Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Gabriel Takeshi Andrade Higa

Bancos de dados orientados a grafos

São Paulo
Dezembro de 2016

Bancos de dados orientados a grafos

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. João Eduardo Ferreira

São Paulo
Dezembro de 2016

Resumo

Neste trabalho, oferecemos uma comparação entre bancos de dados orientados a grafos e bancos de dados relacionais. As duas ferramentas escolhidas para os testes práticos foram o Neo4j e o PostgreSQL. Começamos explicando a diferença entre os modelos lógicos e como eles afetam o processo de modelagem. Além disso, comparamos a facilidade de expressar nossas consultas nas duas linguages estudadas, SQL e Cypher. Em seguida, apresentamos uma análise detalhada explicando como as consultas são executadas e as diferenças de performance para os dois casos. Para complementar, demonstramos como algoritmos em grafos resolvem problemas de forma eficiente no Neo4j.

Palavras-chave: banco de dados, modelo relacional, grafos, comparação de performance.

Sumário

1 Introdução **1**

2 Coleta dos dados **3**

3 Descrição dos modelos de dados **5**

3.1 Modelo relacional 5

3.2 Modelo orientado a grafos 6

4 Consultas realizadas **7**

4.1 SQL – PostgreSQL 7

4.2 Cypher – Neo4j 8

5 Análise e resultados obtidos **11**

5.1 PostgreSQL 11

5.2 Neo4j 13

5.2.1 Adjacência livre de índices 13

5.2.2 Análise das consultas 14

5.3 Comparação de performance 16

6 Algoritmos em grafos no Neo4j **17**

7 Conclusão **19**

Capítulo 1

Introdução

Os bancos de dados (BDs) tradicionais, baseados no modelo relacional, apesar de proverem uma forma estruturada e robusta de lidar com os dados, não são adequados ou suficientes para suprir as necessidades de todas aplicações modernas. Os problemas nesses casos vêm ou da complexidade e variedade dos serviços oferecidos por essas ferramentas, muitos desnecessários para uma dada aplicação, ou a alta restrição imposta pelo modelo de dados, que torna seu uso muito difícil para grandes quantidades de dados ou para lidar com dados distribuídos (escalabilidade horizontal). Aplicações que modificam, extendem ou relaxam condições do modelo tradicional são chamadas de **NoSQL**, um termo geralmente interpretado como *Not Only SQL* (Não apenas SQL) [1, Capítulo 24].

Empresas como Google ou Facebook, que lidam com uma quantidade muito grande de dados diariamente, desenvolveram respectivamente o BigTable e Cassandra, exemplos de ferramentas NoSQL. Naturalmente, nesse caso as limitações do modelo clássico tornavam sua aplicação inapropriada ou inviável para manter a qualidade de serviço das plataformas dessas empresas, mas as vantagens do NoSQL não se restringem a casos extremos como esses. O uso de uma ferramenta mais apropriada pode trazer vantagens de performance, custos de manutenção reduzidos e uma modelagem mais simples, que devem ser considerados mesmo para aplicações muito menores em escala.

Dentre as grandes categorias de sistemas NoSQL, estão os *bancos de dados orientados a grafos*, o foco deste trabalho. A ferramenta estudada é o **Neo4j**, devido a sua apresentação detalhada no livro *Graph Databases* e importância no mercado atual. O Neo4j é um programa de código aberto escrito na linguagem Java. Dentre os usuários do Neo4j, está a multinacional de lojas de departamento Walmart, que viu na ferramenta uma oportunidade de melhoria na performance e simplicidade das consultas necessárias para um bom sistema de recomendações. [6]

O objetivo deste trabalho é compreender e demonstrar como um banco de dados orientado a grafos, como o Neo4j, difere de um banco de dados relacional, além das vantagens e desvantagens de sua utilização. Para isso, uma comparação detalhada dos conceitos dos dois tipos, junto com uma análise de caso com o Neo4j e uma ferramenta SQL de código aberto, o **PostgreSQL**, será oferecida, com experimentos demonstrando sua performance.

A base de dados utilizada para os testes foi gerada a partir da Wikipedia em português e consiste no grafo composto pelos artigos e as hiperligações para outros artigos contidas neles. A inspiração por trás dessa escolha foi a conjectura popular **Seis graus de separação**, elaborada em cima do mesmo modelo *friend of a friend* descrito no livro *Graph Databases*, adaptada para o contexto da Wikipedia. Um artigo da Wikipedia (em inglês) com a ideia pode ser encontrado em https://en.wikipedia.org/wiki/Wikipedia:Six_degrees_of_Wikipedia. Uma descrição sobre como os dados foram coletados para os testes está contida no capítulo 2.

Como o nome indica, a característica que define um banco de dados orientado a grafos é usar um modelo lógico baseado em grafos, estruturas matemáticas compostas por vértices e arestas (ou arcos), chamados de nós e relacionamentos no contexto do Neo4j. A maior diferença nesse modelo é uma representação explícita no modelo lógico do relacionamento entre os dados, que são implícitos no modelo relacional e somente visíveis através das consultas SQL utilizando custosos `JOINs`. Uma explicação detalhada das diferenças entre o modelo relacional e o modelo de grafos é oferecida no capítulo 3.

Naturalmente, é necessária uma linguagem de consulta diferente do SQL para interagir com um banco de dados orientado a grafo, sendo que essa foi projetada para bancos de dados relacionais e não consegue expressar diretamente os conceitos provenientes de uma representação explícita de relacionamentos, como buscas em grafos. A linguagem de consulta utilizada pelo Neo4j é o **Cypher**, mas outros sistemas usam outras linguagens, como o SPARQL ou o Gremlin. Em Cypher, as consultas são representadas de forma simples, misturando sintaxe familiar de uma linguagem de consulta declarativa com arte ASCII, representando o grafo na forma de desenho em texto. Daremos exemplos de consultas em SQL e em Cypher, demonstrando como a segunda expressa as consultas de forma muito mais simples e natural, no capítulo 4.

Na prática, existem diversas formas de implementar o modelo físico subjacente, utilizadas por diferentes ferramentas. O Neo4j implementa o que é chamado de uma *arquitetura nativa de grafos*, ou seja, o modelo físico subjacente dele é uma implementação direta de grafos, em particular, usando listas de adjacência. Outras formas de implementação incluem usar um banco de dados relacional, mas expor um modelo de grafos como interface, mas existem ainda outras. Os resultados práticos obtidos, junto com uma análise de performance explicando os detalhes técnicos intrínsecos do Neo4j e do PostgreSQL ao executar as consultas de teste, estão contidos no capítulo 5.

Adicionamente, mostramos brevemente como algoritmos em grafos permitem resolver problemas naturais do modelo de grafos de forma única e eficiente. Em particular, utilizamos os recursos da linguagem Cypher do Neo4j para isso no capítulo 6.

Capítulo 2

Coleta dos dados

Os dados da Wikipedia foram coletados de duas formas diferentes. Um deles representa toda a Wikipedia em português e o outro utiliza um rastejador para obter dados da Wikipedia em inglês, que é muito maior. Nos dois processos, começamos com um arquivo CSV (*Comma Separated Values*, um formato de arquivo de texto comum para armazenar uma grande quantidade de dados tabulares) que foi carregado primeiro no banco do PostgreSQL para um tratamento preliminar, facilitando a inserção posterior no Neo4j. Inserir um grande volume de dados no Neo4j enquanto tratamos inconsistências não trouxe bons resultados, então tratá-los usando alguma ferramenta antes se mostrou necessário. O PostgreSQL não tem tantos problemas em aceitar o arquivo CSV e foi possível criar tabelas limpas usando comandos em SQL. As duas bases são bem grandes, com a menor delas, gerada pelo rastejador, possuindo quase um milhão de nós (ou seja, artigos da Wikipedia).

No caso da Wikipedia em português inteira, é possível obter imagens do banco de dados onde ela é hospedada no sítio https://dumps.wikimedia.org/ptwiki/. Cada arquivo compactado disponível nesse depósito corresponde a uma tabela no banco original. As tabelas de interesse são a de páginas, a de hiperligações e a de redirecionamentos. A tabela de hiperligações representa uma referência a um artigo em uma página usando um par composto pelo ID desta e o nome do artigo de destino. Incluídos nesses nomes, estão páginas de redirecionamento, que não são normalmente visualizadas pelo usuário e, portanto, não são muito interessantes de se considerar. Resolvemos esse problema usando a tabela de redirecionamentos para substituir essas páginas pelos artigos para os quais elas redirecionam. Assim, o arquivo CSV final é uma tabela com duas colunas, uma com o nome do artigo de origem e a outra com o nome do artigo de destino, cada linha representando uma hiperligação. O grafo representado, devido a esse tratamento, corresponde a nossa intuição como usuários da Wikipedia. Outra consideração importante foi limitar a base de dados aos artigos no *namespace* 0 (chamado de *main*). Essas são simplesmente as páginas "interessantes" da Wikipedia, desconsiderando páginas de discussão, perfis de usuário, etc. [2]

Para tratar os arquivos compactados, que eram simplesmente grandes códigos com inserções em SQL, utilizamos diversas ferramentas de tratamento de texto disponíveis em um sistema Unix por padrão, como *sed* e *awk*, em conjunto com pequenos códigos em Python com expressões regulares um pouco mais elaboradas. No sítio deste trabalho disponibilizaremos os arquivos em Python, mas não nos preocupamos em automatizar as demais correções, sendo elas ajustes menores reagindo a erros que o PostgreSQL acusava na tentativa de executar um `COPY` com o arquivo CSV de entrada. As três tabelas foram convertidas em CSV e carregadas desse modo. O processamento para gerar o grafo como descrito no parágrafo anterior foi feito usando um script SQL que também será disponibilizado.

Essa base foi utilizada para os testes de algoritmos em grafos do Neo4j, já que era do

nosso interesse validar os resultados das consultas. Dessa forma, o PostgreSQL foi utilizado primariamente como uma ferramenta de tratamento de dados, sem que testes feitos nele usando esses dados estejam incluidos nos nossos resultados. O motivo por trás dessa decisão é que os dados gerados dessa forma não levavam a experimentos significativos na prática, por razões que serão explicadas junto com os resultados obtidos. Foi isso que nos motivou a gerar uma segunda base, mais controlada, mas ainda no mesmo contexto da Wikipedia.

Para obter o mesmo grafo para a Wikipedia em inglês, o rastejador *Scrapy* (https://scrapy.org/) foi utilizado. O código da "aranha", como é chamado o *bot* dessa ferramenta, também estará disponível no sítio deste trabalho. Se o método anterior pode ser descrito como uma estratégia *bottom-up*, pois regeramos a base de dados a partir dos scripts em SQL disponíveis na internet, aqui utilizamos uma estratégia *top-down*, onde o rastejador imita um usuário da Wikipedia, seguindo as hiperligações entre os artigos e registrando sua tragetória.

O rastejador começa em um artigo da Wikipedia que o usuário definir através da linha de comando e analisa o HTML do artigo em busca de *href*s no texto, que serão utilizados para encontrar novas páginas para explorar recursivamente. Controlamos o crescimento da busca de duas formas, uma horizontal, limitando o número de hiperligações que a aranha deve explorar por página, e outra vertical, cortando a busca quando atingirmos determinada profundidade. Ambos os parâmetros são ajustáveis de uma forma ou outra, sem muita dificuldade, alterando o código.

Devemos notar algumas diferenças importantes entre os dois métodos, além do fato óbvio que o segundo não gera uma réplica fiel da Wikipedia da forma que queríamos. Primeiro, o tempo que o segundo método leva para gerar a base de dados é muito maior do que o primeiro, no qual invés de simular uma sessão de um usuário, havíamos utilizado apenas processamento de texto. Dependendo dos limites de largura (limite horizontal) e profundidade (limite vertical) na busca, por volta de um dia inteiro para completar a execução foi necessário. Segundo, não nos preocupamos em lidar com redirecionamentos ou em considerar apenas páginas no *namespace* 0 como descrevemos anteriormente. O objetivo principal dessa base foi obter dados, em quantidades grandes o bastante para levar as ferramentas ao limite e nos fornecer resultados significativos, que também estivessem altamente dentro do nosso controle, então gerar dados totalmente limpos não foi uma grande preocupação. Era possível obter um controle ainda maior gerando uma base totalmente artificial, mas não queríamos fugir do modelo proposto inicialmente, além de desejarmos manter certa uniformidade com a outra base.

Capítulo 3

Descrição dos modelos de dados

A seguir, serão oferecidas descrições básicas dos modelos subjacentes de um banco de dados relacional e de um orientado a grafos. Não serão abordados detalhes de implementação nas ferramentas escolhidas para o experimento deste trabalho, o Postgres e o Neo4j. A descrição do modelo relacional a seguir é baseada na do livro *Fundamentals of Database Systems.*[1]

3.1 Modelo relacional

O banco de dados no modelo relacional é representado por um conjunto de *relações*. Informalmente, podemos pensar numa relação como uma tabela, onde cada linha representa algum fato sobre o mundo. As tabelas e suas colunas possuem nomes que nos ajudam a entender o significado dos dados contidos na linha. Por exemplo, no modelo relacional do grafo das hiperligações entre artigos da Wikipedia que usaremos, podemos imaginar que as hiperligações estão sendo representadas numa tabela onde cada linha contém uma referência para a página onde ela se encontra e a página para a qual ela redireciona. Formalmente, chamamos linhas de *tuplas*, os nomes das colunas de *atributos* e o tipo de dado que descreve os possíveis valores em cada linha de *domínio*.

Um **domínio** D é um conjunto de valores atômicos, ou seja, indivisíveis do ponto de vista do modelo relacional. Para o nosso modelo, por exemplo, o domínio relevante é o do nome dos artigos da Wikipedia, mas ele poderia ser estendido para incluir outras coisas, como a data da última modificação de uma página.

Para descrever uma relação de nome R, é necessário definir uma **relação esquema** composta por R e uma lista de atributos $A_1, \ldots, A_n$, a qual denotamos por $R(A_1, \ldots, A_n)$. Cada **atributo** A_i é um nome dado ao papel que algum domínio D tem na relação. Podemos denotar o domínio de A_i por $dom(A_i)$. O **grau** da relação é o número n de atributos de sua relação esquema. No nosso caso, a relação representando as hiperligações seria descrita da forma `links(page_from, page_to)`.

Finalmente, uma **relação** (também chamada de instância da relação) r da relação esquema $R(A_1, \ldots, A_n)$, denotado por $r(R)$, é um conjunto de tuplas $r = t_1, \ldots, t_m$. Cada tupla é uma lista ordenada de n valores $t =< v_1, \ldots, v_n >$, onde cada valor $v_i, 1 \leq i \leq n$ é um elemento de $dom(A_i)$ ou é um valor especial `NULL`.[1, Capítulo 5][5]

As vantagens do modelo relacional estão diretamente ligadas a sua forte origem matemática e a álgebra relacional, estendida para a linguagem SQL usada nos SGBDs relacionais. Entretanto, a tradução do mundo real para ele não é sempre tão direta e, uma vez estabelecido o modelo, muitas vezes não é tão simples estendê-lo para atender a uma nova necessidade. A existência de modelos conceituais, como o modelo entidade-relacionamento,

e algoritmos de conversão para o modelo relacional é o que permite uma boa modelagem nesse caso.

3.2 Modelo orientado a grafos

Existem muitas formas de definir um grafo formalmente, algumas mais genéricas e outras menos. Vamos focar no modelo utilizado pelo Neo4j, o modelo de grafo de propriedades (em inglês, *Property Graph Model*) definido no livro *Graph Databases*[4] da seguinte forma:

- Um grafo de propriedades é composto por **nós**, **relacionamentos** e **propriedades**.
- Uma propriedade é um par ordenado contendo uma chave e um valor. Comparando com o modelo relacional, uma chave é como um nome indicando o significado daquele valor para o objeto que contém a propriedade.
- Nós podem conter rótulos, indicando um tipo aos quais eles pertencem, e uma lista de propriedades, descrevendo a estrutura que ele representa. No nosso modelo, todos nós tem o rótulo `Page` para indicar que são páginas e a propriedade `(Title, "Título")` representando o nome do artigo contido na página.
- Relacionamentos conectam e estruturam nós. Eles sempre possuem uma direção, um rótulo e referências tanto ao nó inicial quanto ao nó final. Além disso, eles podem conter uma lista de propriedades. No modelo da Wikipedia, existe apenas um tipo de relação, `linksTo`, representando uma hiperligação conectando dois artigos.

A grande diferença para o modelo relacional é que relações são componentes diretos do modelo e não são representados implicitamente. Isso se traduz numa maior facilidade para modelar o nosso domínio da Wikipedia. Existe, claro, uma troca sendo feita. O modelo de grafos é muito menos genérico que o modelo relacional e é necessário analisar com cuidado se o que planeja-se modelar tem uma representação simples e clara nele. Se esse for o caso, a modelagem é praticamente instantânea e com baixo esforço, uma vantagem sobre o modelo relacional. Isso não é uma surpresa grande, levando em consideração que existem grandes semelhanças entre o modelo de grafos e o modelo entidade-relacionamento.

Capítulo 4

Consultas realizadas

Para comparar a performance de um banco de dados relacional e um banco de dados orientado a grafos, buscamos a resposta para a seguinte pergunta nas consultas: Que artigos da Wikipedia podem ser alcançados seguindo uma cadeia de n hiperligações a partir de algum artigo inicial qualquer fixo?

Isso é um problema clássico de busca resolvido usando grafos, então esperamos que ele evidencie bem as vantagens do modelo de grafos. Para este trabalho, iremos nos referir a quantidade n de hierpligações como a **profundidade** da busca. Vamos descrever a seguir as consultas com maior detalhe e explicar que recursos das linguages SQL e Cypher foram utilizados. Para cada linguagem foram usadas duas consultas, uma evitando expandir o mesmo nó múltiplas vezes e outra não. Nos referimos a essas consultas pelos termos **sem filtro** e **com filtro**, respectivamente. Essa separação se deve a peculiaridades da linguagem que serão explicadas a seguir.

4.1 SQL – PostgreSQL

Quando pensamos em consultas que envolvem atravessar relações no modelo de grafos, é evidente que para o modelo tradicional isso será traduzido na forma de uma ou mais cláusulas de `JOIN`. Isso é uma consequência da forma que relações são representadas no modelo relacional, implicitamente. Sem esse recurso da linguagem de consulta, não é possível se referir a elas. Pensando no problema em questão, não é uma surpresa que precisamos ainda encadear vários desses em uma mesma consulta para alcançar profundidades maiores. No PostgreSQL isso pode ser feito usando CTEs (*Common Table Expressions*), ou seja, cláusulas `WITH`. Nesse caso em específico, precisamos de um `WITH RECURSIVE`.

A consulta executada no PostgreSQL foi a seguinte:

```
WITH RECURSIVE linksr(page_from, page_to) AS (
    SELECT links.page_from, links.page_to, 1 AS level
    FROM links
    JOIN page
    ON page.pageid = links.page_from
    WHERE page.title = START_NODE
    UNION ALL
    SELECT links.page_from, links.page_to, linksr.level + 1
    FROM links
    JOIN linksr
    ON links.page_from = linksr.page_to
    WHERE level < MAX_DEPTH
)
SELECT DISTINCT count(page.title)
```

```
FROM linksr
JOIN page
ON linksr.page_from = page.pageid;
```

A consulta acima não é uma consulta de SQL válida. Para executá-la, é necessário substituir as variáveis `START_NODE` pelo título do artigo original e `MAX_DEPTH` pela profundidade de busca máxima.

Uma CTE é composta por um termo não recursivo, localizado acima do `UNION`, e um termo recursivo abaixo. Vamos entender como ela é computada, sem entrar nos detalhes mais internos:

- Primeiro, o termo não recursivo é executado, com seu resultado sendo guardado na tabela de resultado final e em uma *tabela de trabalho* temporária.
- Enquanto a tabela de trabalho não estiver vazia, o termo recursivo é executado usando o conteúdo da tabela de trabalho no lugar da autorreferência, com seu resultado sendo adicionado à tabela de resultado e guardado numa *tabela intermediária*. No fim, a tabela intermediária sobreescreve a tabela de trabalho e esse passo é repetido.[3]

Para evitar a expansão sobre nós repetidos, basta substituir o `UNION ALL` por um simples `UNION`. A diferença é que para a segunda cláusula, o processo acima filtra a tabela de resultados e de trabalho para resultados repetidos, similar a execução de um `SELECT DISTINCT`. Não existe muito sentido em sequer usar o `UNION ALL` para esse caso, já que ele só trás informações redundantes. O motivo para considerarmos ele é a proximidade com o que o Neo4j faz na forma mais natural da consulta dele, já que normalmente também queremos informações referentes às relações que foram usadas.

Outro ponto importante e único da consulta no caso do PostgreSQL é a denormalização. Para recuperarmos informações de página são necessários dois `JOIN` adicionais: Um no começo para encontrarmos as hiperligações que saem da página inicial e um no fim para encontrarmos as páginas alcançadas depois de seguir as n hiperligações. Se todas as informações pertinentes às páginas estivessem contidas somente na tabela de hiperligações isso não seria necessário e, para os dados do nosso experimento, isso poderia muito bem ser o caso, mas essa solução não escala muito bem.

O maior problema em manter os dados já denormalizados no banco é redundância. Todas as informações relevantes de uma determinada página precisariam estar repetidas em todas as linhas que as referenciam na tabela de hiperligações. Isso dificulta atualizações no banco de dados, pois elas teriam que ser repetidas em vários lugares para manter a consistência dos dados. Suponha que nosso modelo também precisasse guardar datas de modificação de artigo, informações de autores, etc. Muito rapidamente a decisão de manter os dados denormalizados se mostra inviável na prática. Dessa forma, escolher manter os dados normalizados é mais próximo do cenário real de uso e tornará a comparação com o modelo de grafos mais realista e justa.

4.2 Cypher – Neo4j

A forma mais natural de fazer essa consulta em Cypher é muito mais simples do que no SQL:

```
MATCH (a:Page { title : START_NODE })
      -[:linksTo*1..DEPTH_LIMIT]->
      (b:Page)
RETURN COUNT(DISTINCT b);
```

O que fazemos é fixar um nó inicial a com título `START_NODE` e pedimos para o Neo4j encontrar todos nós b que podem ser alcançados seguindo até `DEPTH_LIMIT` hiperligações a partir de a. Novamente, a consulta como está acima não é válida. É necessário substituir as variáveis por valores.

É evidente que a consulta em Cypher é uma forma muito mais clara de traduzir a pergunta inicial para uma linguagem declarativa. A descrição oferecida no parágrafo anterior é quase exatamente igual a pergunta fornecida no começo dessa seção. Entretanto, a consulta acima sofre do mesmo problema da consulta SQL usando `UNION ALL`, que é considerar várias vezes um mesmo nó em uma mesma profundidade. Para obter uma eficiência maior, a variante abaixo foi utilizada:

```
MATCH (a:PtPage { title : START_NODE })
      -[:linksTo]->
      (b:PtPage)
WITH DISTINCT b AS a
//MATCH (a)-[:linksTo*0..1]->(b:PtPage)
//WITH DISTINCT b AS a
RETURN COUNT(DISTINCT b);
```

Para executar uma consulta equivalente, é necessário repetir a parte comentada $n - 1$ vezes na consulta acima, onde n é `DEPTH_LIMIT`. Essa forma é muito menos elegante, mas não foi encontrada uma alternativa em Cypher para expressar a consulta da mesma forma que é possível em SQL com `UNION`. A parte não comentada é equivalente ao termo não recursivo e cada cópia da parte comentada é quase uma execução do termo recursivo da CTE. Ainda existe um problema nessa versão: Os nós dos passos anteriores são expandidos novamente em cada passo. Uma solução para isso não foi encontrada, mas, como veremos nos resultados obtidos, o Neo4j acaba sendo mais rápido que o PostgreSQL de qualquer forma.

Capítulo 5

Análise e resultados obtidos

A seguir, será oferecida uma análise detalhada sobre como cada consulta é executada, com o objetivo de evidenciar o ponto forte de um BD baseado em grafos. Evidentemente, existem outras operações relevantes que um BD precisa executar eficientemente, como grandes inserções, remoções, etc., mas é importante primeiro validar o principal ponto de venda da ferramenta para entender sua potencial relevância.

Primeiro, vamos usar os recursos de cada ferramenta para entender que operações foram escaladas pelo planejador, qual foi o custo de cada uma e como ele cresce para profundidades maiores. Em seguida, mostramos os resultados práticos obtidos. O foco será nas consultas com filtro, já que são as versões mais eficientes de cada ferramenta, levando a uma comparação mais justa, mas o caso sem filtro será referenciado, com enfoque nas diferenças entre os dois. Apesar de diferenciarmos consultas com e sem filtro no PostgreSQL e no Neo4j, é importante ter em mente que o filtro nos dois não é comparável do ponto de vista de execução. As consultas fazem coisas bem diferentes, como já explicamos.

Além disso, introduzimos o importante conceito de **adjacência livre de índices** para o Neo4j. Como veremos, isso é fundamental para a boa performance do Neo4j, servindo como substituto para todos algoritmos executados pelo PostgreSQL em `JOINs` que explicaremos a seguir.

Uma importante consideração, para os resultados práticos obtidos a seguir, é que as consultas foram executadas mais de uma vez antes dos resultados serem contabilizados. Normalmente, a primeira consulta costuma demorar muito mais, pela ausência de dados em cache, além de ter grande variância, se comparada com as consultas subsequentes. A importância de boa performance no acesso à memória secundária para cada BD é reconhecida, mas analisar isso corretamente é difícil e não está no escopo deste trabalho.

Os testes a seguir foram todos executados no mesmo computador, sob condições similares. As ferramentas foram todas ajustadas seguindo as recomendações de performance fornecidas em suas respectivas documentações. No caso do PostgreSQL, que por padrão utiliza uma configuração bastante conservadora, utilizamos a ferramenta *PgTune* disponível nessa URL http://pgtune.leopard.in.ua/.

5.1 PostgreSQL

Ao executar a consulta no PostgreSQL com a expressão `EXPLAIN ANALYZE`, com profundidade 4, obtemos a resposta seguinte (estimativas de custo removidas e texto ligeiramente formatado para melhor visibilidade):

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate    (actual time=7003.029..7003.029 rows=1 loops=1)
```

```
  CTE linksr
    ->  Recursive Union
        (actual time=0.178..3310.794 rows=985898 loops=1)
          ->  Nested Loop
              (actual time=0.172..0.220 rows=79 loops=1)
                ->  Index Scan using page_title_key on page page_1
                    (actual time=0.082..0.083 rows=1 loops=1)
                      Index Cond: ((title)::text = 'START_PAGE'::text)
                ->  Index Only Scan using links_pkey on links
                    (actual time=0.077..0.112 rows=79 loops=1)
                      Index Cond: (page_from = page_1.pageid)
                      Heap Fetches: 79
          ->  Nested Loop
              (actual time=15.970..467.304 rows=689015 loops=4)
                ->  WorkTable Scan on linksr linksr_1
                    (actual time=15.964..19.748 rows=19492 loops=4)
                      Filter: (level < 4)
                      Rows Removed by Filter: 226983
                ->  Index Only Scan using links_pkey on links links_1
                    (actual time=0.004..0.018 rows=35 loops=77966)
                      Index Cond: (page_from = linksr_1.page_to)
                      Heap Fetches: 2756059
  ->  Hash Join
      (actual time=386.676..4498.879 rows=985898 loops=1)
        Hash Cond: (linksr.page_to = page.pageid)
        ->  CTE Scan on linksr
            (actual time=0.182..3597.959 rows=985898 loops=1)
        ->  Hash
            (actual time=381.997..381.997 rows=907578 loops=1)
              Buckets: 1048576  Batches: 1  Memory Usage: 64890kB
              ->  Seq Scan on page
                  (actual time=0.022..144.075 rows=907578 loops=1)
Planning time: 72.066 ms
Execution time: 7059.992 ms
(24 rows)
```

Vamos analisar apenas a parte de dentro do *Aggregate* mais externo, sendo que ele serve somente para agrupar e contar os valores retornados, de forma a evitar a escrita demorada na saída. Algo similar é usado no caso de grafos, então isso não será uma vantagem injusta. De fato, o tempo de escrita é contabilizado em alguns casos, o que atrapalha o estudo.

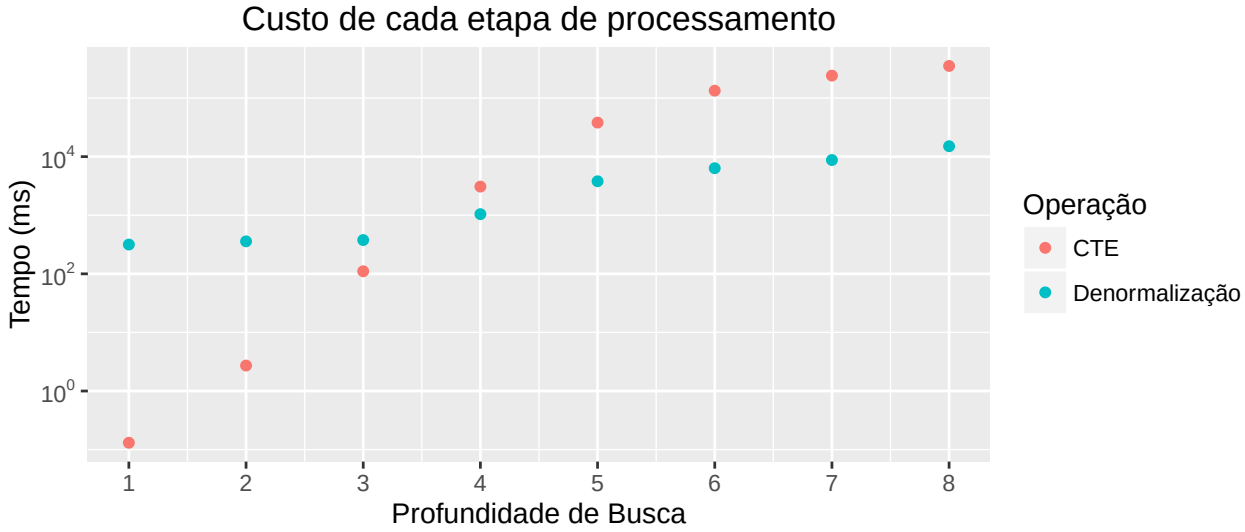

Figura 5.1: *Custo de cada etapa da consulta, evidenciando a contribuição da denormalização e da computação da CTE no custo total.*

A consulta toda é um *Hash Join* entre uma CTE e uma tabela existente. A leitura sobre a CTE contabiliza tanto o custo de leitura em si quanto a sua criação, cujos detalhes e custo individual estão acima dessa etapa. Para estimarmos o custo da denormalização, podemos subtrair o custo da criação da CTE do custo total. O resultado disso pode ser observado na figura 5.1, para o caso com filtro. Podemos supor que a denormalização, apesar de ter um custo elevado, cresce de forma quase linear, enquanto a criação da CTE é exponencial e, a partir de uma certa profundidade, contribui de forma muito mais relevante para o tempo de execução total. No gráfico, a partir da profundidade 6 esse padrão não se mantém, mas isso é causado pelos limites da nossa base de dados serem atingidos nesse ponto. Entretanto,

não podemos ignorar o custo extra para baixas profundidades, que, como veremos, contribui para que o Neo4j tenha uma performance melhor sempre.

Em um *Hash Join*, as linhas de uma tabela são usadas para criar uma tabela hash na memória e, em seguida, lê-se as linhas da outra tabela, inserindo elas no balde correspondente à condição de join. Normalmente, a menor relação é usada para criar a tabela hash, para que se obtenha uma performance melhor. Entretanto, para as consultas analisadas, a tabela de páginas sempre foi a escolhida pelo planejador. Não se tem certeza do porquê dessa escolha, mas provavelmente sua causa são limitações da CTE comparada com tabelas normais.

A parte mais importante dessa consulta, entretanto, é o join recursivo na criação da CTE, sendo que é daí que esperamos um grande ganho de performance em um BD baseado em grafos. Primeiro, observe que todo passo é executado usando um *Nested Loop Join*, uma estratégia simples de join que, para cada linha relevante da tabela do lado esquerdo do join, busca linhas na tabela do lado direito que correspondem a condição de join. Podemos então quebrar a recursão em duas partes:

- **Inicialização.** Primeiro é executado uma busca pelo índice na coluna *title* da tabela de páginas para encontrar as linha com as páginas iniciais. Em seguida, para cada linha encontrada na busca anterior, é executado um join com a tabela de hiperligações.

  Durante a inserção no banco de dados, foram filtradas linhas repetidas, representando arestas paralelas no modelo de grafos. Dessa forma, é impossível existirem resultados repetidos nessa etapa, levando a um custo possivelmente menor na operação de `UNION`.

- **Passo recursivo.** Consiste em um join entre a CTE e a tabela de hiperligações, unindo a página de destino de uma com a de início de outra pelo **id**. A maior diferença entre esse passo e a inicialização é que a tabela esquerda do join terá muito mais entradas, sendo necessário checar a condição de join na tabela direita toda para cada uma delas.

  Dependendo do modelo de dados, é de se esperar que até uma certa profundidade, a quantidade de nós expandidos nessa etapa cresca exponencialmente. Isso é verdade para o modelo da Wikipedia até 6 expansões, que é quando o conjunto de busca corresponde quase à maior parte da base, como vimos anteriormente. Além disso, num caso sem filtro, a expansão sobre nós repetidos inevitavelmente leva a um crescimento muito rápido.

  O `UNION` no caso recursivo precisa eliminar nós repetidos. Para isso, existem duas estratégias comuns possíveis para um banco de dados relacional, entre as quais não foi possível verificar qual o PostgreSQL utiliza. Uma delas consiste em utilizar uma tabela hash, agrupando linhas iguais em um mesmo balde. Na outra, o conjunto resultado é ordenado e, através de uma leitura sequencial, é possível agrupar linhas repetidas. De qualquer forma, o custo dessa operação acaba sendo muito menor do que o custo de expandir nós repetidos desnecessariamente, como veremos.

5.2 Neo4j

5.2.1 Adjacência livre de índices

O principal diferencial de um banco de dados com processamento nativo em grafos é a adjacência livre de índices . Esse conceito se refere a capacidade de, dada uma referência a um nó qualquer, encontrar todos os nós adjacentes sem a necessidade de executar uma busca na base de dados. Embora sua implementação não seja essencial para um BD baseado em grafos,

levando em conta a importância dos relacionamentos entre os dados no modelo, a operação descrita anteriormente pode ser considerada básica e é importante que seja executada de forma eficiente.

Apesar do nome, adjacência livre de índices não se refere a não utilização de índices, estruturas auxiliares que possibilitam a localização dos dados na base através de um meio alternativo, mais rápido [1, capítulo 17]. De fato, não é executada nenhuma busca; cada nó possui, direta ou indiretamente, uma referência aos nós adjacentes.

No caso do *Neo4j*, um nó possui uma referência para uma lista duplamente ligada de relações. Cada relação, por sua vez, possui referências para o nó inicial e o nó final. Dessa forma, o custo para encontrar todos nós adjacentes é diretamente proporcional ao número de relações das quais ele participa. Dependendo do tipo de busca, ainda pode ser necessário filtrar as relações pelos seus atributos ou direção [4, capítulo 6].

Na prática, a maioria dos modelos de dados resultam em grafos esparsos. Isso significa que a afirmação feita no livro *Graph Databases* de que adjacência livre de índices permite localizar nós adjacentes com complexidade $O(1)$ é, muitas vezes, razoável. Entretanto, é necessário considerar que em grafos mais densos ou, mais comumente, na ocorrência de eventuais "super nós" que participam de muitas relações, a performance esperada é muito pior.

Além disso, apesar da implementação mais comum de índices serem árvores B+, que realizam buscas em tempo $O(log_b\ n)$ (onde b é sua ordem), similar ao que o livro afirma, existem outros tipos de índices que podem levar a uma complexidade menor em alguns casos, como *hashes*. Então não é sempre verdade que adjacência livre de índices leva a uma performance maior na prática.

É um fato, entretanto, que adjacência livre de índices é uma técnica natural ao modelo de grafos, onde relações são tratadas como entidades primárias. Além da consideração inicial, de que seus dados são modelados de forma adequada na forma de grafos, através de um estudo dos casos de uso, não é necessário se preocupar muito se as consultas serão realizadas usando o método mais eficiente.

5.2.2 Análise das consultas

Ao executar a consulta no Neo4j com a expressão `PROFILE`, com profundidade 4, obtemos a resposta seguinte (estimativas de custo removidas, junto de algumas colunas de menor importância e texto ligeiramente formatado para melhor visibilidade):

```
lstlistingnner COST

Runtime INTERPRETED

+----------------------+--------+---------+--------------------------------+
| Operator             | Rows   | DB Hits | Other                          |
+----------------------+--------+---------+--------------------------------+
| +ProduceResults      |      1 |       0 | COUNT(DISTINCT b)              |
| |                    +--------+---------+--------------------------------+
| +EagerAggregation    |      1 |       0 |                                |
| |                    +--------+---------+--------------------------------+
| +Filter              | 958080 |  958080 | b:Page                         |
| |                    +--------+---------+--------------------------------+
| +VarLengthExpand(All)| 958080 |  958080 | (a@194)-[:linksTo*]->(b@224)   |
| |                    +--------+---------+--------------------------------+
| +Distinct            |  45638 |       0 | a                              |
| |                    +--------+---------+--------------------------------+
| +Filter              |  78559 |   78559 | b:Page                         |
| |                    +--------+---------+--------------------------------+
| +VarLengthExpand(All)|  78559 |   78559 | (a@137)-[:linksTo*]->(b@167)   |
| |                    +--------+---------+--------------------------------+
| +Distinct            |   2263 |       0 | a                              |
| |                    +--------+---------+--------------------------------+
| +Filter              |   2510 |    2510 | b:Page                         |
| |                    +--------+---------+--------------------------------+
| +VarLengthExpand(All)|   2510 |    2510 | (a@80)-[:linksTo*]->(b@110)    |
| |                    +--------+---------+--------------------------------+
| +Distinct            |     79 |       0 | a                              |
| |                    +--------+---------+--------------------------------+
| +Filter              |     79 |      79 | b:Page                         |
```

```
| |                      +---------+---------+-------------------------------+
| +Expand(All)           |      79 |      80 | (a@7)-[:linksTo]->(b@53)      |
| |                      +---------+---------+-------------------------------+
| +NodeUniqueIndexSeek   |       1 |       2 | :Page(title)                  |
+----------------------+---------+---------+-------------------------------+

Total database accesses: 2078459
```

Novamente, a execução acima é de uma consulta com o filtro que evita expansão sobre nós repetidos em um mesmo passo da busca. Nesse caso, entretanto, não estamos criando uma CTE e sim encontrando diretamente todos os nós nos quais estamos interessados. A diferença entre uma consulta com filtro e sem filtro no caso do Neo4j é bem maior, sendo a primeira na verdade uma composição de consultas em Cypher usando `WITH`. Novamente, podemos quebrar a busca em duas etapas:

- **Inicialização.** Primeiro é executada uma busca pelos nós iniciais usando um *Unique Index*. Existem algumas diferenças entre os vários índices do Neo4j, mas a implementação deles é através do *Apache Lucene*. Os detalhes do funcionamento desse índice estão fora do escopo deste trabalho, mas na consulta dos testes e em muitas outras ele compõe uma fração pequena do processamento total, sendo o foco na adjacência livre de índices. Isso não quer dizer que o Neo4j não se beneficia de uma boa implementação de índices. Por exemplo, durante a inserção de uma grande quantidade de dados usando `MERGE`, como foi o caso para a criação da nossa base de testes, eles são muito importantes.

  Depois de encontrado o primeiro nó, a busca é toda através de expansões nos relacionamentos, tendo benefício completo da adjacência livre de índices. No primeiro passo, para cada nó no conjunto de nós iniciais, se percorre a lista ligada das relações associadas a eles e filtramos pelas que os têm como nó inicial. Isso é necessário pois a consulta define uma direção explícita, expressa pela ponta da seta em ASCII.

- **Passo recursivo.** Para cada nó encontrado no passo anterior, repetimos a expansão usando um caminho de comprimento variável entre nulo e unitário para incluir os nós encontrados até o momento para o próximo passo. Isso significa que expandimos novamente todos nós já encontrados, além dos novos, a cada etapa da recursão. Enquanto não muito eficiente, isso é mais próximo do que o PostgreSQL faz, além de ser a forma mais óbvia de expressar a intenção da busca em Cypher encontrada.

  Uma coisa a se ter em mente, entretanto, é a diferença grande para o caso sem filtro. Como no caso sem filtro usamos apenas uma expressão em Cypher, esse trabalho repetido não é executado. Nós já expandidos em uma determinada profundidade não são expandidos novamente. Entretanto, é possível que eles sejam encontrados e expandidos novamente em uma profundidade maior, talvez múltiplas vezes dependendo do número de caminhos que levam a eles. Para uma profundidade $n \leq 2$ isso não acontece e a consulta sem filtro acaba sendo mais rápida. Entretanto, a quantidade obtida nesses casos é pequena demais para isso ser importante e, a partir dessa profundidade, observamos nos testes um ganho de performance significativo usando a versão com filtro.

Como foi observado na hora que as consultas analisadas foram propostas, a consulta sem filtro é, de certo modo, forçada e não natural para a linguagem Cypher. Entretanto, o ganho de performance com ela, como veremos, é considerável. Uma performance maior poderia ser obtida ainda utilizando a API de Java do Neo4j, embora a comparação nesse caso com o PostgreSQL não fosse justa.

5.3 Comparação de performance

Na figura 5.2 está a comparação das duas ferramentas sem filtro e na figura 5.3 a com filtro, em escala logarítmica. Note que no caso sem filtro uma profundidade menor foi utilizada. Isso é porque, no PostgreSQL, as consultas em profundidades subsequentes eram inviáveis por questões de tempo e memória. A partir da profundidade 6, como explicamos anteriormente, os dados da busca se aproximam do total contido na base, levando a um crescimento quase linear, não acrescentando muito ao estudo.

No caso sem filtro, apenas a consulta do Neo4j para a profundidade 6 foi considerada pois o PostgreSQL não foi capaz de completar para a mesma profundidade. Observando o crescimento do caso sem filtro entre a profundidade 5 e 6 é possível ter uma ideia da quantidade de tempo que ele levaria se fosse possível, entrando já na escala de horas.

O resultado obtido indica que o Neo4j ganha do PostgreSQL por aproximadamente uma ordem de magnitude em profundidades menores, devido a denormalização, e em maiores, devido à vantagem proporcionada pela adjacência livre de índices para grandes expansões. Em profundidades médias a diferença é menor, devido aos dois fatores terem menos impacto para o PostgreSQL, mas o Neo4j ainda tem uma performance melhor.

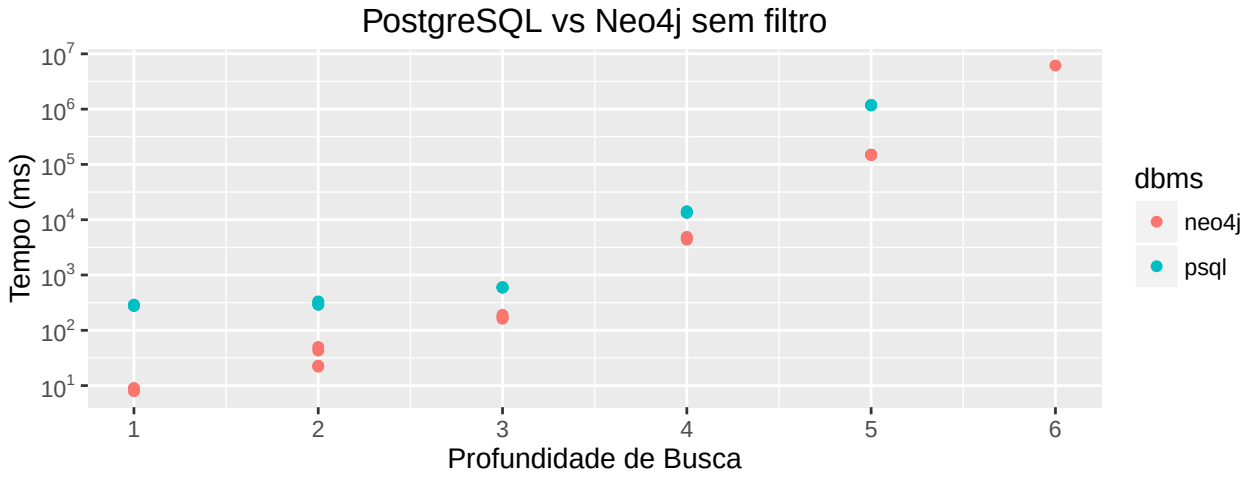

Figura 5.2: *Comparação no caso sem filtro, proveniente da consulta mais natural do Neo4j. O PostgreSQL não termina para profundidades maiores do que* 5.

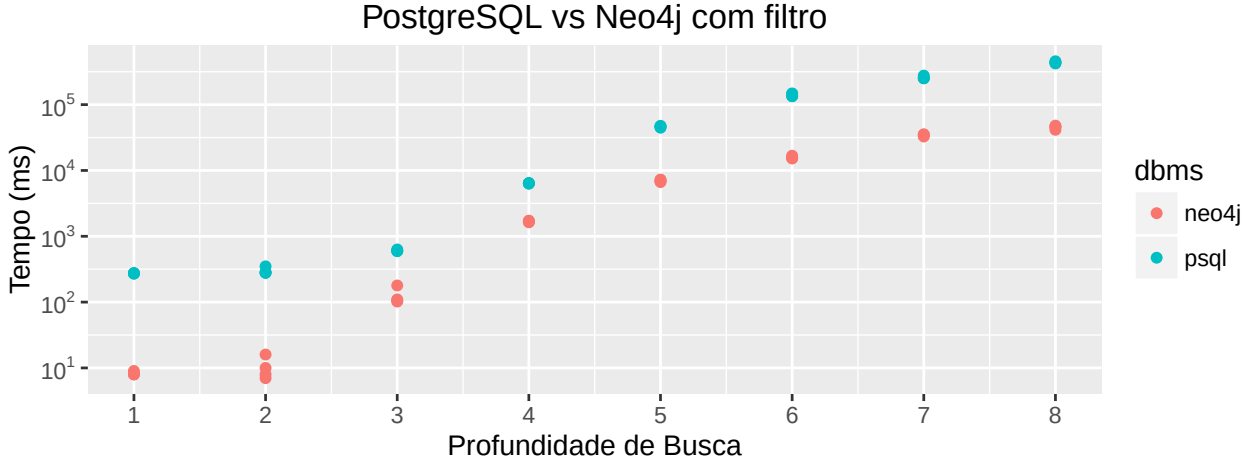

Figura 5.3: *Com filtro, evitando expansão sobre nós repetidos. A partir da profundidade* 6 *os nós encontrados já compoem a maior parte da base total, então o crescimento exponencial para.*

Capítulo 6

Algoritmos em grafos no Neo4j

Como já mencionamos, o modelo de grafos, assim como o modelo relacional, possui uma forte base matemática, nesse caso na teoria dos grafos. Longos anos de estudo e pesquisa nessa área levaram ao desenvolvimento de vários algoritmos eficientes, os quais temos interesse em utilizar para melhorar ainda mais nossa performance e, além disso, responder perguntas sobre nosso modelo que não seriam possíveis utilizando puramente SQL.

Uma pergunta simples que segue diretamente do nosso modelo é tentar descobrir o menor caminho ligando um artigo da Wikipedia a outro. Da teoria dos grafos, definimos o **comprimento** de um caminho como o número de relações seguidas nele e, dados dois nós a e b, a **distância** de a a b como o comprimento do menor caminho ligando a a b. É importante ter em mente que o nosso modelo de dados não é simétrico, ou seja, a existência de uma referência a um artigo numa página não implica que exista uma referência a essa página nesse artigo. Como consequência disso, fixando dois artigos a e b, a distância de a até b pode ser diferente da distância de b até a.

Usando agora nossa base da Wikipedia em português no Neo4j, podemos encontrar o caminho mais curto entre dois artigos com a seguinte consulta em Cypher:

```
MATCH p = shortestPath(
      (a:PtPage { title : "Modelo_relacional" })
      -[:linksto*]->
      (b:PtPage { title : "Teoria_dos_seis_graus_de_separacao" } ))
RETURN p;
```

Para a qual obtemos as seguintes respostas, uma para cada sentido da busca:

```
Node[124272]{title:"Teoria_dos_seis_graus_de_separacao"},:linksTo[43561148]{},
Node[785]{title:"Paul_Erdos"},:linksTo[35182604]{},
Node[27874]{title:"Teoria_dos_conjuntos"},:linksTo[43559714]{},
Node[8078]{title:"Modelo_relacional"}

Node[8078]{title:"Modelo_relacional"},:linksTo[31465730]{},
Node[1040]{title:"Telefone"},:linksTo[43275637]{},
Node[700361]{title:"Rio_de_Janeiro\_(cidade)"},:linksTo[38709191]{},
Node[30773]{title:"Igreja_Adventista_do_Setimo_Dia"},:linksTo[22362536]{},
Node[124272]{title:"Teoria_dos_seis_graus_de_separacao"}
```

Observe que elas diferem em comprimento, como mencionamos. Como nossa base de dados é baseada em uma imagem completa da Wikipedia, foi possível verificar os caminhos acima, mas esse resultado pode se tornar inválido conforme mudanças na Wikipedia em português ocorrerem. Além disso, os caminhos acima não são os únicos com comprimento mínimo. Se trocassemos a expressão `shortestPath` por `allShortestPaths`, seria possível encontrar todas as possibilidades de mesmo comprimento.

Nenhuma consulta desse tipo leva mais do que alguns milisegundos no Neo4j, mesmo para caminhos de grandes comprimentos. Ou seja, o Neo4j é capaz de resolver problemas práticos com eficiência muito maior do que a demonstrada na seção anterior com auxílio de

algoritmos em grafos. Existem formas de resolver o problema acima no PostgreSQL, mas o uso inevitável de `JOINs`, possivelmente numa CTE também, levaria a tempos de consulta similares ao que vimos no capítulo anterior.

Capítulo 7

Conclusão

Neste trabalho, vimos como o modelo orientado a grafos proporciona uma representação direta para o relacionamento implícito entre os dados, além de facilitar a modelagem, sendo ele similar a modelos conceituais como o entidade-relacionamento. Vimos também como podemos expressar nossas consultas de forma curta e intuitiva usando a linguagem Cypher no Neo4j. Além disso, mostramos que existem vantagens do ponto de vista de performance também, principalmente se precisamos resolver problemas clássicos de grafos como busca, por exemplo.

Apesar do estudo oferecido usar como base os dados da Wikipedia, os resultados fornecem evidência o suficiente para afirmar que o processamento de grafos é mais eficiente e substitui a, possivelmente, mais custosa operação de um banco de dados relacional, o `JOIN`. Além disso, existem problemas que não são facilmente modelados em SQL puro, resolvidos de forma natural através da linguagem Cypher, como encontrar caminhos mínimos entre nós.

Entretanto, todas essas vantagens são consequências de utilizar a ferramenta para resolver os problemas mais adequados para ela. Como observamos, houve uma facilidade muito maior para carregar os dados no PostgreSQL, devido ao formato tabular deles. Para o Neo4j, foi necessário um tratamento externo dos dados para diminuir o trabalho em convertê-los ao modelo de dados pela ferramenta. Outro problema observado foi a falta de flexibilidade na linguagem Cypher, sendo ela altamente condicionada para expressar consultas em grafos. Note que esses detalhes são específicos do Neo4j e outros bancos de dados orientados a grafos podem diferir.

Podemos concluir que as vantagens do modelo orientado a grafos certamente justificam seu uso em aplicações onde o relacionamento dos dados é tão ou mais importante do que os próprios dados para a tomada de decisões.

Referências Bibliográficas

[1] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2016. 1, 5, 14

[2] Mediwiki.org. https://www.mediawiki.org/wiki/MediaWiki. Último Acesso: 2016-11-19. 3

[3] PostgreSQL documentation. https://www.postgresql.org/docs/9.6/static/index.html. Último Acesso: 2016-11-13. 8

[4] I. Robison, J. Webber, and E. Elfrem. *Graph Databases*. O'Reilly, 1st edition, 2013. 6, 14

[5] O. K. Takai, I. C. Italiano, and J. E. Ferreira. *Introdução a Banco de Dados*. 2005. 5

[6] Walmart optimizes customer experience with real-time recommendations. https://neo4j.com/case-studies/walmart/. Último Acesso: 2016-11-19. 1