UNIVERSIDADE FEDERAL DE UBERLÂNDIA

João Victor Fernandes de Souza Silva

Comparação de desempenho entre os bancos de dados PostgreSQL e Neo4j para acesso a dados complexos

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

João Victor Fernandes de Souza Silva

Comparação de desempenho entre os bancos de dados PostgreSQL e Neo4j para acesso a dados complexos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Dra. Maria Camila Nardini Barioni

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2023

Resumo

O volume de dados gerados atualmente é enorme e vem aumentando notoriamente. Concomitantemente, os dados estão se tornando mais esparsos e semiestruturados e são vários os SGBDs existentes para o gerenciamento destes. Neste contexto, o presente trabalho abordou a tarefa de realizar a comparação de desempenho entre os SGBDs Neo4j e PostgreSQL ao acesso a dados complexos devido a popularidade e importância de cada um para a sua categoria e o uso das tecnologias de grafo e relacional. Sendo assim, objetivou-se a proposição de uma aplicação para a realização dos testes para a comparação de desempenho, a partir do conjunto de dados de citações de patentes. Os testes foram criados tendo como base os pontos fortes de cada SGBD, onde foram abordadas consultas simples com índices, consultas com agregação, travessia e correspondência de padrão, além de testes de carga de dados. Após a execução dos experimentos, notou-se que os resultados não divergiram dos resultados obtidos por experimentos de outras pesquisas e trabalhos correlatos que usam de mesma abordagem, onde o banco de dados relacional PostgreSQL obteve desempenhos superiores ao Neo4j em vários aspectos.

Palavras-chave: PostgreSQL, Neo4j, dados complexos, comparação de desempenho.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

SQL	*Structured Query Language*
BFS	*Breadth-First Search*
DFS	*Depth-First Search*
MVCC	*Multiversion concurrency control*
CAP	*Consistency*, *Availability* e *Partition tolerence*
API	*Application Programming Interface*
CRUD	Acrônimo em inglês que representa as quatros operações básicas de persistência de dados
TCC	Trabalho de conclusão de curso
SGBD	Sistema de gerenciamento de bancos de dados

Sumário

1 Introdução

Sabe-se que o volume de dados gerados atualmente é enorme e vem aumentando notoriamente, com novas informações geradas todos os dias por meio de aplicativos, sistemas, aparelhos com IoT, redes sociais, dentre outros. Concomitantemente, os dados estão se tornando mais esparsos e semiestruturados, conforme já destacado por Professional... (2011), e é cada vez mais um requisito para os sistemas de informação a manipulação de dados mais complexos. No entanto, como já ressaltado por Barioni (2006), os SGBDs foram tradicionalmente desenvolvidos para suportar o armazenamento e a recuperação eficientes de grandes volumes de dados compostos apenas por números e pequenas cadeias de caracteres. Assim, a manipulação de dados complexos de forma eficiente tornou-se um objetivo importante para a área de tecnologia.

Dentre os principais dados complexos, pode-se mencionar as imagens, documentos, vídeos e arquivos MP3, onde os mesmos podem ser caracterizados por não possuirem uma informação bidimensional e serem armazenados como registro em uma linha que não é necessariamente um vetor e sim um objeto, onde este contém todos os itens de informação para um, e apenas um, registro. Tendo em vista essas caracterizações, os bancos de dados não relacionais se tornaram opções plausíveis para o armazenamento e a recuperação de dados complexos, devido a natureza destes.

Outrossim, são vários os bancos de dados de base não relacional e a aplicação de cada um é determinada pelo contexto no qual o mesmo está inserido. Assim sendo, devido a importância de grafos para a manipulação de *big data*, a facilitação para lidar com grandes volumes de dados, com relacionamentos dinâmicos e altamente conectados, os bancos de dados não relacionais orientados a grafo se tornam uma boa opção para lidar com dados complexos (SADALAGE; FOWLER, 2013). Entretanto, é importante ressaltar que o uso de bancos de dados não relacionais é comparativamente menor do que o uso de bancos de dados relacionais, tendo como base o ranking de popularidade do site DB-Engines (2022), onde o método de cálculo do mesmo recebe como parâmetros o número de menções e resultados em sites de busca, interesse geral no sistema, frequência de discussões técnicas sobre o sistema, número de ofertas de emprego em que o sistema é mencionado, número de perfis no Linkedin em que o sistema é mencionado e a relevância nas redes sociais.

Logo, este trabalho de conclusão de curso tem como objetivo comparar o desempenho ao acesso a dados complexos entre os bancos de dados relacionais e os bancos de dados não relacionais orientados a grafo para identificar em quais situações cada um é mais vantajoso. A escolha pelos bancos de dados PostgreSQL e Neo4j para a realização

da comparação de desempenho neste trabalho de conclusão de curso foi tomada levando em consideração o ranking de novembro de 2022 de cada um no site DB-Engines (2022), onde o Neo4j ocupa a 19ª posição no ranking geral, envolvendo todos os SGBDs, e a 1ª posição no ranking de bancos de dados de grafo e, por sua vez, o PostgreSQL ocupa a 5ª posição no ranking geral, a 4º posição no ranking de banco de dados relacionais, além de ser o banco de dados *open-source* mais usado no mercado. Ademais, a literatura sobre comparação de desempenho entre SGBDs é vasta, cada uma com seus critérios de comparação, mas poucas são as de comparação entre os SGBDs PostgreSQL e Neo4j para o tema de acesso a dados complexos. Entretanto, o assunto é importante levando em conta sua especificidade, e procura-se estimular sua investigação com a revisão do estado da arte no assunto.

1.1 Objetivos

O objetivo geral do trabalho descrito aqui consiste em analisar o desempenho entre os bancos de dados PostgreSQL e Neo4j no acesso a dados complexos. Objetivos específicos estão atrelados a este objetivo geral, sendo os mesmos:

- Verificação da maturidade dos bancos de dados não relacionais orientados a grafo ao acesso e manipulação de dados complexos;
- Verificação do nível de complexidade de acesso a dados complexos com um SGBD relacional;
- Investigação dos principais critérios de desempenho ao acesso a dados;
- Analisar os cenários em que cada banco de dados é mais adequado ao acesso a dados complexos.

1.2 Justificativa

Como já ressaltado anteriormente, os dados estão se tornando mais complexos, onde as aplicações têm cada vez mais que lidar com imagens, documentos, arquivos MP3, filmes, informações geo-referenciadas, dentre outras. Assim, com essa análise de desempenho entre os principais bancos de dados das categorias relacional e não relacional orientado a grafo, o leitor terá o resultado de uma análise profunda de desempenho para ajudá-lo na tomada de decisão para a definição do melhor banco de dados para o contexto do mesmo para acesso a dados complexos, além de auxiliar pesquisadores e estudantes. Além disso, esse tema não é abordado durante o curso de Bacharelado de Sistemas de Informação na Universidade Federal de Uberlândia, mas o mesmo tem muito a agregar a um estudante ou profissional da área de tecnologia.

1.3 Organização do Texto

O texto deste trabalho é composto pelo capítulo 2, Fundamentação Teórica, que visa fazer o embasamento teórico necessário para melhor compreensão dos temas abordados, trazendo conceitos introdutórios relacionados a teoria de grafos, dígrafos, ao modelo relacional, modelo não relacional, SGBDs relacionais e não relacionais, além de conceitos relacionados aos SGBDs não relacionais orientados a grafo, PostgreSQL e Neo4j. O capítulo seguinte, Trabalhos Relacionados, aborda trabalhos relacionados com o mesmo tema ou que se utilizam de abordagens semelhantes as que são abordadas neste TCC. Em seguida, o capítulo 4, Método de Trabalho, descreve o método de trabalho desta pesquisa, apresentando como o conjunto de dados foi obtido e modelado e como a aplicação para a comparação de desempesenho foi desenvolvida e funciona. Já no capítulo Resultados e Discussões (capítulo 5) são apresentados a metadologia dos testes, a infraestrutura utilizada e todos os resultados obtidos referentes aos testes de comparações entre os SGBDs Neo4j e PostgreSQL a partir dos conjuntos de dados de diferentes tamanhos, além de apresentar também discussões acerca destes resultados. Finalmente, o último capítulo conclui o trabalho apresentando os novos conhecimentos adquiridos e, além disso, sugere atualizações na aplicação desenvolvida para a realização de testes para continuidade desta pesquisa.

2 Fundamentação Teórica

Visando o embasamento teórico para melhor compreensão do TCC descrito aqui e dos temas abordados neste, esta seção apresenta as principais teorias e conceitos relacionados aos bancos de dados relacionais e não relacionais orientados a grafos e as principais diferenças e semelhanças entre os SGBDs PostgreSQL e Neo4j.

2.1 Bancos de Dados Relacionais e Não Relacionais Orientados a Grafos

A forma como os dados são modelados e as interfaces disponibilizadas para a manipulação dos mesmos impactam diretamente na eficiência no acesso aos dados. Assim, esta seção resume as principais características e conceitos relacionados aos bancos de dados relacionais e não relacionais orientados a grafo.

2.2 Bancos de Dados Relacionais

Os bancos de dados relacionais são caracterizados, principalmente, por sua base de modelo relacional, pelos seus relacionamentos realizados por meio de campos de chave estrangeira, pelas propriedades ACID que garantem a confiabilidade nas transações e o uso da linguagem de consulta declarativa SQL. Ademais, de acordo com o ranking de novembro de 2022 do site DB-Engines (2022), os bancos de dados de base relacional lideram o ranking de popularidade entre os bancos de dados.

Entretanto, é importante ressaltar também que os bancos de dados relacionais possuem algumas limitações e complexidades, sendo que estas impulsionaram o surgimento dos bancos de dados não relacionais. Dentre suas limitações e complexidades, tendo como viés o tema deste TCC, vale destacar, de acordo com Zhou e Ordonez (2019), a complexidade para lidar com problemas de grafos, problemas importantes para lidar com *big data*, e a limitação de escalabilidade que está relacionada ao crescimento exponencial, pois há uma certa dificuldade para conciliar o tipo de modelo com a demanda da escalabilidade.

2.2.1 Modelo Relacional

Tendo com base o livro do Guimarães (2003), os bancos de dados relacionais têm sua base no modelo relacional, modelo proposto por Edgar Codd em 1970 para representação de dados. Este modelo visa padronizar a representação dos dados, principalmente, por meio de relações, esquemas e formas normais.

No modelo relacional, a relação é a principal construção para a representação dos dados, sendo a mesma uma tabela com linhas não ordenadas e colunas. Outrossim, as relações podem possuir instâncias, onde cada uma representa um conjunto de linhas (tuplas ou registros) distintas entre si, que compõem a relação em um dado momento. Concomitantemente, o esquema especifica o nome da relação e o domínio de cada coluna. Por exemplo, considere a Figura 1 abaixo representando o diagrama Modelo-Relacional, na notação *Crow's Feet* (PUJA; POSCIC; JAKSIC, 2019), de uma relação que define a nota e a quantidade de faltas de um aluno por curso. Para este exemplo, as relações são as tabelas aluno, curso e aluno_curso. Já o esquema de cada tabela, pode ser definido da seguinte forma:

- tabela aluno: aluno(aluno_id: int, nome: char[200], matricula: char[12], data_nasc: date)
- tabela curso: curso(curso_id: int, nome: char[200], carga_horaria: int)
- tabela aluno_curso: curso(aluno_id: int, curso_id: int, nota: float, qtd_faltas: int)

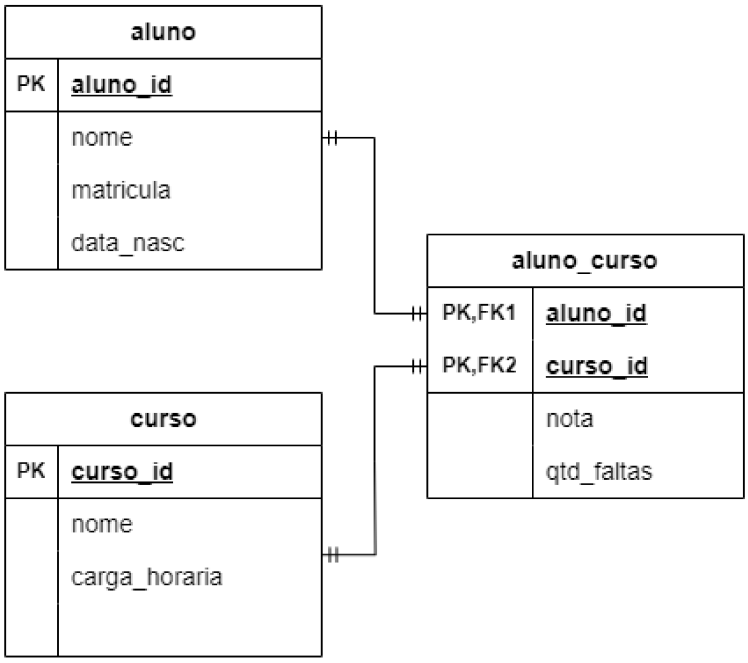

Figura 1 – Diagrama do Modelo-Relacional da relação exemplificada entre aluno e curso.

Por sua vez, as instâncias para as tabelas aluno, curso e aluno_curso podem ser exemplificadas, respectivamente pelas Tabelas 1, 2 e 3.

Além disso, este modelo ainda impõe algumas restrições de integridade. De acordo com Date (2004), uma restrição de integridade pode ser definida como uma expressão

aluno_id	nome	matricula	data_nasc
1	Marcos	11111BSI111	23-06-1999
2	Maria	11111BSI112	01-01-2001

Tabela 1 – Instâncias da tabela exemplificada aluno.

curso_id	nome	carga_horaria
1	Introdução a Banco de Dados	60

Tabela 2 – Instâncias da tabela exemplificada curso.

aluno_id	curso_id	nota	qtd_faltas
1	1	50	10
2	1	75	2

Tabela 3 – Instâncias da tabela exemplificada aluno_curso.

associada ao banco que precisa ser avaliada o tempo todo como verdadeira. Dentre as restrições, vale destacar:

- Restrições de tipo: definem o conjunto de valores de um determinado tipo;
- Restrições de atributo: especificam os valores que um atributo pode assumir;
- Restrições sobre relações: no esquema da base de dados, uma condição é definida a fim de restringir a informação a ser armazenada;
- Restrições de chaves primárias: garante que as tuplas de uma relação sejam únicas;
- Restrições de chave estrangeira: o banco de dados não pode conter quaisquer valores de chaves estrangeiras não correspondentes. Assim, como uma informação de uma relação pode estar ligada à informação de outra relação, se uma delas é modificada a outra também deve ser checada e modificada de maneira a garantir a consistência da informação.

Ademais, o modelo relacional ainda propõe a normalização do banco de dados por meio de pelo menos três etapas de validação de acordo com as formas normais. Resumidamente, de acordo com Machado (2017) essa normalização visa eliminar anomalias e redundâncias nos dados, visando garantir a integridade dos dados e eficiência das consultas.

2.2.2 Transações e Propriedades ACID

Segundo Date (2004), uma transação em bancos de dados relacionais pode ser definida como uma unidade lógica de trabalho, que se inicia com a execução de uma

operação BEGIN TRANSACTION e termina com a execução de uma operação COMMIT ou ROLLBACK. A operação COMMIT indica que uma unidade lógica de trabalho foi executada com sucesso. Por outro lado, o ROLLBACK indica que uma transação não foi executada com sucesso, por alguma falha, e que as atualizações devem ser desfeitas.

Por exemplo, considere o exemplo abaixo para a atualização da nota de um aluno, onde o mesmo terá sua nota incrementada em 30 pontos. Caso a operação não seja concluída por alguma falha, a alteração será desfeita. Caso a atualização seja bem sucedida, o mesmo terá sua nota incrementada com sucesso.

```
BEGIN TRANSACTION;
UPDATE aluno_curso 12 { nota := nota + 30 } ;
IF ocorreu alguma falha THEN GO TO UNDO ;
END IF;
COMMIT; /* término bem sucedido */
GO TO FINISH;
UNDO:
  ROLLBACK;/* término com erro */
FINISH:
  RETURN;
```

Assim, as propriedades ACID garantem a confiabilidade nas transações, sendo as mesmas *Atomicidade*, *Correção*, *Isolamento* e *Durabilidade*. Resumindo:

- Atomicidade: as transações são atômicas, visto que a transação será executada totalmente ou não será executada;
- Consistência: as transações criam um estado válido dos dados ou em caso de falha retorna todos os dados ao seu estado antes que a transação foi iniciada;
- Isolamento: as transações são isoladas umas das outras. Ou seja, as atualizações de uma transação só serão visualizadas por outra transação após o COMMIT;
- Durabilidade: após o COMMIT ser executado com sucesso, as atualizações são registradas no banco de dados de tal forma que mesmo no caso de uma falha e/ou reinício do sistema, os dados estão disponíveis em seu estado correto.

2.3 Bancos de Dados Não relacionais

Conforme já citado por Sadalage e Fowler (2013), os bancos de dados não relacionais surgiram como proposta para resolver problemas referentes a manipulação de grandes volumes de dados não estruturados e semiestruturados, disponibilidade e escalabilidade.

Os bancos de dados não relacionais apresentam algumas características fundamentais que os diferenciam dos sistemas de bancos de dados relacionais, tornando-os adequados para armazenamento de grandes volumes de dados não estruturados ou semiestruturados. Segundo Lóscio, Oliveira e Pontes (2011), as principais características desses bancos de dados são:

- Escalabilidade Horizontal: a ausência de bloqueios permite a escalabilidade horizontal e torna os bancos de dados não relacionais adequados para solucionar os problemas de gerenciamento de volumes de dados que crescem exponencialmente;
- Ausência de esquema ou esquema flexível: a ausência completa ou quase total do esquema que define a estrutura dos dados modelados facilita tanto a escalabilidade quanto contribui para um maior aumento da disponibilidade. Entretanto, não há garantia da integridade dos dados, o que ocorre nos bancos relacionais;
- Suporte nativo a replicação: esta característica permite a replicação de forma nativa e diminui o tempo gasto para recuperar informações. Existem duas abordagens principais para replicação, sendo as mesmas Master-Slave e Multi-Master;
- API simples para acesso aos dados: APIs são desenvolvidas para facilitar o acesso a informações, permitindo que qualquer aplicação possa utilizar os dados do banco de forma rápida e eficiente;
- Consistência eventual: a consistência nem sempre é mantida entre os diversos pontos de distribuição de dados. Assim, esta característica tem como princípio o teorema CAP (*Consistency*, *Availability* e *Partition tolerence*), na qual diz, que em um dado momento, só é possível garantir duas de três propriedades entre consistência, disponibilidade e tolerância à partição;
- *Map/Reduce*: suporte ao manuseio de grandes volumes de dados distribuídos ao longo dos nós de uma rede. Na fase de *map*, os problemas são quebrados em subproblemas que são distribuídos em outros nós na rede. Já na fase *reduce*, os subproblemas são resolvidos em cada nó filho e o resultado é repassado ao pai, que, sendo ele também filho, repassaria ao seu pai, e assim por diante até chegar ao nó raiz do problema;
- *Consistent hashing*: esta característica suporta mecanismos de armazenamento e recuperação em banco de dados distribuído. O uso desta técnica é interessante, pois evita muita migração de dados;
- *Multiversion concurrency control* (MVCC): mecanismo que dá suporte a transações paralelas em um banco de dados. Ao contrário do esquema clássico de gerenciamento

de transações, permite que operações de leitura e escrita sejam feitas simultaneamente;

- *Vector clocks*: são usados para gerar uma ordenação dos eventos acontecidos em um sistema.

De acordo com Hecht e Jablonski (2011), os bancos de dados NoSQL podem ser divididos/classificados em quatro grupos, sendo os mesmos orientados a chave e valor, documentos, colunas e a grafos. Tendo como referência os banco de dados não relacionais orientados a grafo, vale ressaltar também que os mesmos objetivam um desempenho aprimorado na gestão eficiente de bases onde os dados são fortemente vinculados e os mesmos podem ser utilizados, por exemplo, em serviços baseados em localização geográfica, representação do conhecimento, sistemas de recomendação e em qualquer outra aplicação que se utilize de relações complexas (LOPES, 2014).

2.4 Banco de Dados Não Relacionais Orientados a Grafos

Os bancos de dados orientados a grafos são caracterizados, principalmente, por utilizarem a teoria dos grafos em sua implementação, onde faz-se o uso de grafos direcionados (dígrafos), nós e arestas para a representação do conjunto de dados, onde os nós representam as entidades e as arestas os relacionamentos. Esta abordagem oferece diversas vantagens para consulta, modelagem e análise de dados altamente conectados e complexos.

2.4.1 Teoria de Grafos

Tendo em vista o livro do Lucchesi (1979), devido ao seu trabalho, o matemático Euler é frequentemente conhecido como pai da teoria de grafos. Em 1736, Euler resolveu o problema das Pontes de Konigsberg, abstraindo as características físicas das pontes, representando-as por vértices e as conexões entre elas por arestas. Assim sendo, Euler criou o conceito de grafo e iniciou a teoria dos grafos.

Ao passar do tempo, a teoria dos grafos tornou-se uma disciplina da matemática que estuda as propriedades e as relações entre os elementos de um conjunto de vértices (nós) conectados por arestas, sendo amplamente utilizada na área da computação na estruturação de dados, criação de algoritmos de buscas, dentre outras soluções de problemas que envolvem, principalmente, relações e interconexões.

Tendo como base o livro de Chartrand, Lesniak e Zhang (2004), a estrutura de um grafo é composta por um conjunto de vértices (nós) e um conjunto de arestas que conectam esses vértices. Por exemplo, considere a Figura 2, que demostra o grafo com os nós aluno e curso, e as arestas representando a relação entre aluno e curso.

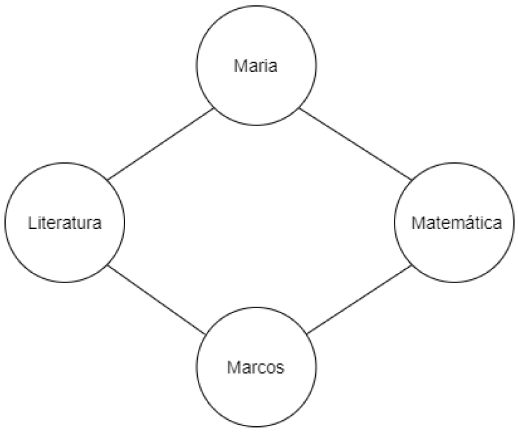

Figura 2 – Grafo para relação entre aluno e curso.

Além disso, os grafos podem ser classificados principalmente em grafos direcionados e não direcionados, além de ponderados e bipartidos. Para a manipulação de um grafo, o mesmo possui propriedades e operações. Para as propriedades, podemos citar essencialmente de acordo com sua classificação:

- Grau de um vértice: para grafos direcionados, é o grau de entrada e saída de um vértice, para um grafo não dirigido, é o número de arestas incidentes a esse vértice;
- Caminho: é a sequência de um vértices conectados por arestas, sendo cada vértice no caminho adjacente ao próximo vértice no caminho;
- Componentes conectados: são subgrafos nos quais todos os vértices estão conectados por caminhos.

Já para as operações em grafos, pode-se citar resumidamente:

- Combinação de grafos: permite a união de dois ou mais grafos;
- Remoção de vértices e arestas: permite excluir vértices e arestas do grafo;
- Consulta de caminhos: permite encontrar caminhos entre vértices específicos, como por exemplo o caminho mais curtos ou que satisfaça condições específicas;
- Consulta de vizinhaça: permite recuperar vizinhos de um vértice por meio de arestas;
- Inserção de vértices e arestas: permite a inserção de novos vértices e arestas;
- Atualização de vértices e arestas: permite modificar propriedades de vértices e arestas do grafo.

Portanto, devido às suas características, a teoria de grafos possui uma ampla gama de aplicações na computação, sendo aplicada em diversos algoritmos. Abaixo estão alguns algoritmos de suma importância para a área de computação que faz o uso da teoria de grafos:

- Dijkstra e Bellman-Ford: algoritmos utilizados para encontrar caminhos mais curtos entre dois vértices ou para calcular a distância mínima a partir de um vértice (SANTOS; BRITO; BARBOSA, 2019);
- BFS e DFS: *Breadth-first search* (BFS) e *depth-first search* (DFS) são algoritmos de busca utilizados para explorar grafos e encontrar vértices ou arestas com properiedades específicas (EVERITT; HUTTER, 2015);
- Algoritmos *Louvain* e *Label Propagation*: De acordo com Neo4j (2023), *Louvain* e *Label Propagation* são algoritmos de agrupamento utilizados para identificar e agrupar comunidades com base em suas conexões.

2.4.2 Principais Conceitos de Dígrafos

Um dígrafo ou grafo direcionado é uma extensão do conceito de grafo, onde as arestas possuem uma direção específica. Segundo Anton e Rorres (2012), os grafos direcionados (dígrafos) possuem um conjunto de vértices/nós (V) e um conjunto de arestas (A), sendo s, t : A $\rightarrow$ V, onde s (e) é a fonte e t (e) é o alvo da aresta direcionada e. Outrossim, os dígrafos estão diretamente relacionados a implementação dos bancos de dados orientados a grafos, visto o significado dado às entidades e seus subnós, interligados pelas arestas. Vale destacar também, segundo AWS (2023), que uma aresta tem sempre um nó inicial, um nó final, um tipo e um direcionamento, possibilitando assim a descrição dos relacionamentos entre pais e filhos, das ações e das propriedades. Além disso, um grafo pode ser atravessado com tipos de arestas específicas ou por todo o grafo, em que essa travessia ocorre muito rapidamente, uma vez que os relacionamentos entre os nós não são calculados no momento das consultas e sim na criação. Considere a Figura 3 para representação de um grafo direcionado para relação entre aluno e curso.

Além do seu conceito fundamental, vale ressaltar também os principais conceitos e propriedades que desempenham um papel fundamental na análise e manipulação de dígrafos de acordo com Cormen et al. (2009), tais como:

- Grau de entrada e grau de saída de um vértice: o grau de entrada e saída de um vértice é importante para compreender a estrutura e conectividade do grafo, referindo-se, respectivamente, ao número de arestas/conexões que chegam e saem desse vértice;

- Caminho direcionado: pode ser definido como sequência de arestas em que cada aresta possui um vértice $_{vi}$ como origem e um vértice $_{vi + 1}$ como destino. Os caminhos direcionados podem ser utilizados para análise de conectividade, algoritmos de caminho mínimo e análise de fluxo;
- Componentes fortemente conectados: podem ser definidos como conjuntos de vértices em um dígrado em que é possível alcançar qualquer vértice a partir de qualquer outro vértice dentro desse conjunto, seguindo caminhos direcionados.

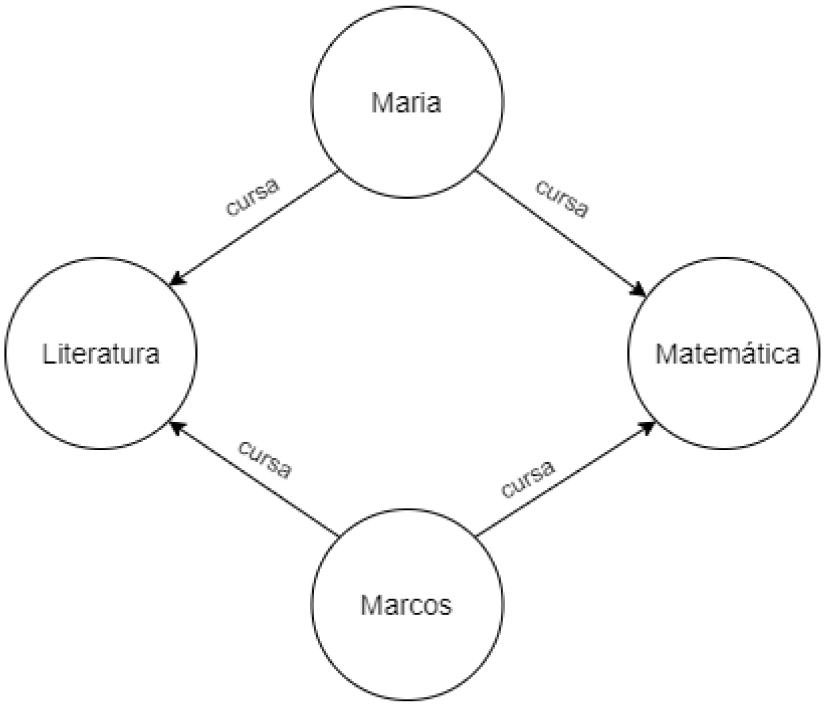

Figura 3 – Dígrafo: representação para relação entre aluno e curso.

2.5 Diferenças e Semelhanças Entre os SGBDs PostgreSQL e Neo4j

Além da base de modelo e a teoria por traz de cada SGBD, é de suma importância também descrever as principais características do PostgreSQL e Neo4j para analisar as principais diferenças e semelhanças entre estes dois SGBDs.

De acordo com o Neo4j (2023), o Neo4j é uma plataforma capaz de proporcionar a manutenção de um banco de dados com diversas características, destancando-se a robustez, a escalabilidade, o alto desempenho em consultas obtido por meio da utilização de *traversals* presentes em linguagens declarativa e, ainda, a capacidade de lidar com escalas de milhares de milhões de nós inter-relacionado. Além disso, é mencionada também a capacidade de garantir uma das mais importantes características dos bancos relacionais,

as propriedades ACID. Outrossim, o Neo4j é o banco de dados não relacional orientado a grafo mais utlizado no mercado, ocupando a 1ª posição no ranking de sua categoria no site DB-Engines (2022).

Ademais, de acordo com PostgreSQL (2023), o PostgreSQL é um SGBD de base relacional *open source*. As origens do PostgreSQL remontam a 1986 como parte do projeto POSTGRES da Universidade da Califórnia em Berkeley e tem mais de 35 anos de desenvolvimento ativo na plataforma principal. Dentre as principais características, destacam-se a arquitetura comprovada, confiabilidade, integridade de dados, conjunto robusto de recursos, extensibilidade e dedicação da comunidade de código aberto por trás do software para oferecer consistentemente soluções inovadoras e de alto desempenho. Além disso, o PostgreSQL é compatível com ACID desde 2001. Outrossim, o PostgreSQL é o banco de dados *open source* mais usado no mercado. Além disso, de acordo com o site DB-Engines (2022), o PostgreSQL ocupa a 5ª posição no ranking geral e a 4º posição no ranking de sua categoria.

Outrossim, vale ressaltar também a diferença referente a cada linguagem de consulta utilizada por cada SGBD. O Neo4j faz o uso da linguagem de consulta Cypher, em contrapartida o PostgreSQL faz o uso da linguagem de consulta SQL.

Tendo como base o artigo de (FRANCIS et al., 2018), a linguagem de consulta Cypher foi desenvolvida especificadamente para bancos de grafos. Ela fornece uma maneira intuitiva e expressiva de interagir com os dados armazenados no formato de grafo. Já de acordo com a documentação do Neo4j (2023), para a manipulação dos dados, o Cypher faz o uso, principalmente, dos operadores lógicos AND, OR e NOT, dos operadores de comparação =, <>, <, >, <= e >=, do operador WHERE para filtragem de dados, dos operadores de criação CREATE, SET e DELETE, do operador de relacionamento ->, das cláusulas RETURN para retornar os elementos e MATCH para localizar padrões de grafos e das cláusulas GROUP BY e ORDER BY para agrupamento e ordenação, dentre outros.

Tendo como referência o artigo de Jamison (2003), a linguagem de consulta SQL foi desenvolvida para interagir com bancos de dados relacionais, visando gerenciar, manipular e recuperar dados em um formato tabular, organizado em tabelas compostas por linhas e colunas. Já de acordo com a documentação do PostgreSQL (2023), para este gerenciamento de dados, ela faz o uso dos operadores lógicos AND, OR e NOT, dos operadores de comparação =, <>, <, >, <=, >= e !=, do operador de comparação de padrões LIKE, das cláusulas SELECT e FROM para selecionar e retornar dados, das cláusulas de manipulação de dados INSERT, UPDATE e DELETE, da clásula JOIN para combinação de tabelas e das cláusulas GROUP BY e ORDER BY para agrupamento e ordenação, dentre outras.

Abaixo são mostrados alguns exemplos de operações CRUD básicas utilizando

cada linguagem de consulta e a tabela/nó aluno. Com SQL:

- *CREATE*:

```
CREATE TABLE IF NOT EXISTS aluno(
    aluno_id INT PRIMARY KEY NOT NULL UNIQUE,
    nome VARCHAR(120) NOT NULL,
    matricula VARCHAR(12) NOT NULL,
    data_nasc DATE NOT NULL
);

INSERT INTO aluno (nome, matricula, data_nasc)
VALUES ('Marcos', '11111BSI111', '23-06-1999')
```

- *READ*:

```
SELECT * FROM aluno
WHERE matricula = '11111BSI111'
```

- *UPDATE*:

```
UPDATE TABLE aluno
SET nome = 'Marcos F. Silva'
WHERE matricula = '11111BSI111'
```

- *DELETE*:

```
DELETE FROM aluno
WHERE matricula = '11111BSI111'
```

Com Cypher:

- *CREATE*:

```
CREATE (a:aluno
{
    nome: 'Marcos F. Silva',
    matricula: '11111BSI111',
    data_nasc: '23-06-1999'
})
```

- *READ*:

```
MATCH (a:aluno)
WHERE matricula = '11111BSI111' RETURN a
```

- *UPDATE*:

```
MATCH (a:aluno)
WHERE matricula = '11111BSI111'
SET a.nome = 'Marcos F. Silva'
```

- *DELETE*:

```
MATCH (a:aluno)
WHERE matricula='11111BSI111'
DELETE a
```

Por fim, a tabela 4, de forma objetiva, mostra um resumo sobre as principais diferenças e similiaridades entre os SGBDs PostgreSQL e Neo4j, de acordo com o objetivo deste TCC.

SGBDs	PostgreSQL	Neo4j
Base de modelo	Modelo relacional	Modelo não relacional e de grafos
Representação dos dados	Conjunto de relações	Grafos dirigidos
Reprensentação dos relacionamentos	Chave estrangeira ou uma tabela de relacionamento	Arestas
Reprensentação das entidades	Linhas e colunas	Nós
Linguagem de consulta	SQL	Cypher2
Possui propriedades ACID	Sim	Sim

Tabela 4 – Principais características dos SGBDs PostgreSQL e Neo4j.

3 Trabalhos Relacionados

Esta seção apresenta trabalhos relacionados com o mesmo tema ou que se utilizam de abordagens semelhantes as que são abordadas neste TCC. Neste contexto, são discutidos alguns dos trabalhos já realizados.

Em Homrich e Mergen (2018) é realizada uma comparação de desempenho ao acesso a dados complexos entre os SGBDs MySQL (sendo aqui o representante dos bancos de dados relacionais) e Neo4j, levando em consideração o tempo de execução (em segundos) em tarefas como a carga de dados e alguns cenários de consulta, com foco na utilização de apenas recursos padrão das linguagens de acesso dos SGBDs. O domínio de dados é composto pelo grafo da rede rodoviária do estado da Califórnia, proveniente da biblioteca de análise de rede e mineração de grafos de uso geral SNAP (2023). Os resultados mostram a prevalência do MySQL, utilizando as engines InnoDB e MyISAM, nos testes. Onde, para cada categoria dos testes, tem-se:

- Desempenho das cargas de dados no MySQL e no Neo4j: O desempenho do Neo4j foi inferior aos do MySQL em qualquer sentido, com inserções consideravelmente mais dispendiosas mesmo na importação, com comandos otimizados para inserção massiva;
- Desempenho das consultas com filtragem de dados com índice: O Neo4j apresentou desempenhos inferiores aos obtidos pelas engines do MySQL, com tempo médio de execução próximo de um segundo. A engine InnoDB apresentou o melhor desempenho para o MySQL, com execução aproximadamente três vezes mais rápida que a de MyISAM na aplicação de um filtro;
- Desempenho das consultas com cruzamento de dados: A engine InnoDB obteve um desempenho mais satisfatório que o Neo4j, sendo aproximadamente sete vezes mais rápida. O tempo de execução da engine MyISAM manteve-se constante;
- Desempenho das consultas por caminhos cujos nós satisfazem uma condição: O MySQL apresentou tempos de execução superiores ao Neo4j, com a engine InnoDB obtendo o melhor desempenho no geral. Os tempos de execução do Neo4j foram de três a cinco vezes maiores que os tempos da engine InnoDB, dependendo do comprimento do caminho analisado. Os tempos de execução dos dois SGBDs apresentaram variação conforme o aumento do comprimento.

Já em Soares (2013) é realizada uma comparação de desempenho entre os SGBDs PostgreSQL e Neo4j na gerência de dados de proveniência. No experimento, foi levado em

consideração o tempo de execução de consultas com travessias e sem travessias e o uso de recursos computacionais. O domínio de dados é composto por três conjuntos de dados diferentes baseados em documentos XML que seguem o formato *open Provenance Model* (OPM), documentos gerados a partir de execuções de *workflows* em diferentes sistemas de gerência de workflows científicos (SGWFC). Os seguintes resultados foram obtidos para cada categoria de teste:

- Uso de recursos computacionais: o Neo4j exigiu muito mais do que o PostgreSQL, independentemente da forma com que as consultas foram implementadas;
- Tempo de execução para consultas com travessias: o Neo4j foi substancialmente melhor que o PostgreSQL em todas as consultas;
- Tempo de execução para consultas sem travessias: A API PostgreSQL JDBC obteve o melhor tempo médio na maioria das consultas.

No trabalho proposto por Uriarte (2018), é realizada uma comparação de desempenho entre os SGBDs MongoDB, Neo4j e PostgreSQL na gerência de dados espaciais provenientes do OpenStreetMap (2023) levando em consideração o tempo de execução. Após os experimentos, os seguintes resultados foram obtidos:

- Desempenho na realização das inserções de dados: O PostgreSQL obteve resultados superiores em todos os testes de inserção, com menor tempo de inserção entre os três SGBDs. O Neo4j por sua vez obteve o maior tempo;
- Desempenho nas realizações de consultas *within*: O MongoDB obteve resultados superiores, já o Neo4j obteve um desempenho nitidamente inferior, com tempo de execeução consideravelmente maior do que o do PostgreSQL e do MongoDB;
- Desempenho nas realizações de consultas *intersects*: O PostgreSQL demonstrou resultados superiores, por outro lado o Neo4j e o MongoDB obtevem resultados similares;
- Desempenho nas realizações de consultas *Near*: o PostgreSQL se comportou melhor para pequenos volumes de dados, já o MongoDB demonstra resultados melhores a medida que o conjunto de dados aumenta.

Por sua vez, o trabalho proposto por Joishi e Sureka (2016) realiza experimentos de comparação de desempenho entre os SGBDs MySQL e Neo4j utilizando os algoritmos de mineração de dados *Similar-Task* e *Sub-Contract* (HAN; KAMBER; PEI, 2011). O conjunto de dados é composto por 466.737 registros, contendo informações sobre a Biblioteca de Infraestrutura de Tecnologia da Informação (ITIL) da *Robobank Information*

and Communication and Technology (ICT) obtido a teoria de grafos possui uma ampla gama de aplicações na computação BPI (2023). Os seguintes resultados foram obtidos para cada algoritmo de mineração:

- *Similar-Task*: Por envolver tarefas com travessia de relacionamentos seguida de computação, o MySQL obteve um tempo de execução superior quando comparado ao Neo4j.
- *Sub-Contract*: O Neo4j obteve um desempenho 7 vezes melhor que o MySQL.

Em termos gerais, o Neo4j obteve um desempenho melhor, devido ao tempo para carga de grandes conjuntos de dados e para realizar travessias superiores, melhor desempenho de gravação conforme o volume de dados aumenta. No entanto, o espaço ocupado pelos elementos do grafo no Neo4j é mais de 30 vezes maior em comparação com o MySQL.

Por último, o trabalho realizado por Miler, MEDAK e Odobašić (2013) visa comparar o desempenho entre os SGBDs Neo4j e PostgreSQL por meio da implementação do algoritmo de caminho mais curto Dijkstra, utilizando conjunto de dados de estradas da Áustria, baseado no OpenStreetMap (2023). Nos experimentos, o Neo4j obteve um resultado superior ao realizar o cálculo do caminho mais curto, mas com um enorme consumo recursos operacionais.

Este trabalho de conclusão de curso apresenta algumas semelhanças e diferenças com os trabalhos citados acima, sendo mais próximo do trabalho proposto por Homrich e Mergen (2018), visto o objetivo deste com o TCC descrito aqui. As diferenças se encontram na escolha do representante do SGBD relacional, no domínio de dados e nos critérios de comparação.

4 Método de Trabalho

O método a ser detalhado a seguir relata como foi o processo de criação da aplicação para realizar a comparação de desempenho entre os SGBDs e como a mesma funciona, além de descrever a forma como o conjunto de dados foi definido e modelado.

4.1 Definição do Conjunto de Dados

O conjunto de dados é definido como uma rede de citações de patentes, sendo os identificadores dos nós relacionados obtidos por meio da base de citação de patentes dos Estados Unidos, por meio da biblioteca de análise de rede e mineração de grafos de uso geral SNAP (2023). Esta base de dados abrange 37 anos e inclui todas as patentes de serviços públicos concedidas durante esse período, totalizando 3.923.922 patentes. Já o grafo de citações inclui todas as citações feitas por patentes concedidas entre 1975 e 1999, totalizando 16.522.438 citações. Este conjunto de dados foi escolhido devido a quantidade de relacionamento e sua aplicabilidade no mundo real.

Para este trabalho, foi considerado que uma patente pode citar uma ou mais patentes como referências relevantes para a invenção reivindicada, criando assim um histórico de patentes relacionadas a uma determinada área. No mundo real, este grafo pode ser utilizado para obter informações sobre o estado da arte em um campo específico e na análise de um pedido de patente para avaliar a originalidade da invenção reivindicada.

4.2 Modelagem

No PostgreSQL, o grafo foi representado pelas tabelas *patent*, referindo-se a entidade patente, e *citation*, sendo esta referente ao relacionamento m:n entre as patentes. As Figuras 4 e 5 demonstram como o grafo foi modelado para o modelo relacional, usando, respectivamente o diagrama Entidade-Relacionamento e o diagrama Modelo-Relacional. A tabela *patent* é composta pelos seguintes atributos:

- patent_id: identificador da patente;
- author: nome do autor da patente;
- classification: classificação da patente;
- registered_at: data de registro da patente.

Já a tabela *citation* é composta pelos atributos from_id e to_id, que, respectivamente, representam o identificador da patente citadora e o identificador da patente citada.

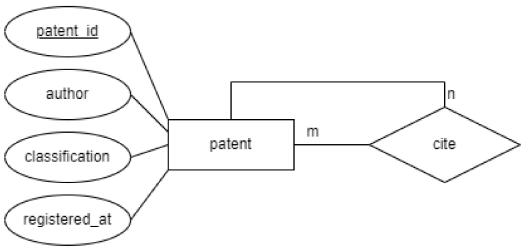

Figura 4 – Diagrama Entidade-Relacionamento de citações de patentes.

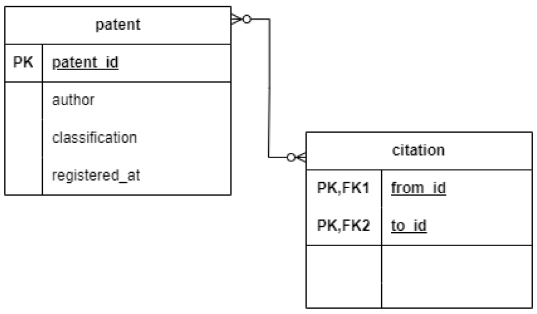

Figura 5 – Diagrama do Modelo-Relacional de citações de patentes na notação *Crow's Feet*.

Já no Neo4j, apesar da ausência de esquema, os nós seguem a mesma estrutura e nome da tabela *patent* do PostgreSQL, enquanto que os relacionamentos foram nomeados como CITE. A Figura 6 visa demonstrar o modelo Entidade-Relaciomento mapeado para o Neo4j de acordo com a modelagem proposta por Jr e Cura (2020).

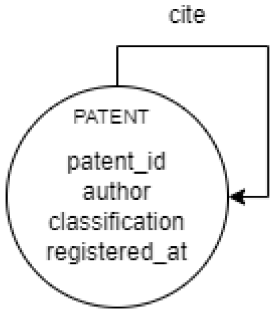

Figura 6 – Mapeamento do modelo Entidade-Relacionamento para a estrutura de grafo de citações de patentes.

4.3 Aplicação para Comparação de Desempenho

Para a realização dos testes, foi desenvolvida uma aplicação utilizando a linguagem Python (2023), o conector do Neo4j oficial para Python Driver (2023) e o conector para o PostgreSQL Psycopg (2023) para conexão com os SGBDs e a biblioteca faker PyPI (2023) para geração de dados falsos. O uso desta aplicação visa automatizar os testes, padronizar o acesso aos dois SGBDs e tornar os testes simétricos. Para a realização dos testes, a aplicação executa algumas funções, sendo as mesmas citadas nas subseções abaixo. O diagrama representando a aplicação pode ser observado na Figura 7.

O repositório do mesmo pode ser encontrado no Github <https://github.com/JoaoVictorfss/UFU.TCC2.ComplexDataAccess>.

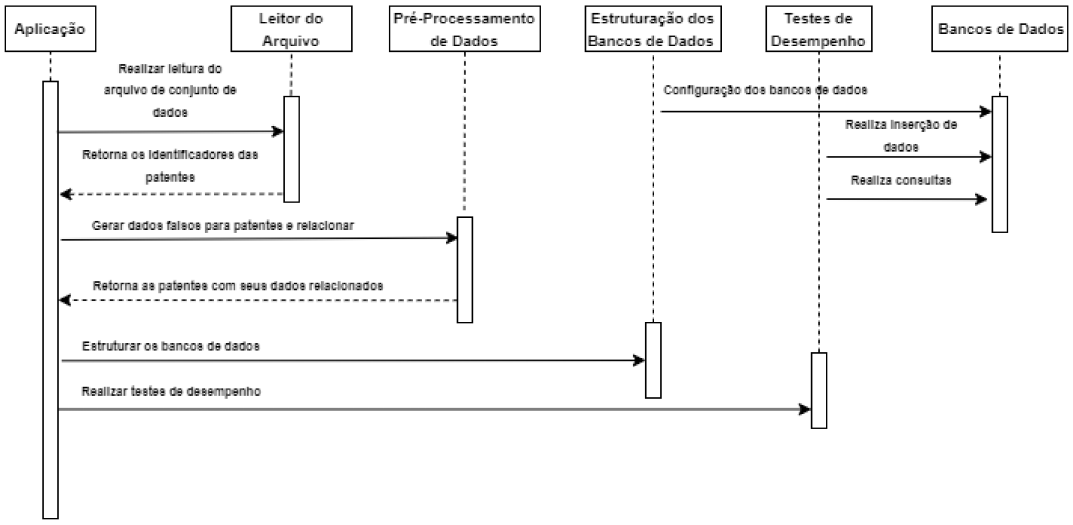

Figura 7 – Aplicação: diagrama de sequência.

4.3.1 Configuração/Estruturação dos Bancos de Dados

Com base nos parâmetros fornecidos no arquivo de configuração, a aplicação cria e configura a estrutura para cada banco de dados, desde tabelas, relacionamentos, índices até restrições com base em alguns scripts. Para o PostgreSQL, toda a estrutura é criada tendo como base os scripts abaixo:

```
#criação da tabela patent
CREATE TABLE IF NOT EXISTS patent (
    patent_id INT PRIMARY KEY NOT NULL UNIQUE,
    author VARCHAR (60) NOT NULL,
    classification VARCHAR (50) NOT NULL,
    registered_at TIMESTAMP NOT NULL)
```

```
#criação da tabela citation
CREATE TABLE IF NOT EXISTS citation (
    id INT  SERIAL PRIMARY KEY,
    from_id INT NOT NULL,
    to_id INT NOT NULL,
    CONSTRAINT fk_from_patent_id FOREIGN KEY (from_id)
    REFERENCES patent (patent_id),
    CONSTRAINT fk_to_patent_id FOREIGN KEY (to_id)
    REFERENCES patent (patent_id))

#criação do índice para o campo patent_id da tabela patente
CREATE INDEX IF NOT EXISTS idx_patent_id
on patent USING btree  (patent_id)

#criação do índice para o campo autor da tabela patente
CREATE INDEX IF NOT EXISTS idx_patent_author
on patent (author)

#criação do índice para o campo classification da tabela patente
CREATE INDEX IF NOT EXISTS idx_patent_classification
on patent (classification)

#criação do índice para o campo registered_at da tabela patente
CREATE INDEX IF NOT EXISTS idx_patent_registered
on patent (registered_at)
```

Já para o Neo4j, não há necessidade de criação dos nós e relacionamentos, pois os mesmos são criados na inserção de um novo registro, mas os seguintes scripts são executados para criação de índices e restrições:

```
#criação do índice para patent_id
CREATE INDEX IF NOT EXISTS
FOR  (p:patent) ON  (p.patent_id)

#criação do índice para o campo autor
CREATE INDEX IF NOT EXISTS
FOR  (p:patent) ON  (p.author)

#criação do índice para o campo classification
```

```
CREATE INDEX IF NOT EXISTS
FOR  (p:patent) ON  (p.classification)

#criação do índice para o campo registered_at
CREATE INDEX IF NOT EXISTS
FOR  (p:patent) ON  (p.registered_at)
```

4.3.2 Pré-processamento de Dados

Após a criação e a configuração da estrutura de cada banco de dados, a aplicação realiza o pré-processamento dos dados. De acordo com a quantidade de autores e classificações de patentes especificados no arquivo de configuração, a aplicação gera uma lista de dados falsos para nomes de autores e classificações de patentes para serem usados em consultas com filtros utilizando a classe implementada *FakerUtilis*, conforme exemplicado pelo código abaixo e a Figura 8.

```
#Method to handle data pre processing
def dataPreProcessing(settings):
  authors = FakerUtils
    .generateNames(settings.fake_authors_total)
  classifications = FakerUtils
    .generateUsPatentClassifications(settings.fake_classifications_total)
```

Logo após, é realizada uma leitura do arquivo contendo os relacionamentos entre as patentes (citações), obtido por meio da biblioteca de análise de rede e mineração de grafos de uso geral SNAP (2023). Cada linha deste arquivo contém respectivamente o identificador da patente citadora e o identificador da patente citada. A aplicação itera sobre essas linhas, onde para cada identificador de patente, é gerada uma tupla contendo o identificador da patente, o nome da autor e a classificação da patente, sendo estes dois últimos valores obtidos de forma aleatória das listas de dados falsos geradas anteriormente, e a data de registro, sendo um valor gerado de forma aleatória tendo como base valores especificados no arquivo de configuração, conforme ilustrado pela Figura 9.

Estas tuplas são inseridas em uma outra tupla, sendo esta referente a relação de citação entre as duas e por fim, esta é adicionada em uma lista. No final da iteração, é retornada uma lista de tuplas, onde para cada posição desta, existe os dados da patente citadora e os dados da patente citada, indicando a relação entre as mesmas, conforme mostrado pela Figura 10.

```python
#Fake's Data Generator
class FakerUtils:
    #Generates unique first names
    @staticmethod
    def generateNames(count):
        names = set()
        faker = Faker()
        while len(names) < count:
           name = faker.name()
           if len(name) <= 60:
            names.add(name)
        return list(names)

    #Generates US patent's classifications
    @staticmethod
    def generateUsPatentClassifications(count):
        classifications = []
        for _ in range(count):
            section = chr(random.randint(65, 72))
            subclass = random.randint(1, 99)
            main_group = random.randint(1, 9)
            subgroup = random.randint(1, 99)
            classification = f"{section}{subclass:02d}{main_group:01d}{subgroup:02d}"
            classifications.append(classification)
        return classifications
```

Figura 8 – Classe *FakerUtilis*: funções de geração de dados falsos.

```python
patentIdentifiers = FileHandler.retrieveData(settings.dataset_file_path, settings.data_max)

def processBatch(start_index, end_index):
  ...
  for i in range(start_index, end_index):
    ids = patentIdentifiers[i][0].split("\t")
    fromNodeData = None
    toNodeData = None

    with lock:
      if ids[0] not in mappedIds:
        fromNodeRegistrationDate = FakerUtils
          .generateDateTime(settings.dataset_start_date, settings.dataset_end_date)
        fromNodeData = (ids[0], authors[getIndex(len(authors))],
          classifications[getIndex(len(classifications))], fromNodeRegistrationDate, ids[1])
        mappedIds[ids[0]] = fromNodeData
      else:
        fromNodeData = (mappedIds[ids[0]][0], mappedIds[ids[0]][1],
          mappedIds[ids[0]][2], mappedIds[ids[0]][3], ids[1])

    with lock:
      if ids[1] not in mappedIds:
        toNodeRegistrationDate = FakerUtils
        .generateDateTime((fromNodeData[3] + timedelta(days=1)).strftime("%Y-%m-%d %H:%M:%S"),
                          settings.dataset_end_date)
        toNodeData = (ids[1], authors[getIndex(len(authors))], classifications[getIndex(len(classifications))],
        toNodeRegistrationDate, None)
        mappedIds[ids[1]] = toNodeData
      else:
        toNodeData = mappedIds[ids[1]]

    if fromNodeData is not None and toNodeData is not None:
      batchRecords.append((fromNodeData, toNodeData))
```

Figura 9 – Representação do código para geração dos dados das patentes.

4.3.2.1 Execução dos Testes de Comparação de Desempenho

Com as estruturas dos bancos de dados criadas e configuradas e os dados pré-processados, a aplicação executa oito testes para cada SGBD para comparação de desempenho baseada no tempo de execução. Os teste são divididos nos seguintes tipos:

- Teste de carga de dados com validação de existência: este teste visa comparar o tempo de inserção de uma patente e do relacionamento da mesma, realizando uma validação para não inserir patentes e relacionamentos duplicados, entre os dois SGBDs;

```
#Method to handle data pre processing
def dataPreProcessing(settings):
  ...
  def processBatch(start_index, end_index):
    batchRecords = []

    for i in range(start_index, end_index):
      ...
      if fromNodeData is not None and toNodeData is not None:
        batchRecords.append((fromNodeData, toNodeData))

    with lock:
      records.extend(batchRecords)

  with ThreadPoolExecutor() as executor:
      futures = []
      for i in range(0, settings.data_max, settings.fake_batchSize):
        start_index = i
        end_index = min(i + settings.fake_batchSize, settings.data_max)
        future = executor.submit(processBatch, start_index, end_index)
        futures.append(future)

      for future in futures:
        future.result()

  return records
```

Figura 10 – Representação do código para geração dos dados dos relaciomentos das patentes.

- Testes com consultas simples: são executados dois testes para cada SGBD com base nos índices das patentes com o objetivo de comparar o tempo de execução das consultas entre os dois SGBDs. A consulta realizada pelo primeiro teste visa retornar as patentes de um autor e a consulta realizada pelo segundo teste visa retornar uma patente com base no id;

- Testes com consultas com travessias: são executados dois testes que realizam consultas com travessias para cada SGBD, com intuito de comparar o tempo de execução. O primeiro teste realiza uma consulta que retorna todas as patentes que citam patentes de um autor específico de acordo com a data de registro. Já o segundo teste realiza uma consulta para encontrar caminhos de citação tripla, visando retornar todas as patentes no cenário em que uma primeira patente cita uma segunda, esta citada por sua vez é citada por uma terceira patente, e esta terceira cita outras.

- Testes com consultas com agregação: são executados dois testes para cada SGBD que realizam consultas com agregação, com objetivo de comparar o tempo de execução de cada um entre os SGBDs. O primeiro teste realiza uma consulta para retornar a quantidade de patentes por classificação, já o segundo realiza uma consulta para retornar a quantidade de citações realizadas e recebidas das 1000 patentes mais recentes, além das porcentagens de citações realizadas e recebidas referente ao total;

- Teste com consulta com correspondência de padrão: este teste visa comparar o

tempo de execução da consulta que retorna todas as patentes que citam patentes de mesma classificação entre os dois SGBDs.

Para a execução dos testes citados acima, os seguintes scripts são utilizados no Neo4j:

- Inserção de patentes e relacionamentos (CITE) se não existirem:

```
#Desdobra a variável rows, que contém os dados das patentes
UNWIND $rows AS row
#Se existir o nó, retorna, caso contrário cria um novo nó
MERGE  (p:patent {patent_id: row.patentId})
#Se o nó foi criado, adiciona as propriedades do mesmo
ON CREATE SET p += {
  author: row.author,
  classification: row.classification,
  registered_at: row.registeredAt
}
WITH p, row
#Se existir o nó da patente citada com base no
#identificador(row.toNodeId), cria o relacionamento
WHERE row.toNodeId IS NOT NULL
MATCH  (relatedP:patent {patent_id: row.toNodeId})
MERGE  (p)-[:CITE]-> (relatedP)
RETURN count (*) as total
```

- Consulta para obter as patentes de um autor:

```
#Recupera os dados de todas as patentes de
#um autor pelo nome
MATCH  (p:patent {author: $author}) RETURN p
```

- Consulta para obter uma patente com base no id:

```
#Recupera os dados de uma patente pelo identificador
MATCH  (p:patent {patent_id: $patentId}) RETURN p
```

- Consulta para obter as patentes que citam patentes de um autor:

```
#Recupera os dados de todas as patentes que citam
#patentes de um autor
MATCH (author:patent {author: $author})
<-[:CITE]- (citingP:patent)
WHERE citingP.registered_at >= $registration_date
RETURN citingP
```

- Consulta para encontrar caminhos de citação tripla:

```
#Recupera todas as patentes onde uma primeira
#patente cita uma segunda, esta citada por sua vez
#é citada por uma terceira patente, e esta terceira
#cita outras
 MATCH  (p:patent)-[:CITE]-> (relatedP:patent)
 <-[:CITE]- (coAuthor:patent)-[:CITE]-> (otherP:patent)
 RETURN p.patent_id AS patentId,
 p.author AS author,
 collect (DISTINCT relatedP.patent_id) AS relatedPatents,
 collect (DISTINCT coAuthor.author) AS coAuthors,
 collect (DISTINCT otherP.patent_id) AS otherPatents
```

- Consulta para obter a quantidade de patentes por classificação:

```
#Retorna a quantidade de patentes por classificação
MATCH  (p:patent)
RETURN p.classification AS classification,
count (*) AS count
```

- Consulta para retornar a quantidade de citações realizadas e recebidas das 1000 patentes mais recentes, além das porcentagens de citações realizadas e recebidas referente ao total:

```
#Recupera estatísticas de citações das
#1000 patentes mais recentes
 MATCH  (:patent)-[:CITE]-> (relatedP:patent)
 WITH COUNT (*) AS TotalGiven
 MATCH  (relatedP:patent)-[:CITE]-> (:patent)
 WITH TotalGiven, COUNT (*) AS TotalReceived
```

```
MATCH  (p:patent)
OPTIONAL MATCH  (p)-[:CITE]-> (given:patent)
WITH p,
   COUNT (DISTINCT given) AS GivenTotal,
   TotalGiven,
   TotalReceived
OPTIONAL MATCH  (received:patent)-[:CITE]-> (p)
WITH p,
   GivenTotal,
   TotalGiven,
   TotalReceived,
   COUNT (DISTINCT received) AS ReceivedTotal
RETURN p.patent_id AS patentId,
   GivenTotal,
   ReceivedTotal,
   ROUND (
   toFloat (GivenTotal) / TotalGiven,
   2) AS GivenPercentage,
   ROUND (
   toFloat (ReceivedTotal) / TotalReceived,
   2) AS ReceivedPercentage
ORDER BY p.registered_at DESC
LIMIT 1000
```

- Consulta para retornar todas as patentes que citam patentes de mesma classificação

```
#Retorna os dados das patentes que citam patentes de mesma
#classificação, além dos dados das patentes citadas
 MATCH  (p:patent)-[:CITE]-> (relatedP:patent)
 WHERE p.classification = relatedP.classification
 RETURN p.author AS citerPatentAuthor,
 p.patent_id AS citerPatentId,
 relatedP.author AS citedPatentAuthor,
 relatedP.patent_id AS citedPatentId,
 p.classification AS classification
```

Já para o PostgreSQL os seguintes scripts são utilizados:

- Inserção de patentes e relacionamentos (citation) se não existirem:

```
#Insere o registro se não existir
INSERT INTO patent
(patent_id, author, classification, registered_at)
SELECT %s, %s, %s, %s
WHERE NOT EXISTS  (
  SELECT 1 FROM patent
  WHERE patent_id = %s
);

#Insere o relacionamento se não existir
INSERT INTO citation  (from_id, to_id)
SELECT %s, %s
WHERE NOT EXISTS  (
  SELECT 1 FROM citation
  WHERE from_id = %s AND to_id = %s
);
```

- Consulta para obter as patentes de um autor:

```
#Recupera os dados de todas as patentes de
#um autor pelo nome
SELECT * FROM patent WHERE author = %s
```

- Consulta para obter uma patente com base no id:

```
#Recupera os dados de uma patente pelo identificador
SELECT * FROM patent WHERE patent_id = %s
```

- Consulta para obter as patentes que citam patentes de um autor:

```
#Recupera os dados de todas as patentes que citam
#patentes de um autor
SELECT p.*
FROM citation c
INNER JOIN patent p
ON c.from_id = p.patent_id
WHERE c.to_id IN  (
    SELECT patent_id
```

```
    FROM patent
    WHERE author = %s
    AND registered_at >= %s
)
```

- Consulta para encontrar caminhos de citação tripla:

```
#Recupera todas as patentes onde uma primeira
#patente cita uma segunda, esta citada por sua vez
#é citada por uma terceira patente, e esta terceira
#cita outras
SELECT p1.patent_id AS patentId,
p1.author AS author,
ARRAY_AGG (DISTINCT p2.patent_id) AS relatedPatents,
ARRAY_AGG (DISTINCT coAuthor.author) AS coAuthors,
ARRAY_AGG (DISTINCT p3.patent_id) AS otherPatents
FROM patent p1
JOIN citation c1 ON p1.patent_id = c1.to_id
JOIN patent p2 ON c1.from_id = p2.patent_id
JOIN citation c2 ON p2.patent_id = c2.to_id
JOIN patent coAuthor ON c2.from_id = coAuthor.patent_id
JOIN citation c3 ON coAuthor.patent_id = c3.from_id
JOIN patent p3 ON c3.to_id = p3.patent_id
GROUP BY p1.patent_id, p1.author
```

- Consulta para obter a quantidade de patentes por classificação:

```
#Retorna a quantidade de patentes por classificação
SELECT classification, COUNT (*) AS Total FROM patent
GROUP BY classification
```

- Consulta para retornar a quantidade de citações realizadas e recebidas das 1000 patentes mais recentes, além das porcentagens de citações realizadas e recebidas referente ao total:

```
#Recupera estatísticas de citações das
#1000 patentes mais recentes
SELECT
```

```
patent_id,
p.registered_at,
COUNT (c1) AS GivenTotal,
ROUND (
100 * CAST (COUNT (c1) AS NUMERIC) /
CAST ( (SELECT COUNT (from_id) FROM citation) AS NUMERIC), 2)
AS GivenPercentage,
COUNT (c2) AS ReceivedTotal,
ROUND (
100 * CAST (COUNT (c2) AS NUMERIC) /
CAST ( (SELECT COUNT (to_id) FROM citation) AS NUMERIC), 2)
AS ReceivedPercentage
FROM patent p
LEFT JOIN citation as c1
   ON c1.from_id = p.patent_Id
LEFT JOIN citation c2
   ON c2.to_id = p.patent_Id
GROUP BY p.patent_id
ORDER BY p.registered_at DESC
LIMIT 1000
```

- Consulta para retornar todas as patentes que citam patentes de mesma classificação

```
#Retorna dados das patentes que citam patentes de mesma
#classificação, além dos dados das patentes citadas
SELECT
p1.patent_id AS citerPatentId,
p1.author AS citerPatentAuthor,
p1.classification AS citerPatentClassification,
p2.patent_id AS citedPatentId,
p2.author AS citedPatentAuthor,
p2.classification AS citedPatentClassification
FROM patent p1
JOIN citation c ON p1.patent_id = c.from_id
JOIN patent p2 ON c.to_id = p2.patent_id
WHERE p1.classification = p2.classification
```

A Figura 11 visa demonstrar o código da função principal que cria e configura os bancos de dados e realiza estes testes, de uma forma abstraída.

```
def main():
  settings = Settings('settings.yml')

  records = None

  if(settings.tests_data_load_enabled):
    Log.information("Try to generate fake data")
    records = dataPreProcessing(settings)
    Log.information("Successfully generated data")

  testsHandler = TestsHandler(settings)

  #Configure Db
  if(settings.tests_configure_db_enabled):
    testsHandler.configureDbs()

  #Executes tests
  if(settings.tests_data_load_enabled):
    testsHandler.executeDataLoadTests(records)
  if(settings.tests_traversal_enabled):
    testsHandler.executeTestsWithTraversingQueries()
  if(settings.tests_patternMatching_enabled):
    testsHandler.executeTestsWithPatternMatchingQueries()
  if(settings.tests_aggregation_enabled):
    testsHandler.executeTestsWithAggregationQueries()
  if(settings.tests_simple_enabled):
    testsHandler.executeTestsWithoutTraversingQueries()

  Log.information("Tests run successfully")

main()
```

Figura 11 – Função main: abstração da execução dos testes.

5 Resultados e Discussão

Este capítulo tem como objetivo apresentar todos os resultados dos testes realizados, bem como informar a infraestrutura utilizada para a execução dos testes de comparação de desempenho, além do método de teste utilizado e levantar discussões acerca destes resultados.

5.1 Ambiente de Testes

Os testes foram realizados por meio da aplicação desenvolvida, sendo a mesma citada na subseção 4.3, visando padronizar o acesso aos dois SGBDs e tornar os testes simétricos.

Para a execução da aplicação e realização dos testes, os processos que estavam sendo executados foram encerrados. Ademais, a aplicação foi executada sobre uma máquina dispondo do processador Intel (R) Core (TM) i5-1135G7 dispondo de 4 núcleos e frequência de operação de 2,42GHz. Além disso, a máquina possui 16GB de memória RAM com frequência de 3200 MHz. Já o sistema operacional utilizado na máquina foi o Windows 10 Enterprise, na versão 21H2 de 64 bits.

Os dados foram armazenados em nuvem, utilizando os servidores ElephantSQL <https://www.elephantsql.com/> para o PostgreSQL e o servidor Neo4j Aura DB <https://neo4j.com/cloud/platform/aura-graph-database/> para o Neo4j, devido a limitação de memória da máquina. Para o plano do ElephantSQL selecionado, há um limite de armazenamento de 20MB e este usa a versão do PostgreSQL mais recente. Já para plano do Neo4j Aura DB selecionado, há um limite de 200.000 nós e 400.000 relacionamentos e o mesmo faz o uso do Neo4j na versão 5.

5.2 Metodologia dos Testes

Devido a limitação do tamanho do armazenamento dos dados, os testes foram executados com dois conjuntos de dados de tamanhos diferentes, o primeiro denominado CD10, formado por 10.000 relacionamentos e 11.869 patentes. Já o segundo, denominado CD50, sendo formado por 50.000 citações e 57.833 patentes.

Para os testes com o conjunto de dados CD10, foram gerados 10.000 possíveis autores e 400 possíveis classificações para cada patente, além do uso de 2 threads executando simultaneamente para a inserção de dados.

Já para os testes realizados sobre o conjunto de dados CD50, foram gerados 50.000

possíveis autores e 400 possíveis classificações para cada patente, além do uso de 20 threads executando simultaneamente para a inserção de dados.

Para cada teste, a aplicação executa o teste primeiramente para o PostgreSQL e logo após para o Neo4j, via os conectores para o Neo4j e PostgreSQL para Python, conforme já citado anteriormente no capítulo 4.

Após a execução de cada teste, a aplicação salva em um arquivo csv, conforme o caminho para o diretório especificado no arquivo de configuração, o tipo do teste, o SGBD utilizado, a descrição do teste, a data de início e fim, além do tempo de execução e o resultado retornado para futura análise.

5.3 Resultados dos testes

Nas subseções abaixo, os resultados dos testes executados por categoria são mostrados e descritos para análise de comparação de desempenho entre os SGBDs PostgreSQL e Neo4j. Cada resultado é representado por uma tabela, com uma coluna para o SGBD utilizado e outra para o tempo de execução em milissegundos levando em consideração o tamanho do conjunto de dados, além de gráficos para auxiliar na visualização e análise.

5.3.1 Testes de Carga de Dados com Validação de Existência

Os resultados mostrados abaixo referem-se aos testes de carga, com a inserção de duas patentes e o relaciomento entre elas por vez. Deve se levar em consideração que para cada inserção de patente e relacionamento, é realizada uma validação para analisar se o registro já não existe. As tabelas 5 e 6 demonstram respectivamente o tempo de execução para cada SGBD levando em consideração o conjunto de dados CD10 e CD50. Já a Figura 12 mostra o gráfico de comparação de desempenho de acordo com o conjunto de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	15.312.150,449
Neo4j	3.444.942,361

Tabela 5 – Resultado do teste de carga, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	40.709.431,159
Neo4j	8.516.380,493

Tabela 6 – Resultado do teste de carga, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

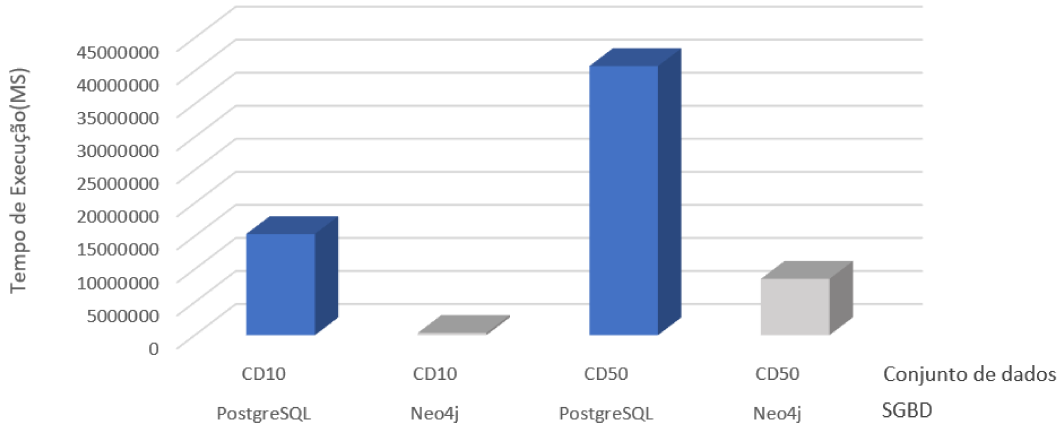

Figura 12 – Gráfico para comparação de desempenho entre os SGBDs para carga de dados de acordo com o conjunto de dados.

Com base nos resultados observados, pode-se concluir que o o Neo4j obteve um tempo de execução consideravelmente maior na inserção de registros não duplicados, aproximadamente 5 vezes mais rápido, quando comparado ao PostgreSQL, independente do tamanho do conjunto de dados. Tais resultados do Neo4j podem estar atrelados ao seu modelo de dados orientado a grafos, permitindo navegar diretamente pelas relações para validação de existência de um nó e a sua estrutura de armazenamento de propriedades. Além disso, deve-se levar em consideração a forma como o teste foi executado, onde foi necessário executar três scripts para o PostgreSQL, um para inserção da patente citada, outro para a inserção da patente citadora e por fim a inserção do registro de relacionamento entre as duas. Já para o Neo4j, foi necessário executar apenas um script para inserção de duas patentes e a relação entre elas.

5.3.2 Testes com Consultas Simples

Os resultados mostrados abaixo referem-se aos testes com consultas simples, baseadas nos índices das patentes.

As tabelas 7 e 8 demonstram respectivamente o tempo de execução da consulta de busca de uma patente pelo identificador para cada SGBD levando em consideração o conjunto de dados CD10 e CD50. A Figura 13 mostra o gráfico de comparação de desempenho de acordo com o conjunto de dados.

Por sua vez, a tabela 9 demonstra o tempo de execução da consulta de busca de todas as patentes de um autor inexistente para cada SGBD levando em consideração o conjunto de dados CD10, já a tabela 10 realiza a mesma demonstração considerando o conjunto de dados CD50. Além disso, a Figura 14 mostra o gráfico de comparação de desempenho para os diferentes conjuntos de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	464
Neo4j	2.018,827

Tabela 7 – Resultado do teste com consulta simples de busca de uma patente pelo identificador, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	465,959
Neo4j	2.392,563

Tabela 8 – Resultado do teste com consulta simples de busca de uma patente pelo identificador, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

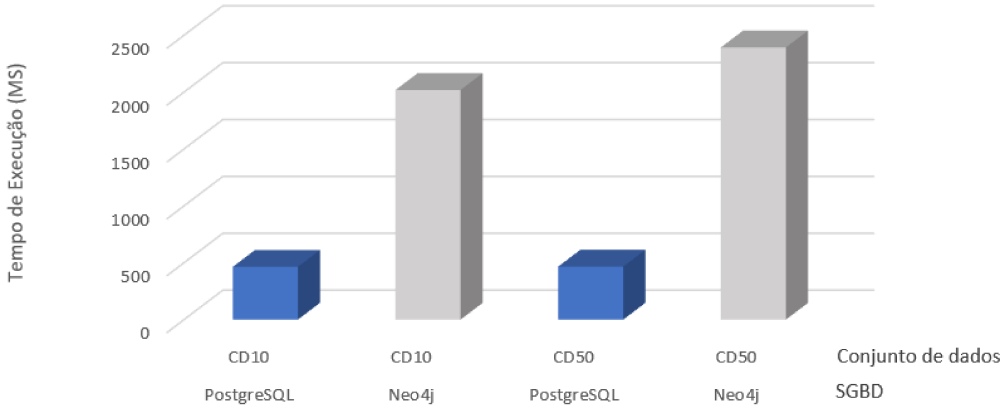

Figura 13 – Gráfico para comparação de desempenho entre os SGBDs para consulta por indentificador de acordo com o conjunto de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	233
Neo4j	329,394

Tabela 9 – Resultado do teste com consulta simples de busca de todas as patentes de um autor, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	235,863
Neo4j	362,373

Tabela 10 – Resultado do teste com consulta simples de busca de todas as patentes de um autor, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

Ao observar esses resultados, concluí-se que o PostgreSQL obteve resultados superiores nas duas consultas simples baseadas nos índices da entidade patente em relação ao Neo4j. No entanto, na consulta por patentes de um autor inexistente, o Neo4j obteve um

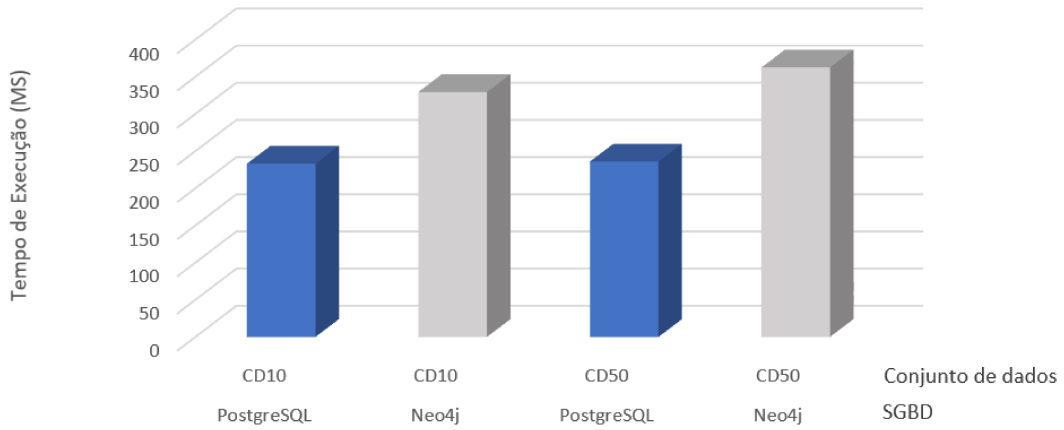

Figura 14 – Gráfico para comparação de desempenho entre os SGBDs para consulta por autor de acordo com o conjunto de dados.

tempo de execução próximo ao do PostgreSQL. Os resultados superiores observados para o PostgreSQL podem ser relacionados ao tamanho do conjunto de dados e aos índices criados, devido ao mecanismo de indexação altamente otimizado do PostgreSQL.

5.3.3 Testes com Consultas de Travessia

Os resultados mostrados abaixo referem-se aos testes com consultas de travessia.

A tabela 11 demonstra o tempo de execução da consulta de busca de patentes que citam patentes de um autor inexistente de acordo com a data de registro para cada SGBD levando em consideração o conjunto de dados CD10, já a tabela 12 realiza a mesma demonstração considerando o conjunto de dados CD50. Por outro lado, a Figura 15 mostra o gráfico de comparação de desempenho para os diferentes conjuntos de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	475,132
Neo4j	2.215,411

Tabela 11 – Resultado do teste com consulta de travessia para busca de patentes que citam patentes de um autor de acordo com a data de registro, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	478,0519
Neo4j	2.055,072

Tabela 12 – Resultado do teste com consulta de travessia para busca de patentes que citam patentes de um autor de acordo com a data de registro, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

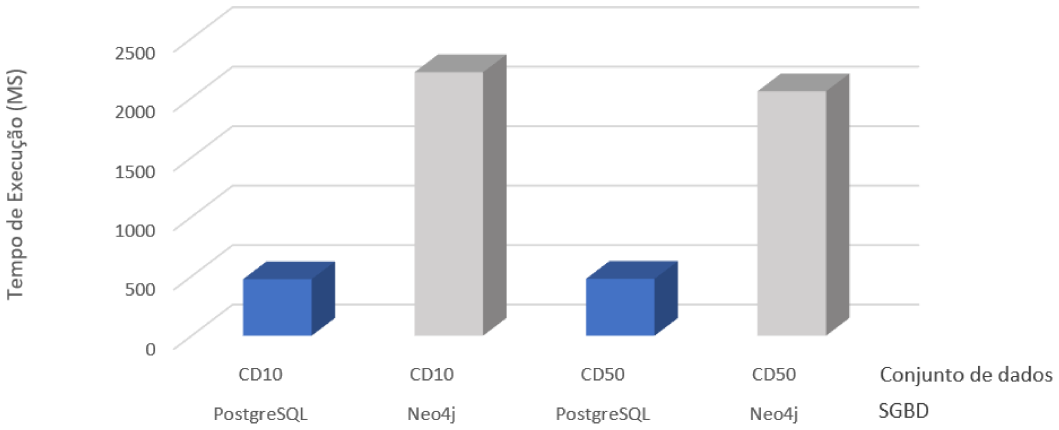

Figura 15 – Gráfico para comparação de desempenho entre os SGBDs para consulta de travessia para retornar as patentes que citam patentes de um autor de acordo com o conjunto de dados.

Em contrapartida, a tabela 13 demonstra o tempo de execução da consulta de busca de caminhos de citação tripla para cada SGBD levando em consideração o conjunto de dados CD10, já a tabela 14 realiza a mesma demonstração considerando o conjunto de dados CD50. Visto isso, a Figura 16 mostra o gráfico de comparação de desempenho para os diferentes conjuntos de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	289,3979
Neo4j	1.036,552

Tabela 13 – Resultado do teste com consulta de travessia para busca de caminhos de citação tripla, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	347,411
Neo4j	3.226,828

Tabela 14 – Resultado do teste com consulta de travessia para busca de caminhos de citação tripla, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

A partir dos resultados obtidos, é possível afirmar que o PostgreSQL obteve resultados superiores ao do Neo4j em consultas com travessia. Vale destacar que para a consulta que retorna as patentes que citam patentes de um autor específico, o PostgreSQL manteve um tempo de execução praticamente constante independente do tamanho do conjunto de dados. O Neo4j por outro lado obteve um resultado melhor, quando comparado ao desempenho do mesmo, a medida que o conjunto de dados aumentou. Além disso, a superioridade do PostgreSQL na execução de consultas com travessias pode estar

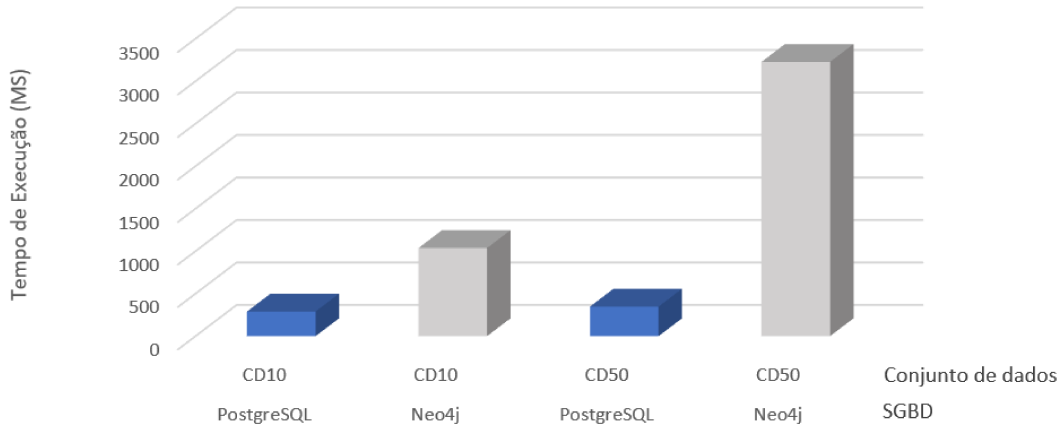

Figura 16 – Gráfico para comparação de desempenho entre os SGBDs para consulta de travessia para retornar caminhos de citação tripla de acordo com o conjunto de dados.

atrelada ao uso dos índices criados, ao tamanho e distribuilção dos dados e ao otimizador de consultas sofisticado do PostgreSQL.

5.3.4 Testes com Consultas de Agregação

Os resultados mostrados abaixo referem-se aos testes com consultas de agregação.

As tabelas 15 e 16 demonstram respectivamente o tempo de execução da consulta de agregação para busca da quantidadde de patentes por classificação para cada SGBD levando em consideração o conjunto de dados CD10 e CD50. Além disso, a Figura 17 mostra o gráfico de comparação de desempenho de acordo com o conjunto de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	236,527
Neo4j	385,841

Tabela 15 – Resultado do teste com consulta de agregação para retornar a quantidade de patentes por classificação, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	250,761
Neo4j	499,226

Tabela 16 – Resultado do teste com consulta de agregação para retornar a quantidade de patentes por classificação, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

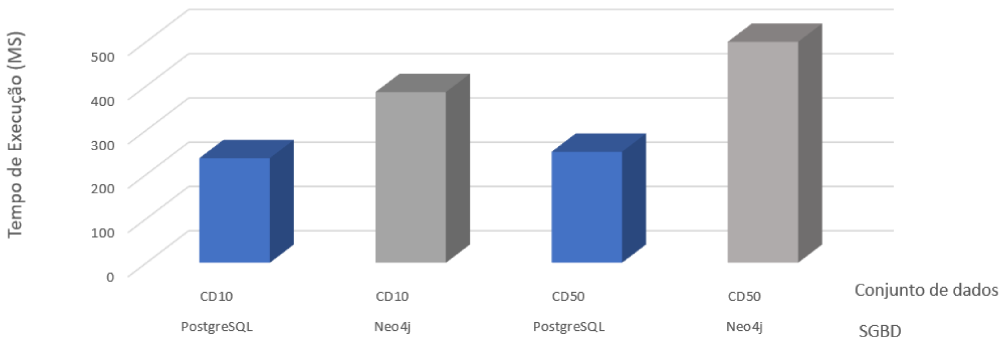

Figura 17 – Gráfico para comparação de desempenho entre os SGBDs para consulta de agregação para retornar a quantidade de patentes por classificação de acordo com o conjunto de dados.

Por sua vez, a tabela 17 demonstra o tempo de execução da consulta para retornar a quantidade de citações realizadas e recebidas das 1000 patentes mais recentes, além das porcentagens de citações realizadas e recebidas referente ao total para cada SGBD levando em consideração o conjunto de dados CD10, já a tabela 18 realiza a mesma demonstração considerando o conjunto de dados CD50. Concomitatemente, a Figura 18 mostra o gráfico de comparação de desempenho de acordo com o conjunto de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	970,81
Neo4j	2.655,495

Tabela 17 – Resultado do teste com consulta de agregação para retornar a quantidade de citações realizadas e recebidas das 1000 patentes mais recentes, além das porcentagens de citações realizadas e recebidas referente ao total, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	1.248,636
Neo4j	4.175,838

Tabela 18 – Resultado do teste com consulta de agregação para retornar a quantidade de citações realizadas e recebidas das 1000 patentes mais recentes, além das porcentagens de citações realizadas e recebidas referente ao total, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

A partir dos resultados obtidos, é possível afirmar que o PostgreSQL obteve resultados superiores ao do Neo4j na realização de consultas com agregação. O PostgreSQL é conhecido, principalmente, por seu desempenho superior em consultas com agregação devido a sua estratégia de execução, forma de gerenciamento de memória durante as operações de agregação, aos recursos de indexação, dentre outros fatores.

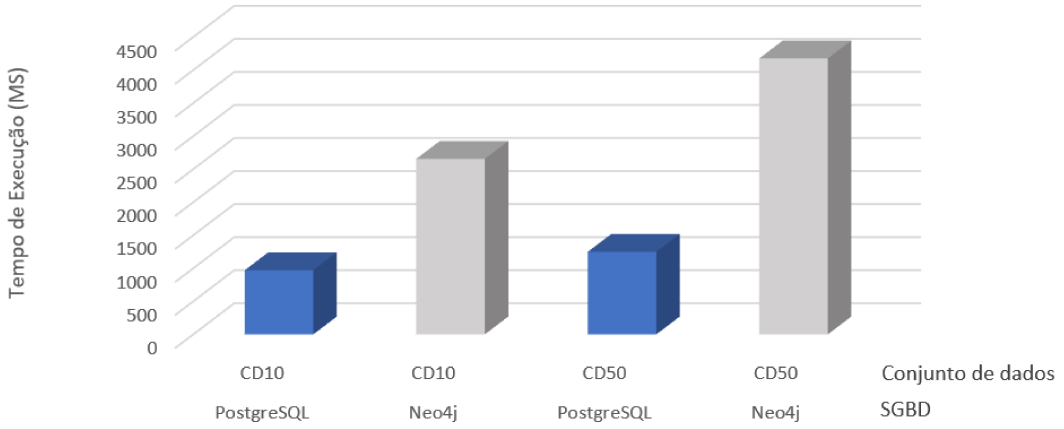

Figura 18 – Gráfico para comparação de desempenho entre os SGBDs para consulta de agregação para retornar estatísticas de citações realizadas e recebidas das 1000 patentes mais recentes de acordo com o conjunto de dados.

5.3.5 Testes com Consulta de Correspondência de Padrão

Os resultados mostrados abaixo referem-se ao teste com consulta de correspondência de padrão, para retornar todas as patentes que citam patentes de mesma classificação. A tabela 19 demonstra o tempo de execução para cada SGBD levando em consideração o conjunto de dados CD10, já a tabela 20 realiza a mesma demonstração considerando o conjunto de dados CD50. A Figura 19 visa apresentar um gráfico de comparação de desempenho de acordo com o conjunto de dados.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	462,647
Neo4j	2.285,366

Tabela 19 – Resultado do teste com consulta de correspondência de padrão, por SGBD e para 10.000 relacionamentos e 11.869 patentes.

SGBD	Tempo de Execução Total (ms)
PostgreSQL	602,869
Neo4j	2.446,551

Tabela 20 – Resultado do teste com consulta de correspondência de padrão, por SGBD e para 50.000 relacionamentos e 57.833 patentes.

Portanto, com base nos resultados obtidos por meio dos testes de correspondência de padrão, o Neo4j obteve um tempo de execução maior do que 4 vezes ao tempo de execução do PostgreSQL. Assim, o PostgreSQL manteve seu desempenho superior conforme já observado nos outros testes de recuperação de registros.

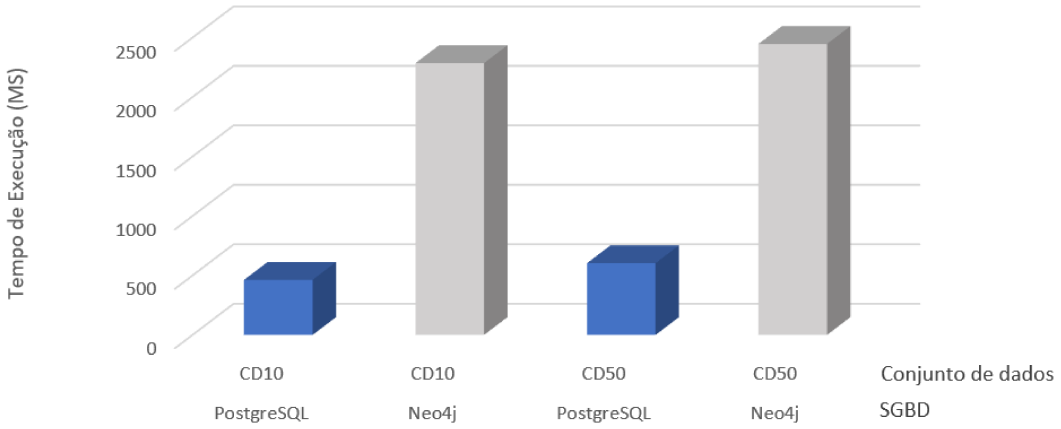

Figura 19 – Gráfico para comparação de desempenho entre os SGBDs para consulta de correspondência de padrão de acordo com o conjunto de dados.

5.4 Discussões

Percebe-se, a partir dos resultados obtidos que os bancos de dados de grafo ainda não atingiram maturidade suficiente para superarem a eficiência dos bancos de dados relacionais ao trabalhar com dados complexos, utilizando apenas suas linguagens declarativas para consultas. No entanto, para a carga de dados, o Neo4j obteve um desempenho nitidamente superior.

Entretanto, os servidores utilizados para armazenamento de dados em nuvem e os planos selecionados dos mesmos, além da máquina onde os testes são executados, a aplicação que os executam, a distribuição dos dados e o tamanho do conjunto de dados devem ser levados em consideração, visto que os resultados podem mudar de acordo com a mudança dos mesmos.

Apesar desses resultados, o Neo4j oferece uma maneira simples de gerenciamento dos dados, sendo o motivo de sua popularidade em detrimento das demais opções de sua categoria, visto que não há necessidade da criação previamente das estruturas dos nós e relacionamentos, a sintaxe clara e objetiva e a forma de representação visual dos dados intuitiva. No Neo4j, as estruturas dos nós e relacionamentos são criadas a medida que os nós e relacionamentos são inseridos. Além disso, a linguagem de consulta utilizada pelo Neo4j, Cypher, oferece um síntaxe clara e objetiva. Vale ressaltar também a representação dos dados como grafos e a interface disponibilizada pelo Neo4j, visto que deixa a análise dos dados visualmente mais fácil, conforme a demonstração da Figura 20.

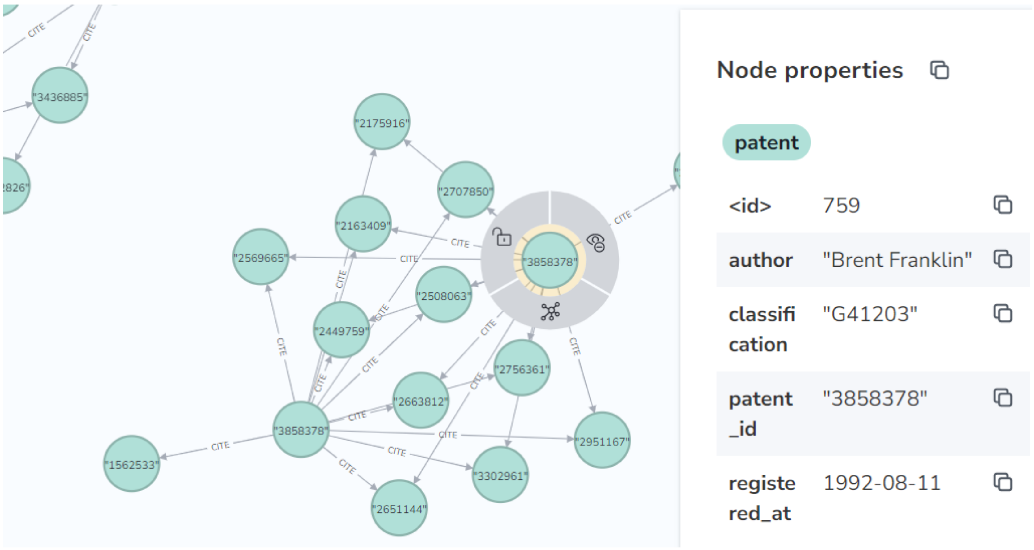

Figura 20 – Imagem do grafo obtido atráves da Interface do AuraDB (2023).

6 Conclusões e Recomendações para Trabalhos Futuros

Tendo em vista o objetivo principal deste trabalho, que consiste em analisar o desempenho entre os bancos de dados PostgreSQL e Neo4j no acesso a dados complexos, conclui-se que foram obtidos resultados que não divergem dos resultados obtidos por experimentos de outras pesquisas e trabalhos correlatos que usam de mesma abordagem.

Ademais, pode-se afirmar também, com base nestes resultados, que os bancos de dados de grafo ainda não atingiram maturidade de recursos suficiente para superarem a eficiência dos bancos de dados relacionais ao trabalhar com acesso a dados complexos utilizando apenas suas linguagens declarativas.

Para esforços futuros, recomenda-se a atualização da aplicação proposta, com adição de testes com maior quantidade de operações em grafos. Além disso, é recomendado também a adição de testes para validação da resiliência dos SGBDs no cenário com várias conexões abertas e operações sendo executadas simultaneamente.

Recomeda-se também a análise de otimização das consultas, a utilização de outras plataformas para armazenamento de dados, testes em cima de um conjunto de dados maior e a adição do parâmetro de consumo de recursos operacionais para realizar a comparação de desempenho.

Referências

ANTON, H.; RORRES, C. Álgebra linear: com aplicações. Porto Alegre: articleman, 2012. Citado na página 20.

AURADB, N. **Neo4j AuraDB**. 2023. Disponível em: <https://neo4j.com/cloud/platform/aura-graph-database/>. Citado 2 vezes nas páginas 4 e 52.

AWS. **Definição do banco de dados de grafos**. 2023. Disponível em: <https://aws.amazon.com/pt/nosql/graph/>. Citado na página 20.

BARIONI, M. C. N. Operações de consulta por similaridade em grandes bases de dados complexos. v. 1, 2006. Citado na página 10.

BPI. **Business Process Intelligence**. 2023. Disponível em: <https://www.win.tue.nl/bpi/2018/index.html>. Citado na página 27.

CHARTRAND, G.; LESNIAK, L.; ZHANG, P. Graphs digraphs. CRC Press, 2004. Citado na página 18.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to algorithms. The MIT Press, 2009. Citado na página 20.

DATE, C. J. Introdução a sistemas de bancos de dados. Gen, 2004. Citado 2 vezes nas páginas 14 e 15.

DB-ENGINES. **The most popular database management systems**. 2022. Disponível em: <https://db-engines.com/en/>. Citado 4 vezes nas páginas 10, 11, 13 e 22.

DRIVER, P. **Neo4j Python Driver**. 2023. Disponível em: <https://neo4j.com/docs/api/python-driver/current/>. Citado na página 30.

EVERITT, T.; HUTTER, M. Analytical results on the bfs vs. dfs algorithm selection problem. part i: Tree search. Springer, 2015. Citado na página 20.

FRANCIS, N.; GREEN, A.; GUAGLIARDO, P.; LIBKIN, L.; LINDAAKER, T.; MARSAULT, V.; PLANTIKOW, S.; RYDBERG, M.; SELMER, P.; TAYLOR, A. An evolving query language for property graphs. v. 1, 2018. Citado na página 22.

GUIMARãES, C. C. Fundamentos de bancos de dados: Modelagem, projeto e linguagem sql. Unicamp, New York, NY, USA, 2003. Citado na página 13.

HAN, J.; KAMBER, M.; PEI, J. Data mining: Concepts and techniques. Elsevier, 2011. Citado na página 26.

HECHT, R.; JABLONSKI, S. Nosql evaluation: A use case oriented survey. 2011. Citado na página 18.

HOMRICH Émerson P.; MERGEN, S. L. S. Comparação entre mysql e neo4j para o acesso a dados complexos usando linguagens declarativas. v. 1, 2018. Citado 2 vezes nas páginas 25 e 27.

JAMISON, C. Structured query language (sql) fundamentals. v. 1, 2003. Citado na página 22.

JOISHI, J.; SUREKA, A. Graph or relational databases: A speed comparison for process mining algorithm. v. 1, 2016. Citado na página 26.

JR, L. S. V. U.; CURA, L. M. del V. Uma proposta de mapeamento de modelo conceitual de entidade-relacionamento estendido para o modelo de dados graph nosql. v. 1, 2020. Citado na página 29.

LOPES, J. M. Um estudo comparativo entre bancos de dados considerando as abordagens relacional e orientada a grafo. 2014. Citado na página 18.

LUCCHESI, C. L. Introdução à teoria dos grafos. IMPA, New York, NY, USA, 1979. Citado na página 18.

LóSCIO, B. F.; OLIVEIRA, H. R. de; PONTES, J. C. de S. Nosql no desenvolvimento de aplicações web colaborativas. v. 1, 2011. Citado na página 17.

MACHADO, F. N. R. Banco de dados projeto e implementação. Érica, 2017. Citado na página 15.

MILER, M.; MEDAK, D.; ODOBAšIć, D. The shortest path algorithm performance comparison in graph and relational database on a transportation network. v. 1, 2013. Citado na página 27.

NEO4J. **Neo4j Graph Database**. 2023. Disponível em: <https://neo4j.com/>. Citado 3 vezes nas páginas 20, 21 e 22.

OPENSTREETMAP. **OpenStreetMap**. 2023. Disponível em: <https://www.openstreetmap.org>. Citado 2 vezes nas páginas 26 e 27.

POSTGRESQL. **PostgreSQL: The World's Most Advanced Open Source Relational Database**. 2023. Disponível em: <https://www.postgresql.org>. Citado na página 22.

PROFESSIONAL NoSQL. Wrox Press Ltd, v. 1, 2011. Citado na página 10.

PSYCOPG. **Psycopg**. 2023. Disponível em: <https://www.psycopg.org/>. Citado na página 30.

PUJA, I.; POSCIC, P.; JAKSIC, D. Overview and comparison of several elational database modelling metodologies and notations. v. 1, 2019. Citado na página 14.

PYPI, F. **Faker PyPI**. 2023. Disponível em: <https://pypi.org/project/Faker/>. Citado na página 30.

PYTHON. **Python**. 2023. Disponível em: <https://www.python.org/>. Citado na página 30.

SADALAGE, P. J.; FOWLER, M. Nosql essencial: Um guia conciso para o mundo emergente da persistência poliglota. Novatec, v. 1, 2013. Citado 2 vezes nas páginas 10 e 16.

SANTOS, J. V.; BRITO, G.; BARBOSA, G. V. Caminho mÍnimo de redes conectadasutilizando grafos. v. 1, 2019. Citado na página 20.

SNAP. **Stanford Network Analysis Project**. 2023. Disponível em: <https://snap.stanford.edu>. Citado 3 vezes nas páginas 25, 28 e 32.

SOARES, R. R. Estudo comparativo entre sistemas de bancos de dados de grafos e relacionais para a gerência de dados de proveniência em workflows científicos. v. 1, 2013. Citado na página 25.

URIARTE, V. F. Análise de desempenho de sgbd não relacionais com dados geográficos do openstreetmap. v. 1, 2018. Citado na página 26.

ZHOU, X.; ORDONEZ, C. Computing complex graph properties with sql queries. v. 1, 2019. Citado na página 13.