Sumário

Introdução

Com o advento da web 2.0, várias tendências surgiram na web, e as redes sócias explodiram. Com um número espantoso de usuários e conteúdo à armazenar, o modelo relacional de banco de dados se torna um modelo inviável para as aplicações.

Com esse cenário, e a busca por alternativas ao problema de gerenciar a grande quantidade de conteúdo, os bancos de dados não-relacionais, ou Bancos de dados NoSQL, se mostram uma ótima alternativa, e também uma ótima área de pesquisa.

A ideia é mostrar como é viável e apropriado o uso desse tipo de banco de dados, para aplicações que tem como objetivo a interação e integração de conteúdo gerado por usuários, assim como integração com redes sociais.

Parte 1

1. Problema

1.1. Delimitação do Problema

Modelo Relacional

Introdução ao Modelo Relacional

O modelo relacional foi criado no início dos anos 1970, e desde então se tornaram o padrão da indústria de softwares, e têm sido utilizados em larga escala por Sistemas Gerenciadores de Bancos de Dados. Ele se tornou padrão, substituindo os modelos hierárquicos e de rede. Os exemplos mais comuns de bancos de dados relacionais são o SQL Server, o PostgreSQL, MySQL, entre outros.

Os bancos de dados relacionais são estruturados através de tabelas, as quais por sua vez são compostas por linhas (ou tuplas) e colunas (ou atributos). O modelo relacional visa estruturar os dados seguindo esse modelo. Outra característica desse modelo é a utilização de restrições de integridade. As restrições mais comuns são as chaves primárias, e chaves estrangeiras.

As chaves primárias têm o objetivo de identificar unicamente uma tupla na tabela, enquanto uma chave estrangeira faz referência à uma dependência de atributos, geralmente, em uma outra tabela.

O modelo Relacional tem um processo de Normalização, com o objetivo de garantir o projeto adequado de tabelas, separando os dados de elementos distintos em tabelas distintas, relacionando-as através de chaves.

O modelo Relacional adotou como linguagem de definição, manipulação e consulta, a SQL (Structured Query Language).

SQL – Structured Query Language

Desenvolvida originalmente pela IBM, o SQL é uma linguagem declarativa para bancos de dados relacionais, baseada na álgebra relacional. Sua simplicidade e alto poder de expressão a tornaram um padrão, e também a linguagem de consulta de dados mais utilizada no mundo, ajudando a consolidar o domínio do Modelo Relacional.

A simplicidade do SQL pode ser demonstrada utilizando-se um exemplo para selecionar todos os registros e seus respectivos atributos, dada uma tabela chamada ALUNO.

```
SELECT * FROM ALUNO
```

Podemos ver a simplicidade do SQL ao utilizar condições para obter somente registros que atendem determinada condição, como por exemplo alunos que contém o texto ‘ander’ no nome.

```
SELECT * FROM ALUNO WHERE nome LIKE ‘%ander%’
```

Limitações do Modelo Relacional

Devido ao intenso e acelerado crescimento do número de aplicações, recursos e tudo o que tange sistemas computacionais, o volume de dados também cresceu de forma extraordinária, como por exemplo, o volume de dados do Google, que provavelmente já excede os Petabytes ($10^{15\ bytes}$).

Em cenários como esse, onde o gerenciamento do grande volume de dados é necessário, o Modelo Relacional se torna problemático e ineficiente. De um modo geral, os principais problemas estão relacionados com a dificuldade de conciliar o modelo a demanda por escalabilidade.

1.2. Objetivos da Pesquisa

Temos por objetivo principal demonstrar como é possível e viável o uso de bancos de dados não-relacionais, para solucionar os problemas do modelo relacional, e as novas necessidades que a web 2.0 trouxe.

1.3. Justificativa

O movimento NoSQL propõe soluções que parecem ser a melhor opção para resolver os problemas apresentados, como listado nos tópicos abaixo.

Modelo Não Relacional

Buscando uma solução para a problemática do Modelo Relacional

A estrutura utilizada pelos modelos relacionais, considerado uma solução até então, passou então a ser um problema a ser contornado, e logo as soluções propostas tinham como objetivo flexibilizar essa estrutura. Várias soluções apresentadas pareciam regredir ao simples gerenciamento de arquivos.

Por uma lado as soluções perdiam grande parte das regras de consistência presentes no Modelo Relacional, porém tinham a vantagem de ganhar em performance, tornando flexíveis os sistemas de bancos de dados, para atender as necessidades de cada companhia.

NoSQL – Not Only Structured Query Language

O termo NoSQL foi utilizado pela primeira vez em 1998 por Carlo Strozzi, fazendo referência ao seu projeto de um banco de dados de código aberto, que ainda utilizava um modelo relacional, porém não possuía interface SQL. Ainda de acordo com Carlo Strozzi, o movimento NoSQL "é completamente distinto do modelo relacional e, portanto, deveria ser mais apropriadamente chamado 'NoREL' ou algo que produzisse o mesmo efeito".

Posteriormente em 2009, foi utilizado por Eric Evans, em um evento organizado pela Last.fm para discutir bancos de dados open source distribuídos. Desde então passou a ser utilizado para representar soluções onde o Modelo Relacional não

apresenta performance adequada. O propósito do NoSQL portanto, não é substituir o modelo relacional como um todo, mas sim quando é necessária uma maior flexibilidade do banco, para grande volumes de dados, necessidade de escalabilidade, necessidade de respostas mais rápidas dos aplicativos, entre outros.

O Surgimento de implantações NoSQL

Uma das primeiras implementações de um sistema não-relacional, que teve um papel muito importante na explosão do movimento NoSQL, surgiu em 2004 quando o Google lançou o Google BigTable, um banco de dados proprietário, de alta performance, com o objetivo de fornecer maior escalabilidade e disponibilidade. Vários projetos da Google utilizam esse sistema, como por exemplo, o Google Web Indexing (Sistema de indexação do Google), Google Earth, entre outros, demonstrando o poder do sistema.

Outra importante implementação foi realizada em 2007 pela Amazon. Denominado Dynamo, a implementação da Amazon tinha como característica básica ser um banco de dados não-relacional de alta disponibilidade, utilizado pelos web services da Amazon.

Outros grandes nomes que tem importante participação no movimento NoSQL são:

- Yahoo: Hadoop com HBase e Sherpa – Hadoop é um framework para processamento de dados em larga escala, que se utiliza do paradigma de programação Map Reduce para realizar computação distribuída em clusters contendo centenas e até milhares de máquinas. HBase é o banco de dados que está por trás desse poderoso framework.
- Facebook e Digg: Cassandra – Projetado para ser um banco de dados de alta disponibilidade para lidar com grandes volumes de dados. No início de 2010 desbancou o MySQL como banco de dados do Twitter.
- LinkedIn: Voldemort – Projetado pelo LinkedIn para resolver problemas de alta-escalabilidade.
- Engine Yard: MongoDB – Banco de dados orientado documentos, que provê alta-escalabilidade, alta performance, entre outros atributos.
- Twitter: FlockDB – Banco de dados orientado à grafos distribuído, de alta performance, e tolerante à falhas.
- Apache: CouchDB – Banco de dados orientado à documentos, que busca replicação e estabilidade horizontal.

1.4. Hipóteses e variáveis

Temos por hipótese então que os sistemas de bancos de dados não-relacionais, podem ser a solução para os problemas encontrado, porém ainda existem vários tipos de

bancos de dados à serem considerados, e ainda algumas limitações e mitos que surgem em toda tecnologia emergente, como mostrado nos tópicos abaixo.

Classificação dos Bancos de Dados NoSQL

No que tange à classificação dos bancos de dados NoSQL, utilizaremos 3 características principais:

- Arquitetura: Distribuidos / Não Distribuídos;
- Armazenamento: Memória / Disco / Configurável;
- Modelo de Dados: Chave-Valor(Key-value) / Documento / Colunas / Grafo

Arquitetura

Distribuídos

Os distribuídos tomam a responsabilidade pela partição dos dados e pela sua replicação.

Exemplos:

- Amazon;
- Voldemort;
- BigTable;
- Cassandra;

Não-Distribuídos

Os bancos não-distribuídos deixam a responsabilidade pela partição dos dados e sua replicação com o usuário.

Exemplos:

- MemCacheDB;
- Neo4j;
- Amazon SimpleDB.

Armazenamento

Os bancos podem ser divididos também, entre aqueles que guardam os dados diretamente no disco, e os que armazenam na memória.

Memória

Exemplos:

- Scalaris;
- Reddis.

Disco

Exemplos:

- CouchDB;
- MongoDB;

- Neo4j.

Configurável

Exemplos:

- BigTable;
- Cassandra;
- HBase.

Modelo de dados

Nessa classificação, os bancos de dados são separados pelo seu núcleo, ou seja, como ele trabalha com os seus dados.

Chave-valor

Esse é o tipo de banco de dados NoSQL mais simples, os seus conceitos são uma chave e um valor para essa chave. Esses tipos de bancos de dados são o que fornecem maior escalabilidade.

Exemplos:

- Dynamo;
- Voldemort.

Documento

Baseado em documentos XML ou JSON, podem ser localizados pelo seu id único ou por qualquer registro que tenha no documento.

Exemplos:

- CouchDB;
- MongoDB.

Documento

Fortemente inspirados pelo BigTable, do Google, eles suportam várias linhas e colunas, além de permitir subcolunas.

Exemplos:

- BigTable;
- HBase;
- Cassandra.

Grafo

Com uma complexibilidade maior, esses bancos de dados guardam objetos, e não registros como os outros tipos de NoSQL. A busca desses itens é feita pela navegação desses objetos.

Exemplos:

- HyperGraphDB;
- FlockDB;
- Neo4j.

Mitos Sobre o NoSQL

Toda tecnologia emergente, passa por um grande processo de ‘hype’, e muitas vezes ficam conhecidas como buzzwords. Esse fato se deve ao uso indiscriminado de tais tecnologias, e também ao fato de acharem que acharam a tão falada silver bullet, ou bala de prata.

Mas como em desenvolvimento, não existe bala de prata, vamos listar alguns mitos gerados em torno dos bancos NoSQL

NoSQL é escalável

Uma das grandes promessas dos bancos NOSQL consiste em dizer que eles são mais escaláveis que os bancos de dados relacionais. O problema com esta mensagem que é vendida por algumas empresas é que ela não é inteiramente verdade. Dizer que seu sistema escala sozinho é vender um sonho. Ele pode até ser mais fácil de escalar se comparado a outras soluções, mas ainda sim exigirá algum esforço para escalar, e escalar de forma eficiente.

Não precisamos de DBAs

No mundo dos bancos relacionais a figura do DBA (DataBase Administrator) sempre está presente. Com sistemas que tem particularidades para cada fornecedor, os DBAs ficam a cargo de instalar, configurar e manter cada banco de dados em suas particularidades. Muita gente diz que quando se trabalha com NoSQL não precisamos de DBAs.

Talvez não no sentido tradicional, mas ainda vamos precisar de alguém responsável por lidar com o banco e com o acesso aos dados. Esta função pode vir a se tornar parte do trabalho de um desenvolvedor ou se tornar a função full time de alguém no seu time que pode ser até um DBA com conhecimentos em NoSQL. Em aplicações reais em produção muito provavelmente será necessário misturar bancos relacionais e não relacionais, possuir alguém que navegue facilmente nos dois mundos em seu time é uma grande vantagem.

NoSQL é mais econômico

Meia verdade. Muitos fornecedores de NoSQL afirmam que suas soluções vão baratear o custo dos seus clientes. Em parte sim, em algumas situações o custo em usar um banco de dados relacional pode ser proibitivo devido à escala, ou a licenças envolvidas. Existem muitos casos, entretanto, em que uma solução relacional atende perfeitamente todas as necessidades do cliente e ainda sim pode ser considerada barata. Bancos de dados Open Source como MySQL e PosgreSQL são usados sem problemas por um grande número de aplicações, e com sucesso.

Parte 2

2. Procedimentos Metodológicos

2.1. Delimitação do Universo

Nossa pesquisa foi baseada em casos reais, e em aplicações de grande uso social e que apresentavam grandes problemas antes das soluções NoSQL.

2.2. Pressupostos da Pesquisa

Procuramos demonstrar através de casos reais, como o uso da tecnologia de bancos de dados não-relacionais solucionou problemas que surgiram com o crescimento da internet, e demanda de dados.

Apresentamos também como uma tecnologia baseada em grafos, solucionou o problema de uma das maiores redes sociais do mundo, o Twitter.

Estudos de Caso

Abaixo citaremos alguns casos de usos, postados na íntegra, encontrados na Internet. Um deles é o caso do Twitter com FlockDB, e o outro com uma solução para um produto brasileiro, o BooBox.

Twitter

O Twitter armazenda vários grafos de relações entre pessoas: Quem você esta seguindo, quem está seguindo você, entre outros.

Algumas das características desses grafos, tem se mostrado verdadeiros desafios em se encontrar uma forma escalável à medida que crescem.

Para completar um tweet, é necessário buscar por todos os seguidores, e paginá-los rapidamente. Além disso, é necessário gerenciar um grande tráfego de escritas, como adicionar e remover seguidores. Algumas dessas operações necessitam de certa aritmética. Essas necessidades são difíceis de implementar usando um banco de dados relacional tradicional

Um corajoso esforço

Depois de várias tentativas com o uso abusivo de tabelas relacionais, e armazenamento de listas chave-valor. Essas tentativas eram boas em gerenciar operações de escritas, ou boas para paginar resultados gigantes, mas nunca para os dois.

Na tentativa de resolver esses problemas, surgiu a necessidade de tentar algo novo, e esses eram os objetivos:

- Escrever as coisas mais simples para que possam funcionar;

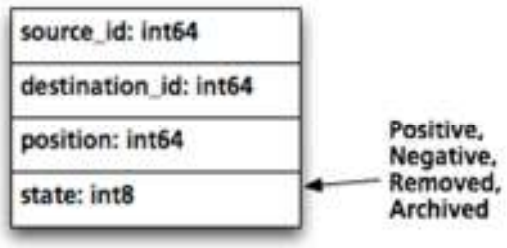
- Usar o sistema off-the-shelf MySQL como sistema de armazenamento, devido ao domínio da tecnologia, e benefícios que a mesma oferece;
- Permitir escalonamento horizontal, para poder adicionar mais recursos, à medida que a necessidade aumentar;
- Permitir operações de escritas sem uma ordem pré-determinada, ou processar várias vezes.

FlockDB foi a solução desenvolvida para solucionar essa problemática.

O Resultado do Esforço

FlockDB é um banco de dados que armazena grafos, mas que não é otimizado para operações de caminhar no gráfico. Em contra-partida, ele é otimizado para grandes listas de adjacência, rápidas operações de leitura/escrita, e conjuntos de consultas aritméticas paginàveis.

Ele armazena os grafos como conjuntos de arestas entre nós indentificados por inteiros de 64bits. Para um grafo de rede social, os IDs dos nós seriam usados como os IDs dos usuários, para um grafo de tweets favoritos, os IDs dos nós podem ser os IDs dos tweets.



Quando uma aresta é deletada, a linha não é realmente deletada do MySQL, ao invés disso, ela é marcada com um estado de ‘Deletado’ e armazenado com uma chave composta do ID, estado e posição, permitindo que seja restaurado posteriormente. Este tipo de otimização, permite que o MySQL trabalhe de forma extremamente eficiente, provendo uma ótima performance.

Experimente

O flockDB é atualmente um banco de dados de código aberto, que pode ser obtido no repositório github: http:github.com/twitter/flockdb.

Boo-Box

Esse é um caso real, que aconteceu com o produto boo-box, um sistema de publicidades para mídias sociais, onde um banco de dados NoSQL, foi a solução.

Abaixo segue o artigo da autoria de Felipe Vieira, Consultor de Tecnologia e Scrum Master na boo-box, que foi publicado na iMasters. O link se encontra na seção de referências.

Trabalho na boo-box e temos uma experiência bem interessante com bancos de dados NoSQL. Os cases abaixo foram apresentados no The Developer's Conference 2010 e são exemplos reais de como utilizamos o Redis em nosso sistema de tecnologia para exibição de anúncios em múltiplos websites.

Compartilhar estas soluções é uma das maneiras de agradecer à comunidade de desenvolvedores por usarmos software livre, difundir o conhecimento criado na empresa e melhorar nossa própria ferramenta.

Bancos NoSQL, entende-se "Not only SQL", surgiram da necessidade de escalar bancos de dados relacionais com propriedades ACID em projetos web de alta disponibilidade que operam em larga escala. Suas principais características são alta performance, escalabilidade, fácil replicação e suporte a dados estruturados.

Este rompimento com os padrões SQL causa sempre grande repercussão e muitas discussões carregadas de sentimentos e emoções, mas a verdade é que os bancos de dados relacionais ainda servem para resolver muitos problemas que nem sempre (veja bem, nem sempre) poderão ser resolvidos com bancos NoSQL, como por exemplo:

1. Necessidade de forte consistência de dados, tipagem bem definida, etc;
2. Pesquisas complexas que exigem um modelo relacional dos dados para realizações de instruções e operações de junção, por exemplo;
3. Dados que excedam a disponibilidade de memória do servidor, por mais que possamos utilizar swap, ninguém quer prejudicar a performance neste caso.

Ao escolher seu banco de dados, o importante é considerar as funções e características específicas do sistema. Os bancos de dados NoSQL podem ser utilizados especialmente para funções descritas neste artigo. Vamos, aqui, abordar particulamente a nossa experiência com o Redis.

Sobre o banco de dados NoSQL Redis

O NoSQL Redis, que atualmente está na versão 2.0.1, é definido como advanced key-value store. Seu código é escrito em C sob a licença BSD e funciona em praticamente todos sistemas POSIX, como Linux ou Mac OS X. Ele foi idealizado e executado por @antirez para escalar o sistema da empresa LLOOGG. Hoje o repósitório é mantido por uma imensa comunidade e patrocinado pela VMWARE.

A simplicidade de operar um banco apenas setando o valor e uma chave continua, entretanto, diferente de soluções famosas como o memcached, podemos fazer diversas operações na camada das chaves, além de contar com um punhado de estruturas de dados.

Além de salvar strings na memória, também é possível trabalhar com conjuntos, listas, ranks e números. De maneira atômica, pode-se fazer operações de união, intersecção e diferenças entre conjuntos, além de trabalhar com filas, adicionando e removendo elementos de maneira organizada.

Assim como outros bancos NoSQL este projeto é completamente comprometido com velocidade, pouco uso de recursos, segurança e opções de configurações triviais para ganhos de escalabilidade. Para manter a velocidade dos dados com garantia de persistência, de tempos em tempos (ou a cada n mudanças) as alterações são replicadas, de maneira assíncrona, da memória RAM para o disco.

Agora, vamos aos cases. Dentro de tantas possibilidades, mostraremos algumas soluções do sistema boo-box utilizando o Redis.

Armazenamento de sessões de usuários

Este é um modelo muito simples de como utilizar o Redis para salvar as informações da sessão de um usuário.

Para cada sessão, gera-se uma chave que é gravada no cookie do navegador. Com essa chave, o sistema tem acesso a um hash com informações desta sessão: status do login, produtos e publicidades clicadas, preferências de idioma e outras configurações temporais, que perdem a validade após algumas horas.

O benefício de não guardar tais informações de sessão diretamente no cookie é evidente: ganhamos a segurança de integridade dos dados, não correndo o risco de algum usuário malicioso modificá-los. Com o Redis, utilizamos operações simples de get/set para acessar estes dados diretamente da memória do servidor (ou servidores, caso exista mais de um), sem desperdício de recursos, graças ao eficiente sistema de expiração promovida por este NoSQL.

O algoritmo de expiração não monitora 100% das chaves que podem expirar. Assim como a maioria dos sistemas de cache as chaves são expiradas quando algum cliente tenta acessá-la. Se a chave estiver expirada o valor não é retornado e o registro é removido do banco.

Em bancos que gravam muitos dados que perdem a validade com o tempo, como neste exemplo, algumas chaves nunca seriam acessadas novamente consequentemente elas nunca seriam removidas. Essas chaves precisam ser removidas de alguma maneira, então a cada segundo o Redis testa um conjunto randômico de chaves que possam estar expiradas. O algoritmo é simples, a cada execução:

```
Testa 100 chaves com expiração setada.
Deleta todas as chaves expiradas.
Se mais de 25 chaves forem inválidas o algoritmo recomeça do 1.
```

Essa lógica probabilística continua a expirar até que o nosso conjunto de keys válidas seja próximo de 75% dos registros.

Cache de produtos de terceiros

Todo dia a boo-box exibe para a audiência milhões de produtos - de diferentes e-commerces - vinculados ao conteúdo de publishers. Os e-commerces fornecem APIs, e através delas é possível buscar produtos para serem mostrados em nossas vitrines.

Num modelo ideal, cada requisição de uma vitrine boo-box faria contato com as APIs dos e-commerces parceiros, em busca de produtos compatíveis com o conteúdo em questão. Mas no mundo real da publicidade online, velocidade e escalabilidade são premissas essenciais para a qualidade de produto e, portanto, requisições síncronas a tais APIs tornariam o processamento lento demais.

Portanto, essas operações são cacheadas num banco Redis. Separamos os e-commerces em bancos distintos e obtemos os produtos segundo a keyword que foi utilizada nas buscas de todas as vitrines da rede boo-box.

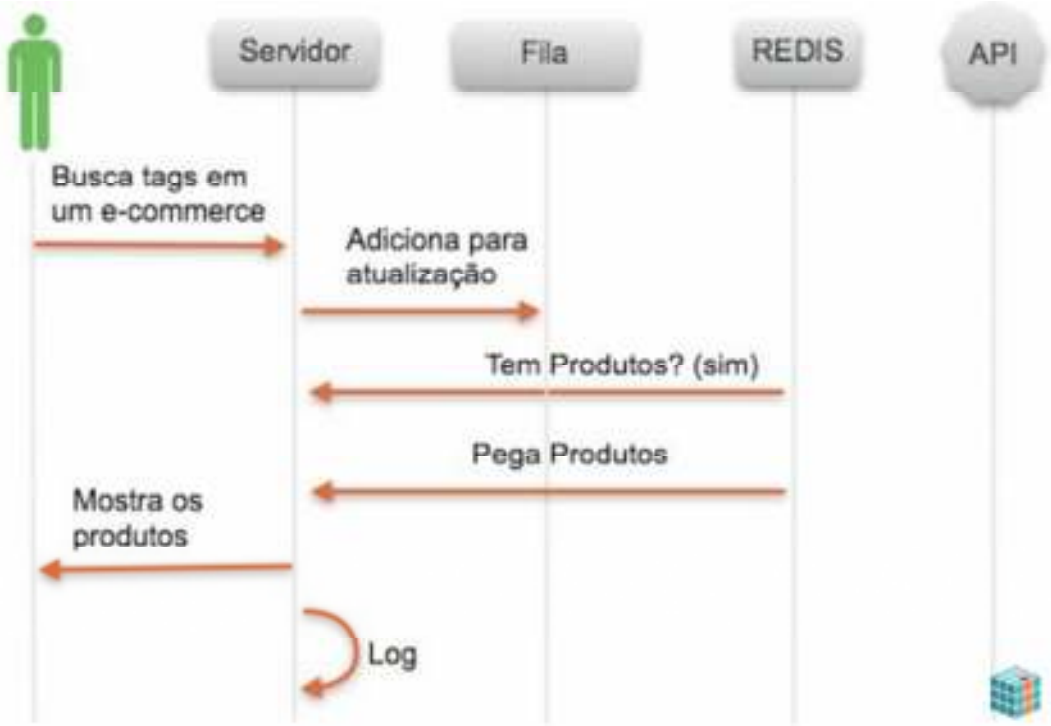


Figura 1 - Execução do cache de produtos quando há resultados para a tag solicitada.

Porém, sempre existe aquele usuário com poucos acessos e com tags que não são tão populares. Neste caso, tentamos fazer a consulta diretamente da API (com um tempo limite pequeno para não complicar o sistema). Caso não encontremos nenhum produto para esta tag, podemos, como já foi dito acima, buscar chaves similares para mostrar nas vitrines deste Publisher, enquanto um evento paralelo é acionado para adicionar esta tag no cache sem restrições de tempo. Assim, em uma próxima visualização, os produtos já estarão quentinhos no cache! Veja como esse luxo pode ser ilustrado:

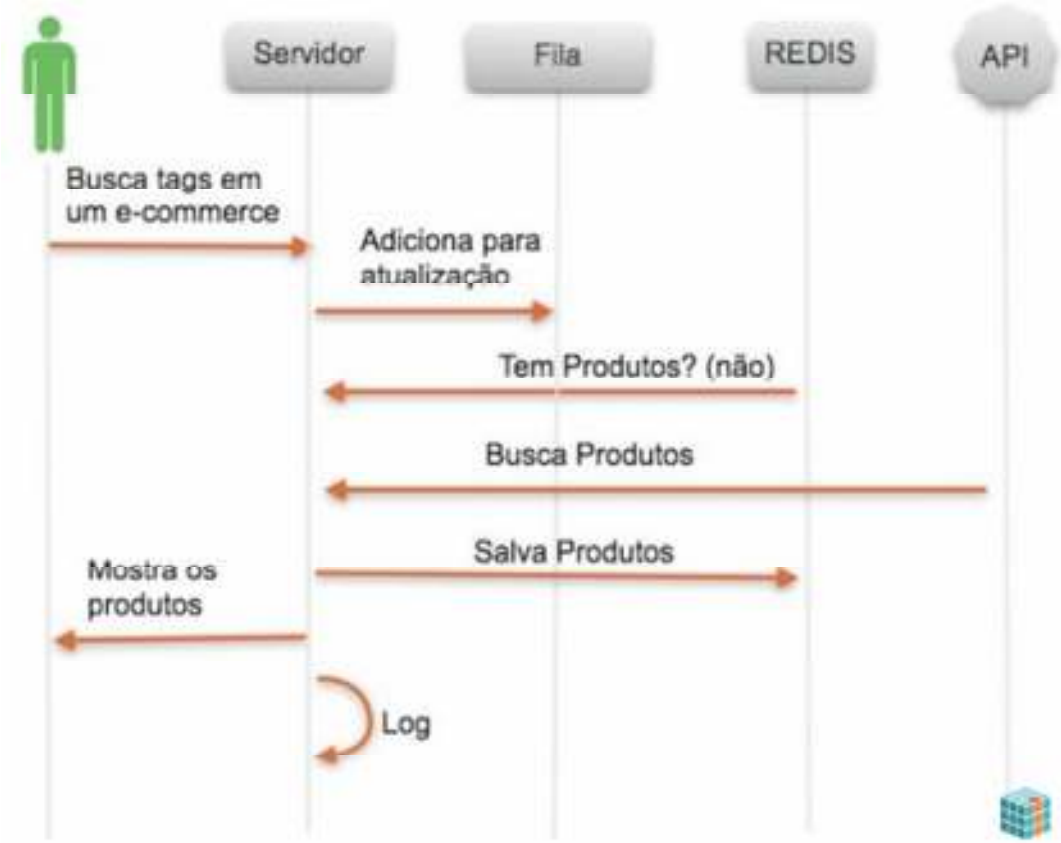


Figura 2 - Quando não havia resultados para a tag solicitada, buscávamos os produtos na API, entregávamos ao usuário e gravávamos os resultados no cache.

A separação de e-commerce em bancos distintos facilita as operações de busca por keywords. O Redis, por padrão habilita 16 bancos que podem ser utilizados separadamente e, por consequência, escalados separadamente. Com as keys de um e-commerce isoladas, podemos buscá-las através de padrões parecidos com regexp e, com sabedoria, isso pode ser um excelente recurso, mas também pode ser um problema tendo em vista que a complexidade desta função é O(n) onde n é o numero de chaves no banco utilizado.

Diagrama de sequência dessa funcionalidade:

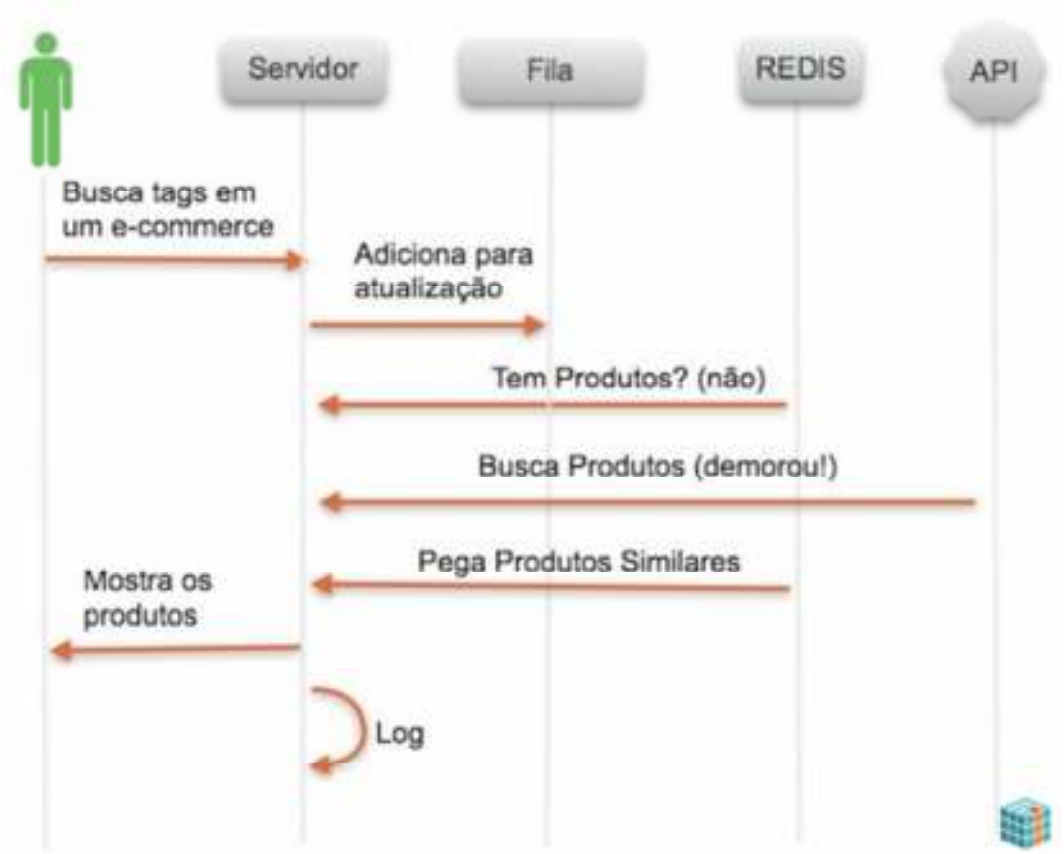


Figura 3 - Diagrama de sequência dessa funcionalidade

Quando não há resultados para a tag solicitada no cache de produtos, exibimos produtos similares, depois buscamos pelos produtos exatos no e-commerce e os entregamos diretamente do cache na próxima solicitação.

Veja os logs dessa funcionalidade em ação:

```
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
the_velvet_underground instead underground 0.012
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
pushing_daisies instead push 0.012
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
deborah_secco instead deborah 0.017
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
oracoes_catolicas instead catolica 0.017
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
uma_linda_mulher instead linda 0.012
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
lutaram instead lutar 0.016
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
videogame_wii instead videogame 0.017
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
mini_craque_prostars instead craque 0.017
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
apple_ipod_shuffle_1_gb_silver instead apple 0.005
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
motocultivador_tratorito_branco_diesel instead motocultivador 0.017
```

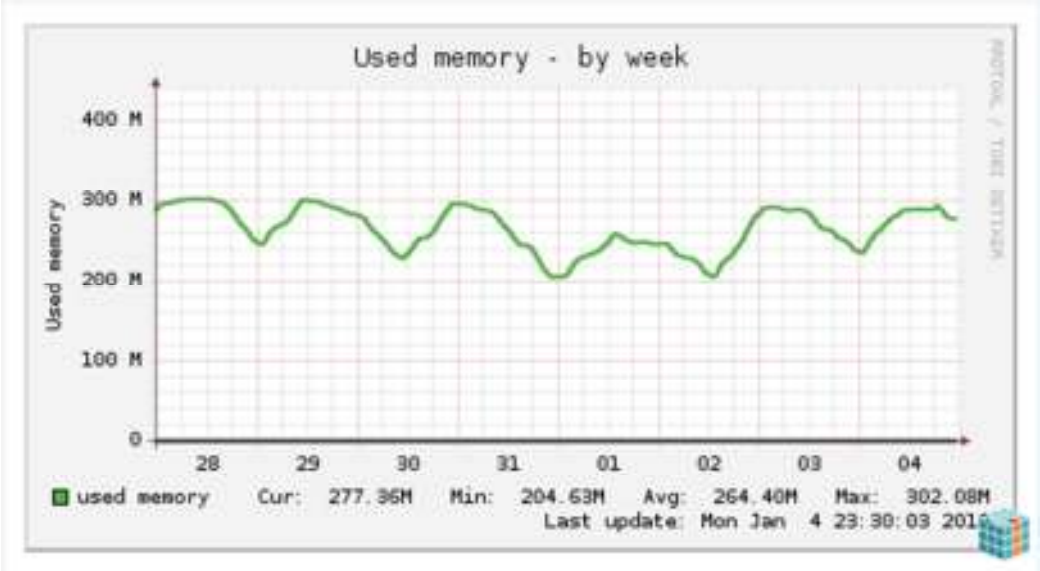
```
merb : worker (port XXXX) ~ DEBUG get similar keys from redis using
suporte_para_bicicleta_automovel instead automovel 0.005
```

Esse cache é muito custoso e não é remontado com facilidade. Portanto, assim como a primeira solução, além da velocidade a persistência dos dados em disco é imprescindível. A expiração, neste caso, também é muito importante, pois mudanças nos catálogos ou nos preços do produto acontecem com frequência. Mesmo com uma expiração curta, não é interessante esperar que produtos populares como videogames, celulares e afins saiam do cache. Para evitar isto, consultamos, a cada requisição, o tempo de vida restante deste produto no cache. Caso ele esteja próximo de expirar, um evento assíncrono é acionado para atualizar este produto na API do e-commerce em questão. Na apresentação Usando Redis para otimizar o sistema boo-box, feita na Campus Party Brasil 2010, mostramos detalhes deste fluxo.

Os resultados desta solução foram supreendentes como mostram as imagens abaixo. Uma queda no tempo de resposta de parte do sistema, uma economia insana de recursos que levou ao desligamentos de servidores e redução de perfil de máquinas, assim como a melhoria da manutenibilidade do processo todo.

Tempo de resposta do sistema:

Mais velocidade no sistema após a implementação do Cache de Produtos:



Todo dia a boo-box exibe milhões (milhões!) de produtos de diversos e-commerces. Com o Redis cacheamos tudo com apenas 300 MB de RAM.

Busca em catálogos de produtos de terceiros

Algumas vezes temos acesso a um catálogos de produtos de um e-commerce por meio de um arquivo XML. Diariamente, este arquivo é atualizado pelo parceiro, parseado e salvo no Redis do sistema boo-box. Além de armazenados, esses produtos são também organizados para a realização de consultas. Modelar NoSQL é um pouco diferente do que modelar bancos relacionais. Não existem joins ou queries complexas, a estrutura é organizada para a pesquisa que deve ser feita.

Vamos a um exemplo prático:

Supondo que o e-commerce disponibilize o seu catálogo de produtos em um arquivo XML que tem a estrutura abaixo:

```
<catalogo>
 <produto>
 <cod>G28T49200F2</cod>
 <nome>Quintette Du Hot Club De France DJANGO REINHARDT</nome>
 <preco>51,90</preco>
 <descricao>CD Quintette Du Hot Club De Frace  1938-1939 - Importado
Appel Indirect;Billets Doux;Japanese Sandman;Three Little
Words;Stompin´ at Decca;Souvenirs;Sweet Georgia Brown;Tornerai
(J´attendrai);If I Had You;It Had to Be You;Nocturne;Black and
White;Night and Day;Honeysuckle Rose;Swing;I Wonder Where My Baby is
Tonight;Why Shouldn´t I?;Them There Eyes</descricao>
 <imagem>http:www.ecomm.com.br/imgs/cds/cover/img2/284922.jpg</imagem>
 <url>http:www.ecomm.com.br/produto/2/284922/</url>
 <categoria>Musica</categoria>
 </produto>
</catalogo>
```

Dado que o campo "cod" é uma referência única para este produto neste catálogo, poderíamos transformá-lo em chave e salvar um hash com todas as propriedades do produto, como mostra o código abaixo:

```
 catalogo_xml.each do |product_xml|
  # Uma vez que transformamos o xml do produto em um hash...
  product = parser_to_hash(product_xml)

  # Para cada identificador cod salvamos todas as
  # informações deste produto
  redis[:product].set(product[:cod], Marshal::dump(product))
end
```

Bonito e inútil! Ter um hash referenciado por um id a princípio não ajudaria a fazer buscas, entretanto esse será o nosso banco principal que guardará todas as informações dos produtos. Se pensarmos em como indexar estes produtos por uma busca mais trivial (por exemplo, nome) devemos criar um novo banco e teríamos que alterar o nossa função de parser para organizar estes produtos ou melhorar suas chaves por nome:

```
catalogo_xml.each do |product_xml|
  # Uma vez que transformamos o xml do produto em um hash...
  product = parser_to_hash(product_xml)

  # Para cada identificador cod salvamos todas as
  # informações deste produto
  redis[:product].set(product[:cod], Marshal::dump(product))

  # Para cada nome (que pode ser repetido no catalogo)
  # adicionamos o cod deste produto em uma estrutura de lista
  redis[:name].addmember(slugfy(product[:nome]), product[:cod])
end
```

A função slugfy foi utilizada para evitar que a mesma palavra seja tratada diferentemente por conta de caracteres de acento ou em caixa alta. Esse é um ponto crucial que pode aumentar muito a contextualização da busca. Muitos algoritmos linguísticos podem ser úteis neste caso, mas voltando ao ponto deste case, um método simples de busca para essa indexação seria:

```
def search(keyword)
  keyword = slugfy(keyword.to_s)
  return [] if keyword.empty?
  names = redis.keys("*" + name + "*")
  cods = redis[:name].union(names.join(" "))
  products = []
  cods.each do |cod|
    products << get_product(cod)
  end
  return products
end
```

Legal! Agora podemos buscar por nome em nosso catálogo, mas essa busca é muito engessada. Para melhorá-la, poderíamos buscar por todas as palavras do produto, esteja no título, no nome, na categoria ou até mesmo na descrição.

Aqui temos um ponto importante. Assim como o uso da função slugfy, precisamos definir algumas stopwords para que, nesta função, palavras muito comuns sem valor semântico não atrapalhem a busca. Por stopwords podemos considerar artigos, pronomes, etc.

A estratégia agora é criar um terceiro banco com ids que contenham uma tag específica. Mudaríamos novamente o nosso parser para preencher este banco de busca e definiríamos uma nova função:

```
catalogo_xml.each do |product_xml|
  # Uma vez que transformamos o xml do produto em um hash...
  product = parser_to_hash(product_xml)

  # Para cada identificador cod salvamos todas as
  # informações deste produto
  redis[:product].set(product[:cod], Marshal::dump(product))

  # Para cada nome (que pode ser repetido no catalogo)
  # adicionamos o cod deste produto em uma estrutura de lista
  redis[:name].addmember(slugfy(product[:nome]), product[:cod])

  # Os campos passiveis de busca deste produtos são transformados
  # em uma grande string e
  raw_tags = [slugfy(product[:nome]),slugfy(product[:categoria]),
slugfy(product[:descricao])].join("-")

  # Apos remover palavras que não tem nenhum valor semantico
  # temos as tags deste produto
  tags = remove_stopwords(raw_tags.split("-").uniq)

  # Para cada tag faremos o mesmo que foi feito com o nome
  # salvamos uma lista que organizará todas os produtos que contenham
uma tag em comum
  tags.each do |tag|
    redis[:tags].addmember(tag, product[:cod])
  end
end
```

Para realizar busca com tags, poderíamos fazer uma intersecção entre as tags, dessa forma teríamos os ids que contemplassem esta busca. Vejamos uma maneira simples de fazer isso:

```
def search_tags(keyword)
  tags = remove_stopwords(slugfy(keyword.to_s).split("-").uniq)
  return [] if tags.empty?
  cods = redis[:tags].inter(tags.join(" "))
  products = []
  cods.each do |cod|
    products << get_product(cod)
  end
  return products
end
```

Este esboço de algoritmo pode nos dar uma idéia de como modelar NoSQL. Existem muitas maneiras de melhorar esta busca: quantidade de vezes que a palavra é citada, proximidade de palavras e até mesmo a posição da palavra. Muitos algoritmos de pagerank, por exemplo, podem nos guiar nessas melhorias.

Um ponto importante, levantado pelo @jdrowell é a utilização da busca por regexp:

```
redis.keys("*" + name + "*")
```

Este é sem dúvida o comando que deve ser utilizado com maior cautela, ele pode ser um gargalo devido a sua complexidade. No nosso caso, como a quantidade de keys é fixa, esse comando não pode nos comprometer devido ao tamanho do catálogo de produtos.

Para quem quer se aprofundar neste case, sugiro a leitura do artigo indicado pelo @jdrowell que mostra um case similar de busca de textos utilizando o Redis.

Validação de visualizações e clicks de produtos

Este último é também o mais recente case de utilização do Redis, ainda esta em fase de testes e validação. O adserver da boo-box hoje exibe cerca de 15 milhões de vitrines de produtos e campanhas diariamente. Todas visualizações e clicks são logadas e, apartir destes logs, exibimos estatísticas para os nossos publishers e anunciantes. Salvamos todos os tipos de informação que podemos: as dimensões das peças, dados sobre o usuário que interagiu com a vitrine como IP, navegador e até mesmo o seu perfil de navegação. Sim, nós gostamos de dados!

Dado o volume de informações, esperar a inserção para pegar um id que possa servir de referência para esta tupla em um banco de dados relacional é muito perigoso, e fazer queries para recuperá-los seria também uma tarefa lenta. Gravar os logs em um banco rápido como o Redis e ainda poder acessá-los diretamente do adserver abre um leque de possibilidades. Podemos por exemplo, confirmar a renderização das vitrines pelo navegador, medir o intervalo entre clicks, controlar a veiculação de campanha no decorrer do dia, entre outras tantas coisas.

Ao colocar pela primeira vez essa feature em produção tivemos que trabalhar muito para - acredite - melhorar o tempo de inserção dos dados das vitrines no Redis. Na verdade, foi necessário tunar diversos pontos da infra estrutura (pois utilizavamos um servidor para diversos fins e tinhamos apenas uma grande instancia do Redis que era utilizado por todas as features). Além do mais, estavamos utilizando uma versão antiga do Redis, com uma política de gravação dos dados da memória para o disco que prejudicava demais a performaçe devido a grande quantidade de alterações. Outro fator relevante para prejudicar a velocidade foi o numeros de bytes por objeto salvo e o tempo de serialização e deserialização deles.

Veja o que modificamos para recolocar esse recurso no ar e melhorar o tempo de inserção chegando a 0.001 milésimos ;)

1. Isolamos o servidor e habilitamos instâncias diferentes do Redis em diversas portas. Dessa forma toda a memória e processamento ficou dedicada para este serviço e o numero de núcleos do processador pode ser melhor aproveitado, o sistema passou a escalonar o Redis em diversas CPUs paralelamente.
2. Atualizamos o Redis e modificamos as suas configurações para não gravar os dados em disco tendo em vista que estes dados são voláteis e não são reutilizados.
3. Otimizamos a maneira como salvamos os dados no Redis. Ao invés de objetos complexos salvamos apenas um hash com as informações essenciais da visualização e modificamos o metodo de serialização de YAML para Marshal.

Com essas modificações o tempo de resposta caiu drasticamente, porém, o uso de recurso ainda é uma questão preocupante. Por mais que a tendência da memória RAM seja ficar a cada dia mais barata, esse recurso ainda é muito custoso. Como a quantidade de clicks é menor que de visualizações, muitos dados são gravados no Redis e nunca acessados (neste caso mais de 99% dos objetos). É fundamental previnir estes problemas configurando adequadamente o uso de swap pelo Redis e limpando os dados antigos para liberar memória. Em casos extremos, podemos monitorar atráves das estátisticas do Redis o processo e automatizar as ações de limpeza da memória, previnindo o uso de swap.

No FAQ do projeto existem dicas preciosas para evitar o disperdício de memória primeramente utilizar diversas instancias de 32 bits ao invéz de uma com 64 bits, modificar variáveis de ambiente e escolher o tipo de dados adequado para a sua aplicação pode ajudar a previnir muita dor de cabeça com este gargalo.

Considerações finais

Estes cases são exemplos bem sucedidos ou promissores do experimento de novas tecnologias no sistema boo-box. Muitas vezes, a tentativa de utilizar uma nova tecnologia não é bem sucedida. Já utilizamos, por exemplo, outros bancos NoSQL, que por muito tempo foram uma boa solução mas passaram a ser um gargalo em um novo contexto.

Muitos são os desafios que enfrentamos para garantir qualidade, velocidade e estabilidades em sistemas altamente escaláveis. Conhecer as novas tecnologias, estudar e aplicar novas soluções é um esforço importante e muitas vezes ainda desconhecido. Como dissemos lá no início, compartilhar soluções é uma das maneiras de agradecer à comunidade de desenvolvedores por usarmos software livre, difundir o conhecimento criado na empresa e melhorar nossa própria ferramenta. Contribua :)

3. Referências

Informações Gerais:

http:www.infobrasil.inf.br/
http:www.nosqlbr.com.br/
http:forum.imasters.com.br/

Links para os artigos:

http:blog.boo-box.com/br/2010/usando-banco-de-dados-nosql-redis/
http:engineering.twitter.com/2010/05/introducing-flockdb.html

Agradecimentos:

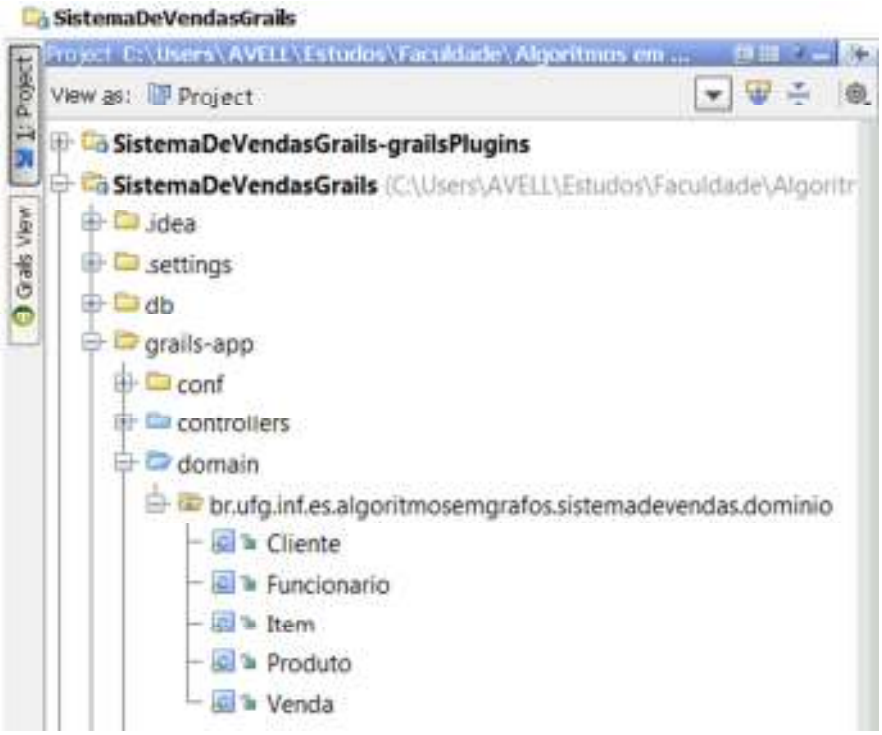
Gostaríamos de deixar o nosso agradecimento ao Consultor de Tecnologia e Scrum Master da empresa, Felipe Luis de Souza Vieira, por permitir a utilização do seu artigo em nosso trabalho.

4. Anexos

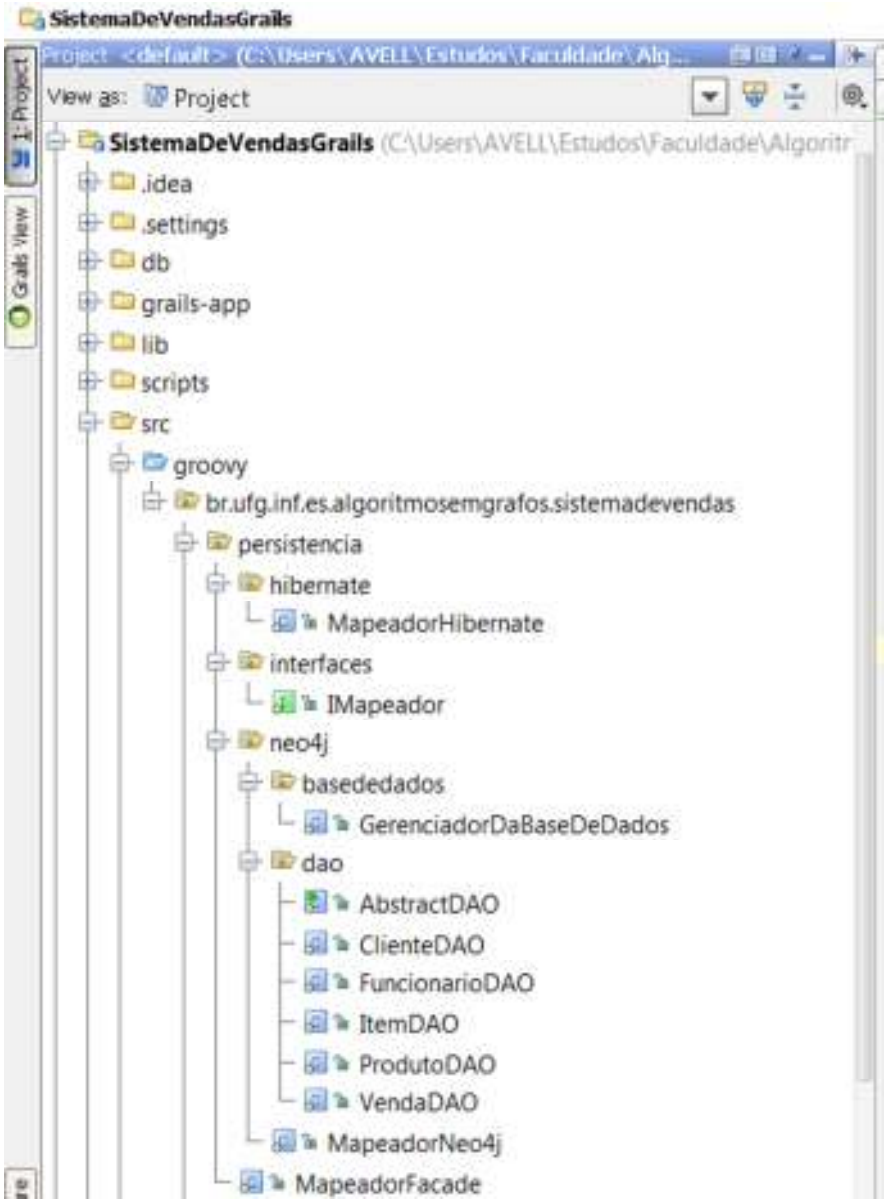
Exemplo de Implementação de um Sistema Simples Usando Um Banco de Dados Orientado a Grafos – Neo4J

Para demonstrar a implementação de um banco de dados orientado à grafos, criamos um sistema simples de vendas, somente algumas classes de domínio. Para implementar utilizamos as tecnologias Groovy e Grails, para tirar maior proveito dos frameworks que já vem pré-configurados, poupando tempo com o mapeamento tradicional, e focando no mapeamento para o Neo4j.

Segue uma imagem das classes de domínio do projeto:



Segue abaixo uma imagem da estruturação da camada de persistência:



Abaixo seguem os códigos das classes de domínio:

Cliente.groovy

```
package br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio

class Cliente {
  Date dataDeCadastro
  Date dataDeNascimento
  String nome = ''
  String endereco = ''
  BigDecimal limite = 0.0

    static constraints = {
      dataDeNascimento(nullable: true)
      dataDeCadastro(nullable: true)
    }
}
```

Funcionario.groovy

```
package br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio

class Funcionario {

  String nome = ''
  Date dataDeNascimento
  String endereco = ''
  BigDecimal salario = 0.0
  BigDecimal bonus = 0.0

  static constraints = {
    dataDeNascimento(nullable: true)
  }
}
```

Item.groovy

```
package br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio

class Item {

  int quantidade = 0;
  Produto produto
}
```

Produto.groovy

```
package br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio

class Produto {

  String codigo = ''
  String nome = ''
  String descricao = ''
  BigDecimal preco = 0.0

  static constraints = {
  }
}
```

Venda.groovy

```
package br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio

class Venda {
  Date data

  Cliente cliente
  Funcionario vendedor
  Set<Item> itens

  static mapping ={
    cliente cascade:'all'
    vendedor cascade:'all'
    itens cascade:'all'
  }
}
```

São basicamente POJOs (Plain Old Java Object), ou POGOs (Plain Old Groovy Object), já que estamos utilizando groovy.

Abaixo segue o código das classes de persistência, para manter o acoplamento baixo, utilizamos o padrão Facade, para que seja possível definir o mapeador até mesmo em tempo real.

IMapeador.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.interfac
es;

public interface IMapeador {

  boolean insira(def objetoPersistente)

  boolean atualize(def objetoPersistente)

  def remova(def objetoPersistente)

  def busque(def objetoPersistente)

  def busque(Long id)

  List liste(def objetoPersistente)
}
```

MapeadorFacade.groovy

```
package br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.interfac
es.IMapeador
import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.utils.ReflectionUtils
import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.utils.StringUtils

class MapeadorFacade implements IMapeador {

  private static MapeadorFacade instancia
  private final String CAMINHO_PADRAO_MAPEADOR =
"br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia"
  private IMapeador mapeador

  static MapeadorFacade obtenhaInstancia() {
    if (instancia == null) {
      instancia = new MapeadorFacade()
    }

    return instancia
  }

  void definaMapeador(String classe) {
    definaMapeador(CAMINHO_PADRAO_MAPEADOR, classe)
  }

  void definaMapeador(String pacote, String classe) {
    String nomeClasse = pacote + "." + classe.toLowerCase() + "." +
"Mapeador" + StringUtils.capitalize(classe)
    mapeador = ReflectionUtils.obtenhaInstancia(nomeClasse)
  }

  boolean insira(def objetoPersistente) {
    return mapeador.insira(objetoPersistente)
  }

  boolean atualize(def objetoPersistente) {
    return mapeador.atualize(objetoPersistente)
  }

  def remova(def objetoPersistente) {
    return mapeador.remova(objetoPersistente)
  }

  def busque(def objetoPersistente) {
    return mapeador.busque(objetoPersistente)
  }

  List liste(def objetoPersistente) {
    return mapeador.liste(objetoPersistente)
  }

  def busque(Long id) {
    return mapeador.busque(id)
  }
}
```

MapeadorHibernate.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.hibernat
e;

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.interfac
es.IMapeador
import org.codehaus.groovy.grails.commons.ApplicationHolder
import org.hibernate.SessionFactory

public class MapeadorHibernate implements IMapeador {
  SessionFactory sessionFactory =
ApplicationHolder.application.mainContext.sessionFactory

  boolean insira(def objeto) {
    if (objeto.save()) {
      return true
    } else {
      println objeto.errors
    }

    return false
  }

  boolean atualize(def objeto) {
    if (!objeto.class.get(objeto.id)) {
      println "Não existe objeto com o id ${objeto.id}"
      return false
    }

    println "Atualizando o objeto com o id ${objeto.id}"
    return insira(objeto)
  }

  def remova(def objeto) {

    if (!objeto.class.get(objeto.id)) {
      return null
    }

    try {
      objeto.delete()
    } catch (Exception ex) {
      return null
    }

    return objeto
  }

  def busque(def objeto) {
    objeto.class.findById(objeto.id)
  }

  def busque(Long id) {
    objeto.class.findById(objeto.id)
  }

  List liste(Object objeto) {
```

```
    objeto.class.findAll()
  }
}
```

Tudo apresentado até agora, é somente a parte do hibernate, e bancos de dados relacionais, a parte nova começa agora.

Para utilizar o banco de dados do neo4j, é necessário inicializar um servidor neo4j, ou utilizar uma versão embarcada, adicionando os jars que podem ser baixados diretamente do site do neo4j. O link se encontra nas referências.

Para inicializar o banco de dados embarcado, utilizamos a classe EmbeddedGraphDataBase, passando como parâmetro o caminho para criar o banco de dados.

Nós optamos por encapsular essa funcionalidade, e algumas outras em uma classe que será utilizada para trabalhar com os objetos.

GerenciadorDaBaseDeDados.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.ba
sededados

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.utils.StringUtils
import grails.util.Environment
import org.neo4j.graphdb.GraphDatabaseService
import org.neo4j.graphdb.Node
import org.neo4j.graphdb.Transaction
import org.neo4j.kernel.EmbeddedGraphDatabase
import org.neo4j.kernel.impl.transaction.TxModule
import org.neo4j.kernel.impl.transaction.XaDataSourceManager
import org.neo4j.kernel.impl.transaction.xaframework.XaDataSource

class GerenciadorDaBaseDeDados {
  String caminhoBaseDeDados =
"db/neo4j/sistemaDeVendas${StringUtils.capitalize(Environment.current.
name)}"
  private GraphDatabaseService baseDeDados
  private static GerenciadorDaBaseDeDados instancia
  private static Transaction transaction

  static GerenciadorDaBaseDeDados getInstancia() {
    return obtenhaInstancia()
  }

  static GerenciadorDaBaseDeDados obtenhaInstancia() {
    if (!instancia) instancia = new GerenciadorDaBaseDeDados()

    return instancia
  }

  public void inicializeBaseDeDados() {
    baseDeDados = new EmbeddedGraphDatabase(caminhoBaseDeDados);

 get the XaDataSource for the native store
    TxModule txModule = baseDeDados.config.txModule;
    XaDataSourceManager xaDsMgr = txModule.xaDataSourceManager;
    XaDataSource xaDs = xaDsMgr.getXaDataSource("nioneodb");

 turn off log rotation
    xaDs.setAutoRotate(false);

 or to increase log target size to 100MB (default 10MB)
    xaDs.setLogicalLogTargetSize(1000 * 1024 * 1024L);
  }

  public void finalizeBaseDeDados() {
    baseDeDados.shutdown()
  }

  public Transaction obtenhaTransacao() {
    if (!transaction) {
      transaction = baseDeDados.beginTx()
    }

    transaction
  }
```

```
  public Node criarNo() {
    Transaction tx = obtenhaTransacao()
    Node no
    try {
      no = baseDeDados.createNode()
      tx.success()
    } finally {
      tx.success()
    }

    return no
  }

  public Node buscaNoPorId(Long id) {
    return baseDeDados.getNodeById(id)
  }

  public List<Node> obtenhaNosDoTipo(String tipo) {
    baseDeDados.allNodes.findAll {it.hasProperty("tipo")}.findAll
{it.getProperty("tipo") == tipo}
  }

  public int count(def objeto) {
    def nosDoTipo = obtenhaNosDoTipo(objeto.class.simpleName)
    return nosDoTipo.size()
  }
}
```

Para estruturar e mapear as classes de domínio para o neo4j, utilizamos o padrão DAO (Data Access Object), e deixamos de uma forma bem simples somente para efeitos de aprendizado.

AbstractDAO.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.da
o

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.ba
sededados.GerenciadorDaBaseDeDados
import org.neo4j.graphdb.Node
import org.neo4j.graphdb.Transaction

abstract class AbstractDAO {

  protected GerenciadorDaBaseDeDados baseDeDados =
GerenciadorDaBaseDeDados.instancia

  boolean insira(Object objeto) {
    Date antes = new Date()

    Transaction tx = baseDeDados.obtenhaTransacao()
    try {
      Node node = (objeto.id) ? baseDeDados.buscaNoPorId(objeto.id) :
baseDeDados.criarNo()

      objeto.id = node.nodeId

      atualize(objeto)

      tx.success()
      return true
    } catch (Exception ex) {
      println "message: " + ex.message
      println "cause: " + ex.cause
      return false
    } finally {
      tx.finish()
    }
  }

  Object remova(Object objeto) {
    Transaction tx = baseDeDados.obtenhaTransacao()
    try {
      Node node = baseDeDados.buscaNoPorId(objeto.id)
      if (node.hasRelationship()) {
        node.getRelationships().each {
          it.delete()
        }
      }

      node.delete()
      tx.success()
      return objeto
    } catch (Exception ex) {
      println "message: " + ex.message
      println "cause: " + ex.cause
      ex.printStackTrace()
      return null
    } finally {
      tx.finish()
    }
```

```
  }

  Object busque(Long id) {
    monteObjetoComNo(baseDeDados.buscaNoPorId(id))
  }

  Object busque(Object objetoPersistente) {
    List nos = liste(objetoPersistente)
    Object objeto
    List paramsDeBusca = obtenhaParamsDeBusca(objetoPersistente)
    objeto = nos.find {
      List paramsDoNo = obtenhaParamsDeBusca(it)
      paramsDoNo.containsAll(paramsDeBusca)
    }
    return objeto
  }

  List liste(Object objetoPersistente) {
    List objetos = []

    for (Node node:
baseDeDados.obtenhaNosDoTipo(objetoPersistente.class.simpleName)) {
      def objeto = monteObjetoComNo(node)
      objetos += objeto
    }

    return objetos
  }

  boolean atualize(Object objeto) {
    Transaction tx = baseDeDados.obtenhaTransacao()
    try {
      Node node = baseDeDados.buscaNoPorId(objeto.id)
      node.relationships.each {
        it.delete()
      }

      monteNoComObjeto(objeto, node)
      tx.success()
      return true
    } catch (Exception ex) {
      println "message: " + ex.message
      println "cause: " + ex.cause
      return false
    } finally {
      tx.finish()
    }
  }

  protected abstract Object monteObjetoComNo(Node node)

  protected abstract void monteNoComObjeto(Object objeto, Node node)

  protected abstract List obtenhaParamsDeBusca(Object
objetoPersistente)
}
```

ClienteDAO.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.da
o

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Cliente
import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.utils.DataUtils
import org.neo4j.graphdb.Node

class ClienteDAO extends AbstractDAO {

  private static instancia

  static ClienteDAO getInstancia() {
    return obtenhaInstancia()
  }

  static ClienteDAO obtenhaInstancia() {
    if (!instancia) instancia = new ClienteDAO()

    return instancia
  }

  protected void monteNoComObjeto(Object cliente, Node node) {
    node.setProperty("tipo", cliente.class.simpleName)
    node.setProperty("nome", cliente.nome)
    node.setProperty("endereco", cliente.endereco)
    node.setProperty("limite", cliente.limite?.toDouble())
    node.setProperty("dataDeNascimento",
DataUtils.convertaDate(cliente.dataDeNascimento))
    node.setProperty("dataDeCadastro",
DataUtils.convertaDate(cliente.dataDeCadastro))
  }

  protected Cliente monteObjetoComNo(Node node) {
    Cliente cliente = new Cliente()
    cliente.setNome node.getProperty("nome")
    cliente.setEndereco node.getProperty("endereco")
    cliente.setLimite node.getProperty("limite")
    cliente.setId node.nodeId
    cliente.dataDeNascimento =
DataUtils.obtenhaDate(node.getProperty("dataDeNascimento"))
    cliente.dataDeCadastro =
DataUtils.obtenhaDate(node.getProperty("dataDeCadastro"))
    return cliente
  }

  protected List obtenhaParamsDeBusca(Object objetoPersistente) {
    String textoDeBusca = ''

    String separador = ' && '

    textoDeBusca += (objetoPersistente.id) ? 'id:' +
objetoPersistente.id : ''
    textoDeBusca += (objetoPersistente.nome) ? separador +
objetoPersistente.nome : ''
    textoDeBusca += (objetoPersistente.limite) ? separador + new
BigDecimal(objetoPersistente.limite) : ''
```

```
    textoDeBusca += (objetoPersistente.endereco) ? separador +
objetoPersistente.endereco : ''
    textoDeBusca += (objetoPersistente.dataDeNascimento) ? separador +
DataUtils.convertaDate(objetoPersistente.dataDeNascimento) : ''
    textoDeBusca += (objetoPersistente.dataDeCadastro) ? separador +
DataUtils.convertaDate(objetoPersistente.dataDeCadastro) : ''

    if (textoDeBusca.startsWith(separador)) {
      textoDeBusca = textoDeBusca.replaceFirst(separador, '')
    }

    return textoDeBusca.split(separador)
  }
}
```

FuncionarioDAO.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.da
o

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Funcionario
import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.utils.DataUtils
import org.neo4j.graphdb.Node

class FuncionarioDAO extends AbstractDAO {

  private static instancia

  static FuncionarioDAO getInstancia() {
    return obtenhaInstancia()
  }

  static FuncionarioDAO obtenhaInstancia() {
    if (!instancia) instancia = new FuncionarioDAO()

    return instancia
  }

  protected void monteNoComObjeto(Object objeto, Node node) {
    node.setProperty("tipo", objeto.class.simpleName)
    node.setProperty("nome", objeto.nome)
    node.setProperty("endereco", objeto.endereco)
    node.setProperty("salario", objeto.salario?.toDouble())
    node.setProperty("bonus", objeto.bonus?.toDouble() )
    node.setProperty("dataDeNascimento",
DataUtils.convertaDate(objeto.dataDeNascimento))
  }

  protected Object monteObjetoComNo(Node node) {
    Funcionario funcionario = new Funcionario()
    funcionario.id = node.nodeId
    funcionario.nome = node.getProperty("nome")
    funcionario.endereco = node.getProperty("endereco")
    funcionario.salario = node.getProperty("salario")
    funcionario.bonus = node.getProperty("bonus")
    funcionario.dataDeNascimento =
DataUtils.obtenhaDate(node.getProperty("dataDeNascimento"))

    return funcionario
  }

  protected List obtenhaParamsDeBusca(Object objetoPersistente) {
    String textoDeBusca = ''

    String separador = ' && '

    textoDeBusca += (objetoPersistente.id) ? 'id:' +
objetoPersistente.id : ''
    textoDeBusca += (objetoPersistente.nome) ? separador +
objetoPersistente.nome : ''
    textoDeBusca += (objetoPersistente.endereco) ? separador +
objetoPersistente.endereco : ''
```

```
    textoDeBusca += (objetoPersistente.salario) ? separador + new
BigDecimal(objetoPersistente.salario) : ''
    textoDeBusca += (objetoPersistente.bonus) ? separador + new
BigDecimal(objetoPersistente.bonus) : ''
    textoDeBusca += (objetoPersistente.dataDeNascimento) ? separador +
DataUtils.convertaDate(objetoPersistente.dataDeNascimento) : ''

    if (textoDeBusca.startsWith(separador)) {
      textoDeBusca = textoDeBusca.replaceFirst(separador, '')
    }

    return textoDeBusca.split(separador)
  }
}
```

ItemDAO.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.da
o

import br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Item
import org.neo4j.graphdb.DynamicRelationshipType
import org.neo4j.graphdb.Node
import org.neo4j.graphdb.Relationship
import org.neo4j.graphdb.RelationshipType

class ItemDAO extends AbstractDAO {

  private static instancia
  RelationshipType contemProduto =
DynamicRelationshipType.withName('contemProduto')

  static ItemDAO getInstancia() {
    return obtenhaInstancia()
  }

  static ItemDAO obtenhaInstancia() {
    if (!instancia) instancia = new ItemDAO()

    return instancia
  }

  protected void monteNoComObjeto(Object item, Node node) {
    node.setProperty("tipo", item.class.simpleName)
    node.setProperty("quantidade", item.quantidade)
    ProdutoDAO.instancia.insira(item.produto)

node.createRelationshipTo(baseDeDados.buscaNoPorId(item.produto.id),
contemProduto)
  }

  protected Item monteObjetoComNo(Node node) {
    Item item = new Item()
    item.setId node.nodeId
    item.setQuantidade node.getProperty("quantidade")

    node.getRelationships(contemProduto).each { Relationship
relationShip ->

item.setProduto(ProdutoDAO.instancia.monteObjetoComNo(relationShip.get
EndNode()))
    }

    return item
  }

  protected List obtenhaParamsDeBusca(Object objetoPersistente) {
    String textoDeBusca = ''
    String separador = ' && '

    textoDeBusca += (objetoPersistente.id) ? 'id:' +
objetoPersistente.id : ''
    textoDeBusca += (objetoPersistente.quantidade) ? separador +
objetoPersistente.quantidade : ''
```

```
    List argumentosDeBusca = textoDeBusca.split(separador)
    if (objetoPersistente.produto) {
      argumentosDeBusca.addAll
ProdutoDAO.instancia.obtenhaParamsDeBusca(objetoPersistente.produto)
    }

    argumentosDeBusca.remove('')
    return argumentosDeBusca
  }
}
```

ProdutoDAO.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.da
o

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Produto
import org.neo4j.graphdb.Node

class ProdutoDAO extends AbstractDAO {

  private static instancia

  static ProdutoDAO getInstancia() {
    return obtenhaInstancia()
  }

  static ProdutoDAO obtenhaInstancia() {
    if (!instancia) instancia = new ProdutoDAO()

    return instancia
  }

  protected Object monteObjetoComNo(Node node) {
    Produto produto = new Produto()
    produto.id = node.nodeId
    produto.nome = node.getProperty("nome")
    produto.codigo = node.getProperty("codigo")
    produto.descricao = node.getProperty("descricao")
    produto.preco = node.getProperty("preco")
    return produto
  }

  protected List obtenhaParamsDeBusca(Object objetoPersistente) {

    String textoDeBusca = ''

    String separador = ' && '

    textoDeBusca += (objetoPersistente.id) ? 'id:' +
objetoPersistente.id : ''
    textoDeBusca += (objetoPersistente.nome) ? separador +
objetoPersistente.nome : ''
    textoDeBusca += (objetoPersistente.codigo) ? separador +
objetoPersistente.codigo : ''
    textoDeBusca += (objetoPersistente.descricao) ? separador +
objetoPersistente.descricao : ''
    textoDeBusca += (objetoPersistente.preco) ? separador + new
BigDecimal(objetoPersistente.preco) : ''

    if (textoDeBusca.startsWith(separador)) {
      textoDeBusca = textoDeBusca.replaceFirst(separador, '')
    }

    return textoDeBusca.split(separador)
  }

  protected void monteNoComObjeto(Object objeto, Node node) {
    node.setProperty("tipo", objeto.class.simpleName)
    node.setProperty("nome", objeto.nome)
```

```
    node.setProperty("codigo", objeto.codigo)
    node.setProperty("descricao", objeto.descricao)
    node.setProperty("preco", objeto.preco?.toDouble())
  }
}
```

VendaDAO.groovy

```
package
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.persistencia.neo4j.da
o

import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Cliente
import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Funcionario
import br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Item
import br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.dominio.Venda
import
br.ufg.inf.es.algoritmosemgrafos.sistemadevendas.utils.DataUtils
import org.neo4j.graphdb.DynamicRelationshipType
import org.neo4j.graphdb.Node
import org.neo4j.graphdb.RelationshipType

class VendaDAO extends AbstractDAO {
  RelationshipType vendidoPara =
DynamicRelationshipType.withName("vendidoPara")
  RelationshipType vendidoPor =
DynamicRelationshipType.withName("vendidoPor")
  RelationshipType vendeu = DynamicRelationshipType.withName("vendeu")
  RelationshipType comprou =
DynamicRelationshipType.withName("comprou")
  RelationshipType contemItem =
DynamicRelationshipType.withName("contemItem")

  private static instancia

  static VendaDAO getInstancia() {
    return obtenhaInstancia()
  }

  static VendaDAO obtenhaInstancia() {
    if (!instancia) instancia = new VendaDAO()

    return instancia
  }

  protected void monteNoComObjeto(Object venda, Node node) {
    Cliente cliente = (ClienteDAO.instancia.busque(venda.cliente)) ?:
venda.cliente
    ClienteDAO.instancia.insira(cliente)

    node.setProperty 'tipo', venda.class.simpleName
    Node noCliente = baseDeDados.buscaNoPorId(cliente.id)
    node.createRelationshipTo(noCliente,
vendidoPara).setProperty('data', ((venda.data) ?
DataUtils.convertaDate(venda.data) : ''))

    noCliente.createRelationshipTo(node, comprou).setProperty('data',
((venda.data) ? DataUtils.convertaDate(venda.data) : ''))

    Funcionario funcionario =
(FuncionarioDAO.instancia.busque(venda.vendedor)) ?: venda.vendedor
    FuncionarioDAO.instancia.insira(funcionario)

    Node noFuncionario = baseDeDados.buscaNoPorId(funcionario.id)
```

```
    node.createRelationshipTo(noFuncionario,
vendidoPor).setProperty('data', ((venda.data) ?
DataUtils.convertaDate(venda.data) : ''))

    noFuncionario.createRelationshipTo(node,
vendeu).setProperty('data', ((venda.data) ?
DataUtils.convertaDate(venda.data) : ''))

    venda.itens?.each { Item it ->
      Item item = (ItemDAO.instancia.busque(it)) ?: it
      ItemDAO.instancia.insira(item)

      Node noItem = baseDeDados.buscaNoPorId(item.id)
      node.createRelationshipTo(noItem, contemItem)
    }
  }

  protected Object monteObjetoComNo(Node node) {
    Venda venda = new Venda()
    venda.id = node.nodeId

    node.getRelationships(vendidoPara).each {
      venda.cliente =
ClienteDAO.instancia.monteObjetoComNo(baseDeDados.buscaNoPorId(it.endN
ode.id))
    }

    node.getRelationships(vendidoPor).each {
      venda.vendedor =
FuncionarioDAO.instancia.monteObjetoComNo(baseDeDados.buscaNoPorId(it.
endNode.id))
    }

    venda.itens = []
    node.getRelationships(contemItem).each {
      venda.itens +=
ItemDAO.instancia.monteObjetoComNo(baseDeDados.buscaNoPorId(it.endNode
.id))
    }

    return venda
  }

  protected List obtenhaParamsDeBusca(Object objetoPersistente) {
    String textoDeBusca = ''
    String separador = ' && '

    textoDeBusca += (objetoPersistente.id) ? 'id:' +
objetoPersistente.id : ''
    textoDeBusca += objetoPersistente.data ? separador +
DataUtils.convertaDate(objetoPersistente.data) : ''
    List argumentosDeBusca = textoDeBusca.split(separador)
    if (objetoPersistente.cliente) argumentosDeBusca.addAll
ClienteDAO.instancia.obtenhaParamsDeBusca(objetoPersistente.cliente)
    if (objetoPersistente.vendedor) argumentosDeBusca.addAll
FuncionarioDAO.instancia.obtenhaParamsDeBusca(objetoPersistente.vended
or)
    objetoPersistente.itens?.each {
      argumentosDeBusca.addAll
ItemDAO.instancia.obtenhaParamsDeBusca(it)
    }
```

```
    argumentosDeBusca.remove('')
    return argumentosDeBusca
  }
}
```