4
Testes e experimentos realizados

4.1.
Implementação e banco de dados

Devido à própria natureza dos *sites* de redes sociais, é normal que a maior parte deles possua uma grande quantidade de usuários e de informações compartilhadas entre eles. O enorme volume de dados faz com que sejam necessárias algumas medidas a fim de minimizar o esforço computacional requerido para gerar recomendações. Em alguns casos, é proibitivo gerar recomendações no momento de exibi-las para o usuário. Quando existe essa possibilidade, é importante que o algoritmo seja eficiente e que os dados estejam bem estruturados.

Em relação à melhor estruturação dos dados, é válido considerar alternativas que têm surgido, diferentes das baseadas em bancos de dados relacionais. Considerando que em uma rede social os usuários e as relações entre eles podem ser facilmente modeladas em um grafo, é intuitivo pensar em um banco de dados em grafo como uma solução mais vantajosa.

Por isso, antes de construir a versão final do sistema de recomendação de pessoas para redes sociais com base em conexões entre usuários, foram realizados testes com um banco de dados tradicional (relacional, PostgreSQL) e outro, em grafo (Neo4j). Optamos por empregar nos testes primeiramente os dados do *site* Peladeiro, que será a base para nosso estudo de caso, detalhado na sessão correspondente deste trabalho.

Para a importação dos dados relacionados aos usuários e suas amizades armazenadas nas tabelas do banco de dados relacional original do *site* Peladeiro, foi criado um programa utilizando a linguagem Java. Cada usuário da tabela de usuários é representado por um nó no banco de dados em grafo. E, para cada relação entre dois usuários, da tabela de relações, os dois nós que os representam são ligados por um arco. Ou seja, nós adjacentes no grafo representam usuários

que são amigos entre si. Por exemplo, na Figura 10 os usuários C e D são amigos do usuário A.

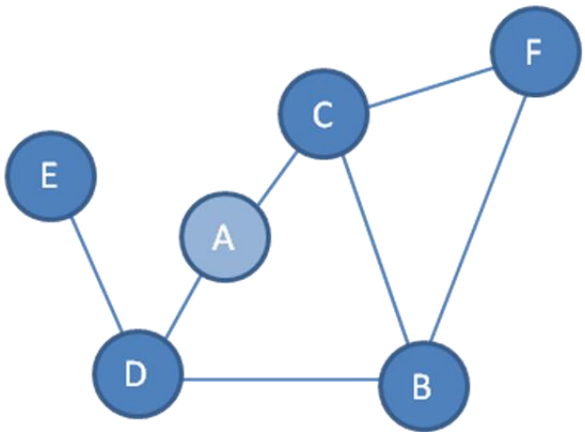

Figura 10 - Amigos em um grafo

O algoritmo de importação primeiramente lista todos os usuários através de uma consulta SQL e cria um nó para cada usuário no grafo. Depois que todos os nós foram criados, cada usuário da lista de usuários retornada pela consulta SQL é visitado no grafo. Para cada um, é feita uma consulta SQL que lista os seus amigos e baseado nela são criados os arcos para interligar aquele usuário com os nós do grafo que representam os seus amigos.

O ambiente de testes também foi implementado em linguagem Java. Os algoritmos retornam uma lista com recomendações de usuários considerados potenciais amigos para um dado usuário. A lista é formada por objetos do tipo *Recomendação*. Cada objeto do tipo *Recomendação* é composto por um objeto do tipo *Usuário*, que representa o usuário que está sendo recomendado, e uma lista de objetos do tipo Usuário, onde cada item da lista representa um usuário que é amigo em comum entre o usuário para o qual a recomendação é destinada e o usuário recomendado (vide diagrama UML da Figura 11).

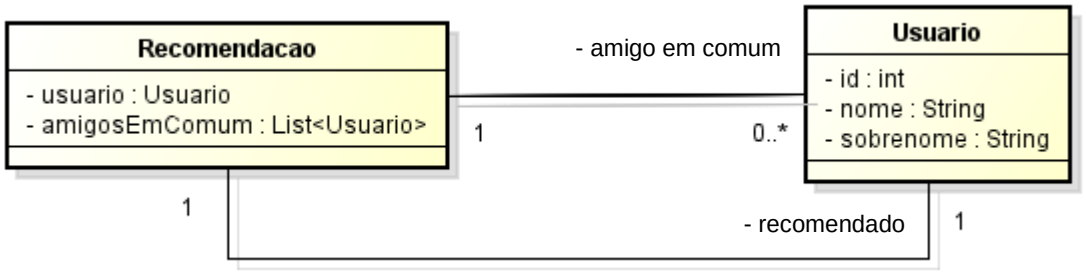

Figura 11 - Diagrama UML das entidades do programa de testes

A lista de recomendações é ordenada decrescentemente pela quantidade de amigos em comum (tamanho da lista de amigos em comum) e alfabeticamente (nome do usuário recomendado), no caso da quantidade de amigos em comum ser igual. A ordenação alfabética foi aplicada apenas para facilitar a comparação entre as saídas das diferentes implementações.

Como descrita anteriormente no texto, a solução será baseada na intuição de que usuários com muitos amigos em comum provavelmente são amigos entre si. A primeira implementação (vide Figura 12), embora não seja eficiente, é bastante simples e de fácil entendimento. Ela utiliza como base uma pequena consulta SQL que lista os amigos de um dado usuário. Essa consulta é executada para obter todos os amigos do usuário para o qual a recomendação será destinada (usuário consumidor). Depois, essa lista de amigos é percorrida e, para cada um, é executada novamente a mesma consulta. Dessa forma são obtidos os amigos dos amigos do usuário consumidor. Cada um é armazenado na lista de recomendações, caso não seja amigo do usuário consumidor e não faça parte da lista. Caso o usuário não seja amigo do usuário consumidor e já esteja na lista, a lista de amigos em comum do objeto Recomendação é atualizada através da adição do usuário que é amigo em comum com o usuário e o usuário consumidor.

```
RECOMENDACOES <- vazio
AMIGOS <- consulta_amigos(USUARIO)
para I <- 1 até tamanho(AMIGOS)
    AMIGOS_DE_AMIGO <- consulta(AMIGOS[I])
    para J <- 1 até tamanho(AMIGOS_DE_AMIGO)
        se AMIGOS_DE_AMIGO[J] não é amigo de USUARIO
            se não está em RECOMENDACOES
                // cria nova recomendação
                RECOMENDACAO.USUARIO <- AMIGOS_DE_AMIGO[J]
                RECOMENDACAO.AMIGOS_EM_COMUM <- AMIGOS(I)
                RECOMENDACOES <- RECOMENDACOES + RECOMENDACAO
            caso contrário
                // atualiza recomendação existente
                RECOMENDACAO.AMIGOS_EM_COMUM <- RECOMENDACAO.AMIGOS_EM_COMUM +
                        AMIGOS(I)
```

Figura 12 - Implementação 1 (PostgreSQL)

A segunda implementação (vide Figura 13) executa apenas uma consulta SQL (no apêndice ao final do trabalho) que retorna diretamente todas as informações necessárias para criar a lista de objetos do tipo Recomendação. Assim como nos outros casos, cada item da lista tem um objeto que representa o

usuário recomendado e a informação sobre a quantidade de amigos em comum com o usuário recomendado e o usuário consumidor.

```
RECOMENDACOES <- consulta_direta(USUARIO)
```

Figura 13 - Implementação 2 (consulta direta em PostgreSQL)

A terceira implementação (vide Figura 14) aplica o mesmo princípio da primeira em um grafo, utilizando o banco de dados Neo4j. No grafo, cada usuário é um nó e cada arco indica uma relação de amizade entre os usuários representados pelos nós ligados. A diferença em relação à primeira está em como os amigos de um dado usuário são obtidos. Enquanto na primeira implementação é utilizada uma consulta SQL, nesta é feita uma busca em largura de um nível no grafo a partir do nó que representa o usuário consumidor para assim obter os nós adjacentes a ele (seus amigos).

```
RECOMENDACOES <- vazio
AMIGOS <- nos_adjacentes(USUARIO)
para I <- 1 até tamanho(AMIGOS)
    AMIGOS_DE_AMIGO <- nos_adjacentes(AMIGOS[I])
    para J <- 1 até tamanho(AMIGOS_DE_AMIGO)
        se AMIGOS_DE_AMIGO[J] não é amigo de USUARIO
            se não está em RECOMENDACOES
                // cria nova recomendação
                RECOMENDACAO.USUARIO <- AMIGOS_DE_AMIGO[J]
                RECOMENDACAO.AMIGOS_EM_COMUM <- AMIGOS(I)
                RECOMENDACOES <- RECOMENDACOES + RECOMENDACAO
            caso contrário
                // atualiza recomendação existente
                RECOMENDACAO.AMIGOS_EM_COMUM <- RECOMENDACAO.AMIGOS_EM_COMUM +
                        AMIGOS(I)
```

Figura 14 - Implementação 3 (repetição da primeira, em Neo4j)

A quarta implementação (vide Figura 15) objetiva explorar as facilidades do Neo4j. Ela lista diretamente todos os usuários cuja distância para o usuário consumidor das recomendações no grafo é 2 usando um método disponibilizado em sua API (vide código no apêndice ao final do trabalho). Porém, essa lista de usuários não é suficiente para montar a lista de recomendações, pois é necessário também saber quais são os usuários que são amigos em comum com os usuários recomendados e o usuário consumidor. Ou seja, o algoritmo não satisfaz as características almejadas para o sistema de recomendação e apenas está sendo considerado para testar a utilidade no Neo4j em um contexto semelhante.

```
RECOMENDACOES <- vazio
AMIGOS_DE_AMIGO <- nos_distancia_2(USUARIO)
para I <- 1 até tamanho(AMIGOS_DE_AMIGO)
    RECOMENDACAO.USUARIO <- AMIGOS_DE_AMIGO[I]
    RECOMENDACOES <- RECOMENDACOES + RECOMENDACAO
```

Figura 15 - Implementação 4 (utilizando a API do Neo4j)

As implementações foram aplicadas primeiramente em uma base de dados real do *site* Peladeiro extraída em fevereiro de 2012, composta por 520982 usuários dos quais 327459 ativos e 6718615 relações de amizade das quais 6698444 concretizadas. Embora tenham sido considerados apenas os usuários ativos e as relações de amizades já concretizadas, os registros não aproveitados não foram excluídos de suas respectivas tabelas no banco de dados.

Apesar de a base de dados do *site* Peladeiro ser formada por 141 tabelas, apenas aquelas que armazenam os usuários (*ustb_usuario*) e as relações de amizade entre eles (*ustb_amigo*), cujos esquemas estão exibidos na Figura 16, foram utilizadas. E, devido à falta de dados consistentes, problema que já foi abordado anteriormente no presente trabalho, apenas as colunas relacionadas com identificadores, nomes e estado (usuário ativo e amizade concretizada) foram aproveitadas.

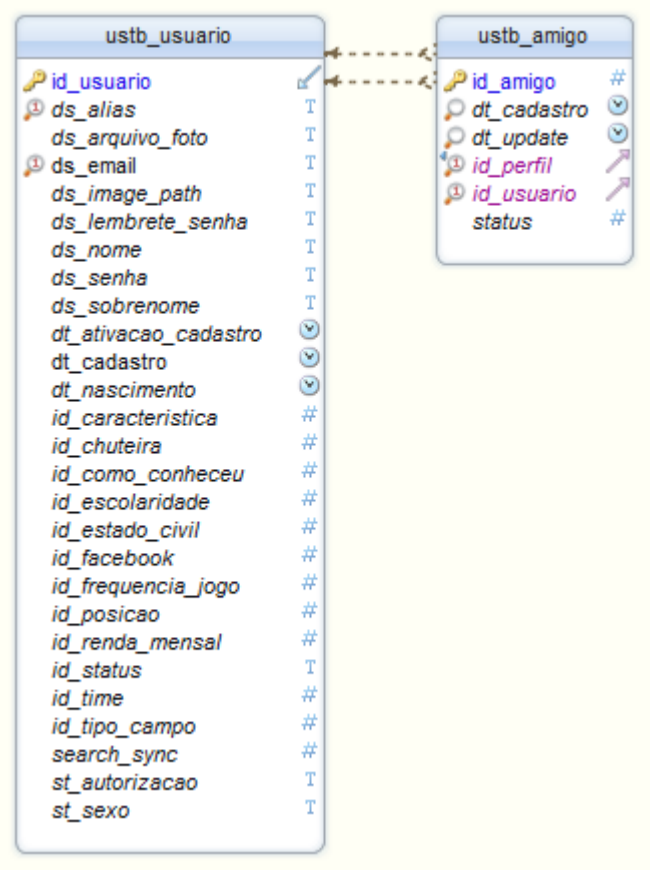

Figura 16 - Esquemas das tabelas do *site* Peladeiro

A execução das diferentes implementações se deu em um computador doméstico com sistema operacional Microsoft Windows 7 (64 bits), 4 GB de memória RAM e processador Intel Core i5 de 2.67 GHz, utilizando banco de dados PostgreSQL 9.1 e Java 1.6. Na Tabela 1 estão listados os tempos de execução, onde cada um foi obtido como resultado da média de 100 execuções. Os tempos são comparados na Figura 17.

Implementação	Banco de dados	Tempo (ms)
1	PostgreSQL	686
2	PostgreSQL	32
3	Neo4j	1496
4	Neo4j	1483

Tabela 1 - Tempos de execução das implementações

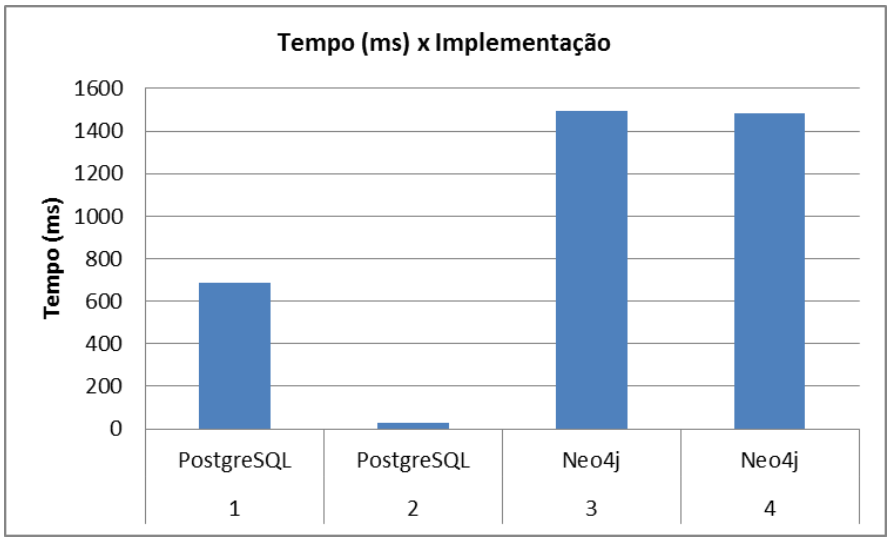

Figura 17 - Tempos de execução das implementações

Uma breve análise nos resultados aponta para uma grande vantagem do banco de dados relacional PostgreSQL frente ao baseado em grafo, Neo4j. Ainda que seja possível fazer otimizações nos algoritmos baseados em grafo ou empregar uma máquina melhor ajustada para seu funcionamento, o uso do Neo4j foi questionado.

Como a maioria dos *sites* utiliza um banco de dados relacional, para adotar uma solução baseada em um banco de dados como o Neo4j seria necessário exportar periodicamente seus dados para o grafo, o que demanda tempo e cria problemas de dados desatualizados, ou manter sempre dados duplicados (em banco de dados relacional e Neo4j ao mesmo tempo). Outra alternativa é modificar o sistema existente no *site* para utilizar o grafo como sua forma de armazenamento padrão, algo ruim visto que demanda trabalho e altera algo que funcionava bem anteriormente. Como as medidas apresentam problemas e os resultados obtido nos experimentos apontam para uma desvantagem do Neo4j, sua utilização foi descartada. É importante salientar que os resultados não descartam uma possível vantagem do Neo4j em outros contextos.

As abordagens NoSQL que tem surgido parecem ser bastante interessantes e já vem sendo empregadas em grandes projetos. Apesar disso, nem sempre uma mudança do modelo tradicional é vantajosa. Além dos problemas de suporte limitado inerente às novas tecnologias e os custos da mudança, principalmente no caso de sistemas legados, não há garantias de ganhos de performance. O ideal é considerar cada caso separadamente e realizar testes antes de escolher a abordagem mais adequada para cada projeto.

4.2. Performance e Escalabilidade

Visto que a segunda implementação é a mais simples de ser implantada, já que é basicamente uma consulta em um banco de dados relacional, e atende as necessidades do sistema, ela foi escolhida para ser utilizada. A fim de testar o seu comportamento com diferentes volumes de dados, foram criados usuários e relações de amizades fictícias e aplicou-se a consulta nessas novas bases de dados fictícias.

A criação das bases de dados fictícias foi feita através de um programa em linguagem Java desenvolvido para esse fim. O banco de dados fictício segue exatamente a estrutura do real utilizado nos testes anteriores, embora só possua as duas tabelas que são necessárias para a geração das recomendações (de usuários e de relações de amizade).

Primeiramente o programa insere um dado número de usuários fictícios no banco de dados fictício. Depois, percorre todos os usuários inseridos e, para cada usuário, são inseridas uma certa quantidade de relações de amizades com outros usuários do banco de dados escolhidos de forma “aleatória” (o programa gera números pseudoaleatórios que correspondem aos identificadores dos usuários que serão os amigos).

Após a inserção das relações de amizade, é executada uma consulta SQL para excluir qualquer registro de amizade duplicado. As duplicidades acontecem, pois o código de inserção não possui qualquer lógica para verificar se aqueles usuários já são amigos. Portanto, o algoritmo pode “aleatoriamente” adicionar uma relação de amizade com um usuário que já é amigo. Os índices da tabela de relações de amizade são criados depois que todas as inserções foram feitas. A ausência de verificação e a criação tardia dos índices objetivam aumentar a velocidade da inserção dos dados.

Na Tabela 2 estão listados os tempos de execução para quantidades de usuários e amigos variadas. Os tempos são comparados na Figura 18. Cada tempo foi obtido como resultado da média de 100 execuções.

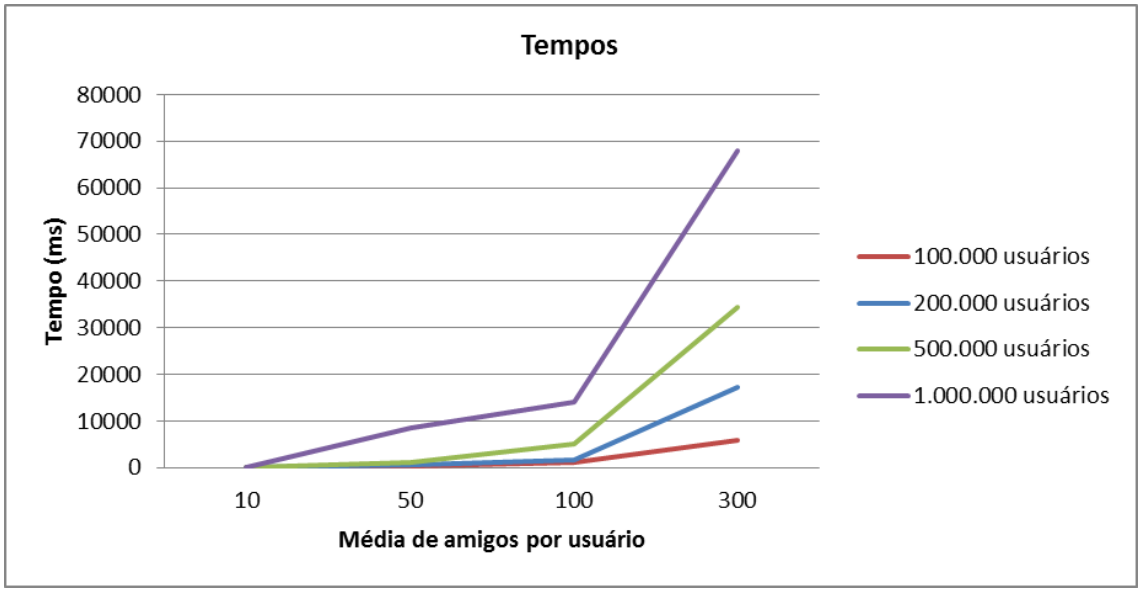

Figura 18 - Tempos de execução para quantidades de usuários/amigos variadas

Quantidade total de usuários	Média de amigos por usuário	Tempo (ms)
100.000	10	32
	50	436
	100	1230
	300	5871
200.000	10	33
	50	607
	100	1589
	300	17262
500.000	10	66
	50	1141
	100	4974
	300	34343
1.000.000	10	59
	50	8407
	100	13939
	300	67942

Tabela 2 - Tempos de execução para quantidades de usuários/amigos variadas

Vale ressaltar que os tempos obtidos com o banco de dados criado com dados fictícios não reflete exatamente o comportamento esperado em um banco de dados real e apenas está sendo analisado para estimar de forma bastante simplificada o comportamento do sistema em situações similares.