse).

Um exemplo de um grafo possível pode ser visto na imagem 1.

Em nosso caso, indicamos que uma pessoa possui interesse em alguns produtos. Os produtos, por sua vez, possuem uma relação com sua categoria. A partir do momento em que o cliente realiza uma compra, também passa a existir um relacionamento entre o cliente e a compra, que pode possuir vários produtos associados a ela. Os clientes podem possuir amizade com outros clientes.

No grafo, alguns pontos são importantes de serem notados. Os nós são representados pelos círculos e podem possuir propriedades. Enquanto os relacionamentos são representados pelas setas, que são sempre direcionadas, ou seja, possui um nó de saída e um nó de chegada, como, por exemplo, o cliente que possui um relacionamento "INTERESSE" com um produto. Os relacionamentos podem possuir propriedades também, como, por exemplo, o nível de interesse, que mediremos em um ranking de 1 a 5 estrelas.

É importantíssimo verificar também que em nenhum momento definimos chaves estrangeiras em nosso modelo. Essa é uma característica vital na modelagem através de grafos, onde os dados se relacionam de forma natural, ou seja, através de seu papel no domínio, sem a necessidade da criação de uma estrutura específica para tal.

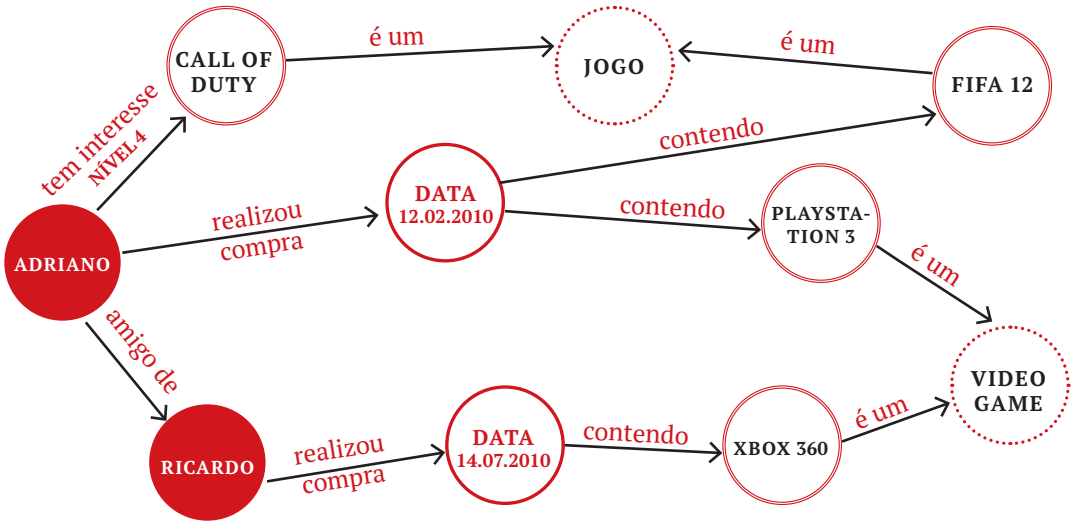


Imagem 1. *Grafo demonstrando diversos relacionamentos entre diferentes tipos de informações.*

Com isso, temos o nosso grafo modelado e já conhecemos suas estruturas. No entanto, ainda precisamos dar vida à nossa modelagem no banco de dados Neo4j.

A instalação do Neo4j e a persistência dos primeiros dados

O primeiro passo para conseguirmos criar uma aplicação que utilize o banco de dados Neo4j, é realizarmos o seu download e instalação. O arquivo .zip contendo todos os componentes necessários para utilizar o banco de dados pode ser encontrado em *http://neo4j.org/download/* onde é possível escolher o arquivo mais adequado para cada tipo de ambiente. Uma vez escolhido e baixado o pacote, basta descompactar o arquivo no seu sistema. Dentro da estrutura de diretórios do pacote, todos os jars necessários para que o banco de dados funcione podem ser encontrados dentro da pasta "lib". Pronto, com isso, podemos criar um projeto em nossa IDE preferida, adicionando os jars ao classpath de nossa aplicação. Note que o processo é extremamente simples e se o leitor quiser, também é possível adicionar Neo4j em seu projeto maven adicionando o código da Listagem 1 ao seu arquivo pom.xml.

Listagem 1. Código de importação do Neo4j via maven.

```
<dependency>
   <groupId>org.neo4j</groupId>
   <artifactId>neo4j</artifactId>
   <version>1.5</version>t
</dependency
```

Com um projeto criado, podemos persistir nossa primeira informação. No caso, será um produto.

A criação do banco de dados

A partir do momento em que temos as bibliotecas do Neo4j, é possível criar o primeiro banco de dados, para que seja possível populá-lo.

Para isso, precisamos utilizar a API do próprio Neo4j, simplesmente instanciando um objeto do tipo EmbeddedGraphDatabase, que implementa a interface GraphDatabaseService, passando como parâmetro para seu construtor, uma String contendo o caminho no disco para onde o banco de dados estará armazenado. Um exemplo dessa instrução é mostrado no código da Listagem 2, onde existe um método main que cria o banco de dados.

Listagem 2. Classe que cria o banco de dados em um diretório.

```
public class CriaBanco {
   public static void main(String[] args) {
      GraphDatabaseService db = new
EmbeddedGraphDatabase("/tmp/db");

      db.shutdown();
   }
}
```

É imprescindível que sempre que a criação de um objeto do tipo EmbeddedGraphDatabase, que representa um banco de dados do Neo4j, é realizada, antes de a aplicação terminar, o mesmo objeto seja corretamente fechado através da invocação do método shutdown(). Agora que já sabemos como um banco de dados pode ser criado, podemos partir para a persistência das informações.

ACID e controle de transações

Uma das principais características do Neo4j é manter um sistema transacional que respeita as tão conhecidas propriedades ACID, muito comuns em bancos de dados relacionais, na qual é dito que as transações devem ser atômicas, levar o banco de dados de um estado consistente a outro, manter o isolamento das transações e cuidar da durabilidade dos dados. Juntas, essas características formam um dos principais pilares que sustentam os sistemas de bancos de dados relacionais.

Com o Neo4j, o ponto de entrada para trabalhar com transações é a classe EmbeddedGraphDatabase, através do seu método beginTx(), definido na interface GraphDatabaseService, que retorna um objeto do tipo Transaction. A partir do momento em que o método beginTx() é invocado, todas as operações envolvendo o banco de dados, naquela Thread, é considerado parte da transação, até o momento em que o método success() ou failure() sejam invocados na mesma. Por fim, a transação deve ser finalizada através do método finish(). Um exemplo de um bloco que cria e finaliza as transações do Neo4j é mostrado na Listagem 3.

Listagem 3. Abertura e fechamento de transações com o Neo4j.

```
public class PersisteProduto {
   public static void main(String[] args) {
      GraphDatabaseService db =
         new EmbeddedGraphDatabase("/tmp/db");
      Transaction tx = db.beginTx();
```

```
        try {
            // operações dentro da transação vão aqui

            tx.success();
        } finally {
            tx.finish();
        }
        db.shutdown();
    }
}
```

Criação do primeiro produto e categoria

A partir do momento em que já é possível trabalhar com transações com o Neo4j, está tudo pronto para realizar a gravação de um nó para o produto no banco de dados. Para isso, basta invocarmos o método createNode() da classe EmbeddedGraphDatabase, que devolve um objeto do tipo Node, no qual é possível definir suas propriedades. No caso de um novo nó para o produto, o código da transação ficará similar ao da Listagem 4.

Listagem 4. Cria um novo nó e define as propriedades.

```
Transaction tx = db.beginTx();
try {
    Node noProduto = db.createNode();
    noProduto.setProperty("nome","Playstation 3");
    noProduto.setProperty("preco", 850.99);

    System.out.println("Novo produto gravado com id:" +
        noProduto.getId());

    tx.success();
} finally {
    tx.finish();
}
```

O mesmo processo pode ser aplicado para a criação de uma categoria, cujo exemplo de código é mostrado na Listagem 5 e com isso é possível realizar a persistência dos primeiros nós no banco de dados.

Listagem 5. Cria um novo nó para categoria e define as propriedades.

```
Transaction tx = db.beginTx();
try {
    Node noCategoria = db.createNode();
    noCategoria.setProperty("nome","Video Game");

    System.out.println("Nova categoria gravada com id:" +
        noCategoria.getId());

    tx.success();
} finally {
    tx.finish();
}
```

Gravação de relacionamentos

Seguindo a definição do nosso domínio e a modelagem realizada no começo do artigo, é necessário relacionarmos os produtos com suas categorias. No nosso caso, podemos dizer que o Playstation 3 é um videogame, logo, precisamos representar o relacionamento "é um" em nossa aplicação. Uma das maneiras de definirmos o relacionamento de uma forma que não se baseie em Strings e seja type safe, é criarmos uma Enum que implemente a interface RelationshipType que é uma interface de marcação do Neo4j, ou seja, não possui métodos para ser implementados. Dessa maneira, podemos ter uma Enum chamada Relacionamentos, contendo o elemento E_UM, como na Listagem 6.

Listagem 6. Enum que representa o relacionamento.

```
public enum Relacionamentos implements
RelationshipType {
    E_UM;
}
```

O próximo passo é realizar o relacionamento das duas informações, ou seja, relacionarmos os dois nós que criamos previamente. Para tanto, precisaremos recuperar os dois nós do banco de dados. No momento, podemos realizar essa tarefa, através de uma simples busca pelo id no banco de dados, que é possível através do método getNodeById() da classe EmbeddedGraphDatabase. Uma vez com os dois objetos Node, é possível criarmos um relacionamento de um nó para o outro, através do métdo createRelationshipTo() da classe Node, que precisa de dois parâmetros, o nó de destino do relacionamento e o tipo de relacionamento, respectivamente. O nó de origem é definido através do nó no qual o método createRelationshipTo foi invocado. Considerando que o id do produto seja 1 e o id da categoria seja 2, o código para criar o relacionamento será equivalente ao da Listagem 7.

Neo4j e os ids

Devido ao fato de o Neo4j reaproveitar os ids, ou seja, quando um nó é excluído, o id passa a ser passível de reaproveitamento para futuros nós, o uso do método getNodeById é extremamente desencorajado. Em seu lugar, deve ser utilizado o mecanismo de indexação, que também é apresentado neste artigo.

Listagem 7. Relacionamento entre dois nós.

```
Transaction tx = db.beginTx();
try {
   Node noProduto = db.getNodeById(1);
   Node noCategoria = db.getNodeById(2);

   Relationship r = noProduto.createRelationshipTo(
      noCategoria, Relacionamentos.E_UM);
   // É possível chamar o método setProperty e definir
   // uma propriedade no relacionamento

   tx.success();
} finally {
   tx.finish();
}
```

Por fim, após a execução das classes, um grafo similar ao da imagem 2 existirá no banco de dados.

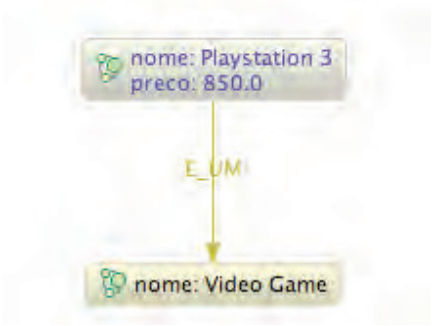


Imagem 2. *Relacionamento entre um produto e categoria.*

A partir desse instante, o mesmo processo pode ser realizado para cadastrar todas as outras informações da aplicação, inclusive, representando o modelo inicial da imagem 1 no banco de dados.

Indexação de informações e pesquisas simples

Conforme a aplicação vai ganhando mais e mais dados e os nós vão sendo persistidos, pode surgir a necessidade de realizar pesquisas, como, por exemplo, procurar o produto chamado "Geladeira", ou então, pesquisarmos a categoria "DVD".

O Neo4j provê esse tipo de pesquisa através da integração com a biblioteca Lucene pelas classes contidas no arquivo neo4j-lucene-index.jar, que se encarregam de realizar o trabalho de indexação dos dados. O próprio Lucene já vem distribuído com o conjunto de jars do Neo4j, bastando utilizá-la.

Indexação de informações

Para que seja possível realizar as pesquisas, primeiramente, é necessário indexar as informações. Na API do Neo4j, a classe responsável por ser o ponto de entrada para o processo de indexação é a IndexManager, da qual conseguimos uma instância através do método index() em EmbeddedGraphDatabase. Com o IndexManager, podemos invocar o método forNodes(), indicamos que queremos indexar nós do nosso banco de dados, e recebe como parâmetro o nome do índice, por fim, ele retorna um objeto do tipo Index<Node>, que nos permite manipular o índice, como mostrada na Listagem 8, que cria um índice para os produtos.

Listagem 8. Criação do índice para os nós referentes a produtos.

```
Transaction tx = db.beginTx();
try {
   IndexManager index = db.index();
   Index<Node> produtos = index.forNodes("produtos");

   tx.success();
} finally {
   tx.finish();
}
```

Com o objeto do tipo Index<Node> em mãos, é possível adicionarmos informações nele, para que o Lucene realize a indexação. Para isso, basta invocarmos o método add(), passando como parâmetro o nó que será indexado e um conjunto de chave e valor para a indexação. Através da chave, podemos identificar a informação que será indexada e o valor indica o conteúdo que será efetivamente indexado. Dessa maneira, é possível indexar o primeiro produto já adicionado, primeiramente buscando-o no banco de dados (vamos realizar a busca através do id) e em seguida indexando-o, como na Listagem 9.

Listagem 9. Indexação de um produto.

```
Transaction tx = db.beginTx();
try {
   IndexManager index = db.index();
   Index<Node> produtos = index.forNodes("produtos");

   Node noProduto = db.getNodeById(1);
   produtos.add(noProduto, "nome",
      noProduto.getProperty("nome"));
   tx.success();
} finally {
   tx.finish();
}
```

Realização de pesquisas através do Lucene

A partir do momento em que os nós estão indexados, é possível pesquisá-los através da API do Neo4j. Para tanto, é possível utilizar o método get() da classe Index que devolve um objeto do tipo IndexHits, que é um objeto Iterable, contendo os nós devolvidos pela pesquisa, além de possuir alguns métodos utilitários, como o getSingle(), que devolve apenas um elemento,

caso a pesquisa devolva somente um resultado. A iteração sobre o IndexHits para mostrar o nome do produto no banco de dados é mostrada na Listagem 10.

Listagem 10. Pesquisa de produto através do índice.

```
Transaction tx = db.beginTx();
try {
    IndexManager index = db.index();
    Index<Node> produtos = index.forNodes("produtos");

    IndexHits<Node> indexHits = produtos.get("nome",
       "Playstation 3");
    for (Node node : indexHits) {
       System.out.println(node.getProperty("nome"));
    }

    tx.success();
} finally {
    tx.finish();
}
```

Outra possibilidade é o uso do método query() ao invés do método get(), que permite a realização de pesquisas similares ao like do SQL. Nesse caso, para pesquisar os produtos cuja primeira letra é P, pode-se fazer como na Listagem 11.

Listagem 11. Pesquisa de produto com wildcard.

```
Transaction tx = db.beginTx();
try {
    IndexManager index = db.index();
    Index<Node> produtos = index.forNodes("produtos");

    IndexHits<Node> indexHits = produtos.query("nome",
       "P*");
    for (Node node : indexHits) {
       System.out.println(node.getProperty("nome"));
    }

    tx.success();
} finally {
    tx.finish();
}
```

Além da possibilidade de realizar pesquisas exatas e também similares ao like, como no caso das Listagens 10 e 11, o Neo4j também permite, através do uso do Lucene, a realização de pesquisas full-text.

Pesquisas através de relacionamentos

Apesar de a pesquisa através do índice permitir retornar as informações do banco de dados, muitas das vezes é necessário conhecer as adjacências de um nó, como, por exemplo, para saber as categorias de um determinado produto. Nesse caso, é possível simplesmente invocar o método getRelationships(), que possui várias sobrecargas, inclusive uma que recebe qual o relacionamento que se deseja buscar e a direção do relacionamento com relação ao nó. No caso do exemplo, no produto, desejamos todos os relacionamentos E_UM que sejam de saída do produto, ou seja, OUTGOING. A invocação ao método getRelationships devolve um Iterable<Relationship>, com o qual podemos percorrer todos os relacionamentos. Os objetos do tipo Relationship, possuem métodos chamados getStartNode() e getEndNode(), que permitem descobrir os nós envolvidos no relacionamento. No caso, podemos utilizar essas classes em conjunto para buscar as categorias do produto Playstation 3, como exemplificado na Listagem 12.

Listagem 12. Busca dos relacionamentos do produto.

```
IndexManager index = db.index();
Index<Node> produtos = index.forNodes("produtos");
IndexHits<Node> indexHits = produtos.get("nome",
   "Playstation 3");

Node noProduto = indexHits.getSingle();
Iterable<Relationship> relationships = noProduto.
    getRelationships(Direction.OUTGOING,
       Relacionamentos.E_UM);

for (Relationship rel : relationships) {
    Node noCategoria = rel.getEndNode();
    System.out.println(noCategoria.getProperty("nome"));
}

tx.success();
```

Pesquisa de dados no grafo através dos traversals

Algumas pesquisas tornam-se mais complicadas de serem realizadas através da busca simples pelo relacionamento, como, por exemplo, quando precisamos saber quais pessoas compram que produtos. Nesse caso, tomando como ponto de partida um cliente chamado João, primeiramente, precisamos conhecer as compras realizadas por ele. Nesse instante, estamos no primeiro nível de profundidade do nosso processo de percorrer o grafo. Em seguida, já que conhecemos as compras, podemos saber os produtos comprados por ele, o que nos leva ao segundo nível de profundidade da pesquisa.

O próximo passo é sabermos de quais outras compras eles fazem parte, o que nos leva ao terceiro nível de profundidade. E, por fim, já sabendo as compras que esses produtos fizeram parte, temos como descobrir os outros produtos que fazem parte dessas compras. Repare que temos que realizar uma pesquisa em um alto nível de profundidade do nosso grafo, o que não necessariamente é uma pesquisa simples de ser feita em um banco de dados relacional, mas totalmente passível de ser realizada.

Note que para tanto precisamos percorrer o grafo atrás dessas informações, o que é um processo conhecido como Traversal.

O Neo4j permite a realização de traverses no grafo através da classe TraversalDescription, que faz o papel de um builder para um processo de traverse. No caso, precisamos indicar quais relacionamentos devem ser percorridos e de qual profundidade os dados nós deverão ser devolvidos. Dessa forma, para realizarmos a pesquisa descrita anteriormente, teremos um código similar ao da Listagem 13.

Listagem 13. Descrição e realização de um processo de traverse.

```
TraversalDescription td = Traversal.description()
.breadthFirst().relationships(Relacionamentos.
    REALIZOU, Direction.OUTGOING)
.relationships(Relacionamentos.
    CONTEM, Direction.OUTGOING)
.relationships(Relacionamentos.
    CONTEM, Direction.INCOMING)
.relationships(Relacionamentos.
    CONTEM, Direction.OUTGOING)
.evaluator(Evaluators.atDepth(4));

Node noInicial = db.getNodeById(42);

Iterable<Node> traverse = td.traverse(noInicial).nodes();
for (Node node : traverse) {
    System.out.println(node);
}
```

Realização de pesquisas com a query language Cypher

Apesar do poder existente na escrita de traversals através da API do Neo4j, muitos desenvolvedores não se sentem confortáveis escrevendo suas consultas em código Java, o que é algo totalmente compreensível. Além disso, a escrita de consulta em código Java pode atrapalhar também o processo de operações, no qual, administradores de banco de dados precisam manipular os dados existentes no banco. Justamente para satisfazer os diversos gostos, o Neo4j possui também uma linguagem de consulta que permite pesquisar informações dentro dos grafos, no caso, o Cypher.

Uma simples query com o Cypher para buscar um nó no banco de dados através de seu id pode ser escrita em poucos caracteres, como visto na Listagem 14.

Listagem 14. Query simples com o Cypher.

```
start n=node(1) return n
```

Na pesquisa, indicamos que queremos buscar um nó, cujo id é 1 e ele deverá ser retornado pela pesquisa. No entanto, podemos facilmente evoluir essa pesquisa para realizar tarefas mais complexas, como, por exemplo, pesquisar através de algum índice, o que é exemplificado na Listagem 15, ao mostrar como buscar a categoria videogame através de seu índice.

Listagem 15. Query por índice através do Cypher.

```
start n=node:categorias(nome="Video Game") return
n
```

Apesar de demonstrar a sintaxe de uma query Cypher, as duas pesquisas acima não exemplificam todo o poder da linguagem de pesquisas, por realizarem pesquisas muito simples. Em nosso caso, desejamos transformar a mesma pesquisa que escrevemos com a API de Traversals, com o Neo4j.

Para realizar pesquisas que são definidas através de caminhos no grafo, é necessário adicionar a cláusula match à pesquisa, indicando quais relacionamentos serão trabalhados na pesquisa e em quais sentidos, como na Listagem 16.

Listagem 16. Pesquisa complexa com Cypher.

```
start n=node(42) match (n)-[:REALIZOU]->(compras)-
[:CONTEM]->(produtosComprados)<-[:CONTEM]-
(outrasCompras)-[:CONTEM]->(outrosProdutos) return
outrosProdutos
```

Note que na pesquisa é indicado através de caracteres como as setas para direita e esquerda se os relacionamentos são de chegada ou saída em um determinado nó. No caso dessa pesquisa, é possível ler a cláusula match como: "onde n realizou compras que continham produtosComprados que estavam em outrasCompras que por sua vez continham outrosProdutos".

Para realizarmos a consulta dentro da aplicação Java, é necessário criar uma instância de ExecutionEngine, passando o db como parâmetro, em seguida, invocando o método execute no objeto criado, passando a String da consulta como parâmetro, como exemplificado na Listagem 17, onde o resultado também é pego através do Iterator retornado, e exibido na console.

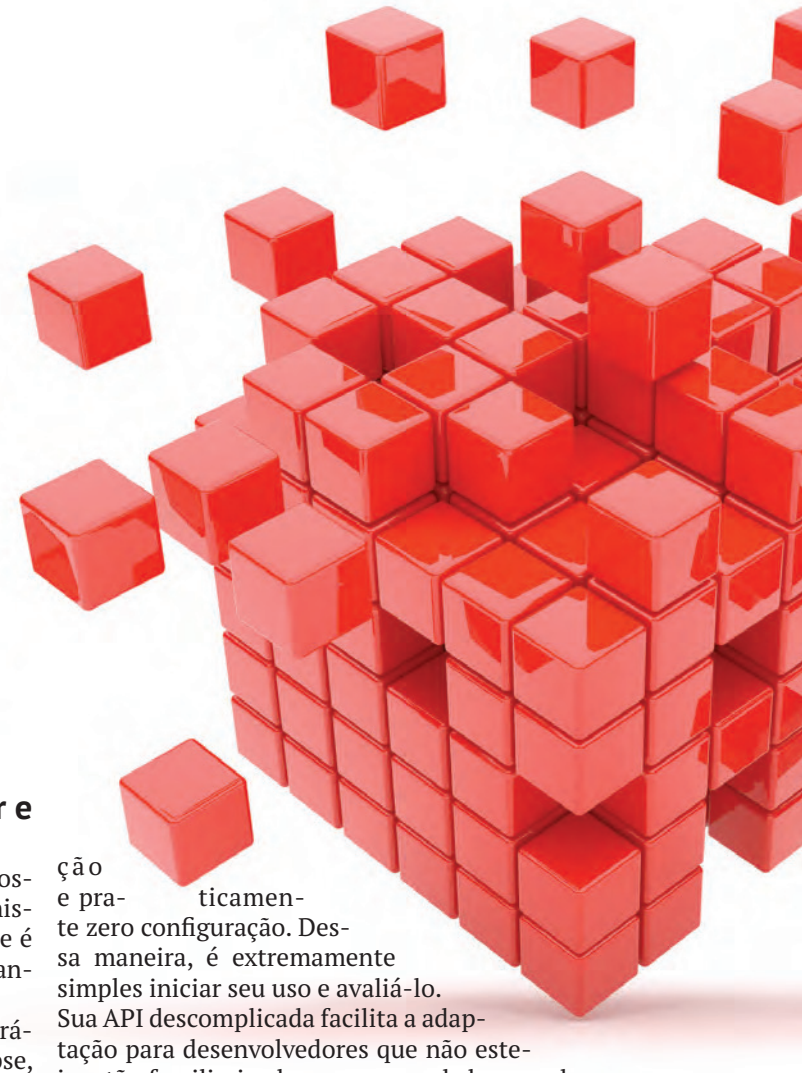
Listagem 17. Pesquisa complexa com Cypher dentro de código Java.

```
ExecutionEngine engine = new ExecutionEngine(db);
ExecutionResult result = engine
      .execute("start n=node(9) match (n)-[:REALIZOU]-
      >(compras)-[:CONTEM]->(produtosComprados)<-
       [:CONTEM]-(outrasCompras)-[:CONTEM]-
      >(outrosProdutos) return outrosProdutos");
```

```
Iterator<Node> results = result.
columnAs("outrosProdutos");

while(results.hasNext()) {
    Node no = results.next();
    System.out.println(no.getProperty("nome"));
}
```

Por fim, a linguagem Cypher permite que se utilize recursos muito comuns aos bancos de dados relacionais e ao SQL, como group by, funções de agregação como SUM e AVG, cláusulas where caso haja a necessidade de filtrar a pesquisa pelo valor de alguma propriedade de um nó e assim por diante.



Ferramentas auxiliares, o Neo4j Server e escalabilidade

Além de toda a parte da persistência, o Neo4j possui ferramentas periféricas, que permitem a administração do banco de dados, como o Neo4j Shell, que é uma aplicação que possibilita a manipulação do banco de dados através do terminal.

Para usuários que preferem uma visualização gráfica, pode-se utilizar também a ferramenta Neoclipse, onde é possível visualizar os nós e relacionamentos do banco de dados, consultar índices, e manipular as informações com poucos cliques. O Neoclipse, por padrão, não vem com o pacote do Neo4j, mas pode ser baixado em separado na própria página de downloads do projeto.

Outro ponto importante é que no decorrer deste artigo foi mostrado o Neo4j embedded, no qual o desenvolvedor simplesmente adiciona o jar ao seu projeto e utiliza-o como se fosse apenas mais uma biblioteca em seu projeto. No entanto, caso seja necessário utilizar o Neo4j em outras plataformas, que não Java, a distribuição vem com uma versão server, que inicializa um servidor do Neo4j que fica disponível para ser manipulado via chamadas REST, permitindo assim o uso do banco de dados com outras plataformas que não Java. O Neo4j Server também vem com uma interface web de administração, onde é possível manipular e administrar os dados on-line.

Por fim, é importante ressaltar que o Neo4j pode suportar até bilhões de nós e relacionamentos armazenados em seu banco de dados e, ainda assim, atingir alta performance ao percorrer grafos, tudo possível ao modelo de dados e armazenamento interno otimizado para esse tipo de operação.

Considerações finais

O Neo4j é um banco de dados de simples instalação e praticamente zero configuração. Dessa maneira, é extremamente simples iniciar seu uso e avaliá-lo. Sua API descomplicada facilita a adaptação para desenvolvedores que não estejam tão familiarizados com o uso de bancos de dados não-relacionais.

A realização de consultas também é outro ponto que merece destaque, permitindo consultas complexas com poucas linhas de código, caso seja feita através da API Java, ou de maneira expressiva, como se tivesse desenhando o caminho de um grafo, através da linguagem Cypher.

A maneira diferente de modelar as informações é um ponto que pode causar bastante estranheza no começo, como um dia a modelagem relacional também causou nos desenvolvedores, mas com o hábito, tende a se tornar algo natural.

/referências

> *Neo4j – http://neo4j.org*

> *Neo4j Downloads – http://neo4j.org/download*

> *Blog Neo4j – http://blog.neo4j.org*

> *Cypher's internal – http://ahalmeida.com*

> *Artigo sobre o Neo4j no Infoq – http://www.infoq.com/articles/graph-nosql-neo4j*

> *Apresentação sobre o Neo4j – http://www.infoq.com/presentations/emil-eifrem-neo4j*