SQL2Neo: Uma Interface de Acesso SQL para o Neo4j

Pablo Vicente Passos[1], **Lucas Santos de Oliveira**[1], **Geomar André Schreiner**[2], **Vanessa Lago Machado**[1,3], **Ronaldo dos Santos Mello**[1]

[1] PPGCC, Departamento de Informatica e Estátística (INE)
Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

[2]Universidade Federal da Fronteira Sul (UFFS)
Chapecó – SC – Brasil

[3]Instituto Federal Sul-Rio-Grandense (IFSul), Passo Fundo - RS - Brasil

pablo.vicente@outlook.com.br,lucas.santos.mi@outlook.com,

geomarschreiner@gmail.com,vanessalagomachado@gmail.com,r.mello@ufsc.br

Abstract. *With the increasing need of several applications to continuously and quickly manipulate large data volumes, NoSQL Database (DB) raises as a good choice. In a scenario where applications intend to migrate relational data to NoSQL DBs, developers are challenged to deal with different access languages or methods, as well as different DB technologies. Given this context, we propose a solution for mapping the traditional SQL access interface to graph-oriented NoSQL DBs access interface with focus on Neo4j DB system. Different from related work, our proposal accomplishes the mapping of the main SQL DDL/DML instructions to the Cypher language of Neo4j. We also present mapping rules between relational and graph-oriented data models. Experiments demonstrate that our solution is viable and allows data manipulation in Neo4j without the need to modify the SQL access interface to Cypher by the application.*

Resumo. *Com a necessidade proeminente de diversas aplicações para manipular grandes volumes de dados de forma rápida e constante, os Bancos de Dados (BDs) NoSQL vêm se consolidando como uma escolha. Em um cenário no qual aplicações desejam migrar dados relacionais para BDs NoSQL, desenvolvedores são desafiados a lidar com diferentes linguagens ou métodos de acesso entre diferentes tecnologias de BD. Dado este contexto, este trabalho propõe uma solução para o mapeamento da interface de acesso SQL para BDs NoSQL orientados a grafos com foco no sistema de gerência de BD Neo4j. Diferente dos trabalhos relacionados, esta proposta realiza o mapeamento das principais instruções SQL DDL/DML para a linguagem Cypher do Neo4j, apresentando também regras de mapeamento entre os modelos de dados relacional e orientado a grafo. Experimentos demonstram que a solução proposta é viável e permite a manipulação de dados no Neo4j sem a necessidade de modificação da interface de acesso SQL para Cypher por parte da aplicação.*

1. Introdução

Ao longo dos anos, os bancos de dados relacionais (BDRs) se tornaram os mais populares sistemas de gerenciamento de BD (SGBDs), pois atendem as necessidades da

maior parte das aplicações, entregando simplicidade e confiabilidade na manipulação dos dados. No entanto, para grandes volumes de dados heterogêneos eles acabam não sendo compatíveis, pois precisam gerenciar e manter a consistência de dados, além do controle rígido do esquema lógico, comprometendo o desempenho [Abadi 2009]. Para suprir essas necessidades, foram introduzidos novos modelos de dados e SGBDs denominados *NoSQL* [Sadalage and Fowler 2012]. A arquitetura destes SGBDs valoriza a alta disponibilidade, distributividade e escalabilidade atrelados à capacidade de processamento de grandes volumes de dados com esquemas flexíveis. Os BDs NoSQL são classificados de acordo com o seu modelo de dados. Os 4 principais modelos de dados são o chave-valor, orientado a colunas, orientado a documentos e orientado a grafos. Este último modelo representa dados através de nodos e relacionamentos entre nodos. BDs NoSQL orientados a grafos são também denonimados de BDs de *grafos de propriedades*, uma vez que tanto nodos quanto arestas podem ter atributos.

O foco deste trabalho é a definição e a manipulação de dados relacionais mantidos em BDs NoSQL orientados a grafos de forma transparente. O mapeamento do modelo relacional para o modelo de grafos é uma alternativa para otimizar principalmente operações de manipulação de dados que acessam diversos relacionamentos entre estes dados. Isso porque, em um grafo, os relacionamentos são definidos de forma mais natural e são acessados de maneira mais eficiente. Nodos podem guardar dados indexados não apenas de suas propriedades, mas também dos seus relacionamentos (arestas) em diferentes direções, criando assim uma estrutura mais orgânica [Stoica et al. 2019]. Para tanto, este trabalho propõe uma interface de acesso SQL para o SGBD *Neo4j* denominada *SQL2Neo*. Neo4j é o principal SGBD NoSQL orientado a grafos na indústria. Ele possui um modelo de dados robusto para representar grafos de propriedades e uma linguagem de acesso declarativa e igualmente robusta chamada *Cypher*.

Trabalhos relacionados exploram principalmente o mapeamento de esquemas e a migração de dados entre os modelos relacional e orientado a grafos [Virgilio et al. 2013, Orel et al. 2017, Megid et al. 2018, Alotaibi and Pardede 2019, Gueidi et al. 2021]. Alguns poucos trabalham mencionam a tradução de SQL para Cypher [Li et al. 2021, Gueidi et al. 2021, Boudaoud et al. 2022], porém o foco está apenas na instrução `SELECT` e o processo de tradução não é detalhado. Neste sentido, *SQL2Neo* é uma proposta pioneira no sentido de oferecer suporte à tradução das principais instruções DDL e DML da SQL além de descrever regras e o processo de tradução associados. Experimentos atestam a viabilidade da proposta em termos de tempo de processamento.

Este artigo está organizado conforme segue. A Seção 2 introduz Neo4j e Cypher. A Seção 3 discute o estado da arte e a Seção 4 é dedicada à *SQL2Neo*. Por fim, as Seções 5 e 6 apresentam os experimentos e a conclusão deste trabalho, respectivamente.

2. Neo4j e Cypher

O SGBD *Neo4j* é o principal representante dos BDs NoSQL orientados a grafos[1]. Um item de dado é representado por um nodo que tem relações com outros itens através de arestas. Tanto nodos quanto arestas podem ter propriedades, como mostra a Figura 1, onde são mostrados dois nodos (*Alberto* e *Beatriz*) e duas arestas conectando-os (*KNOWS*). Ambos os nodos possuem uma propriedade *nome* e as arestas têm uma propriedade *since*.

[1]https://neo4j.com/

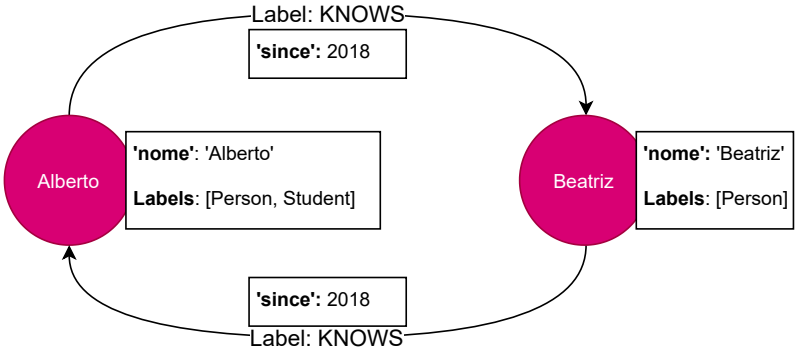

Figura 1. Exemplo de um BD no Neo4j

Nodos e arestas podem ter também rótulos (ou *labels*), que são usados para associar tipos. Arestas devem ter um rótulo (p.ex.: *KNOWS*), enquanto nodos podem ter mais de um rótulo (p.ex: *Person*), permitindo que um nodo instancie mais de um tipo.

Cypher é uma linguagem declarativa, criada inicialmente para execução de consultas no SGBD Neo4j[2]. O código a seguir exemplifica uma consulta em Cypher que retorna nomes de pessoas que conhecem pessoas:

```
MATCH (p:Person)-[:KNOWS]->(p1:Person)
WHERE p.nome IS NOT NULL RETURN p.nome
```

A cláusula `SELECT` da SQL é representada em Cypher com a cláusula `RETURN`. A cláusula `MATCH` define padrões de busca no grafo e a cláusula `WHERE` é similar à SQL.

3. Trabalhos Relacionados

Diversas propostas na literatura lidam com a problemática de mapeamento entre BDRs e BDs NoSQL orientados a grafos. Entretanto, muitas delas se dedicam ao mapeamento entre o modelo relacional e o modelo orientado a grafos com diferentes focos: *(i)* processo de mapeamento [Virgilio et al. 2013, Sayeb et al. 2017, Megid et al. 2018]; (ii) regras de mapeamento [Orel et al. 2017, Stoica et al. 2019, Alotaibi and Pardede 2019, Boudaoud et al. 2022]; e (iii) migração de dados entre SGBDs [Gueidi et al. 2021]. Há um consenso, na grande maioria destes trabalhos, quanto às regras de mapeamento do modelo relacional para o modelo de grafos de propriedades, conforme ilustra a Tabela 1.

Tabela 1. Regras de mapeamento relacional-grafo

Relacional	tabela	tupla	atributo	chave primária	chave estrangeira
Grafo	rótulo	nodo	propriedade	propriedade "ID"	aresta

Em alguns casos, restrições de integridade são definidas no BD orientado a grafos para garantir a unicidade das chaves primárias [Gueidi et al. 2021]. Outro diferencial em alguns trabalhos é a geração de arestas para tabelas representando relacionamentos muitos-para-muitos [Orel et al. 2017, Alotaibi and Pardede 2019].

A única abordagem com objetivo similar ao proposto neste trabalho é o *SQL2Cypher* [Li et al. 2021]. Entretanto, seu foco é na ferramenta desenvolvida e é

[2]https://neo4j.com/product/cypher-graph-query-language/

permitido apenas a execução de instruções `SELECT` com junções. O trabalho também não detalha regras de mapeamento entre os modelos de dados nem apresenta experimentos para avaliar a proposta de alguma forma. Outras duas propostas também ilustram superficialmente a execução de consultas SQL sobre o Neo4j [Gueidi et al. 2021, Boudaoud et al. 2022], porém com as mesmas capacidades limitadas da *SQL2Cypher*.

Assim sendo, este trabalho se diferencia do estado da arte ao propor uma interface de acesso SQL para um SGBD NoSQL orientado a grafos com as seguintes inovações: *(i)* suporte ao mapeamento das principais instruções DDL; *(ii)* suporte ao mapeamento das principais instruções DML para atualizações de dados (INSERT, UPDATE e DELETE) e; *(iii)* uma avaliação experimental que demonstra a sua viabilidade. A solução aqui proposta, denominada *SQL2Neo*, baseia-se em um processo de mapeamento que executa as instruções SQL suportadas no Neo4j. Ela é detalhada a seguir.

4. SQL2Neo

SQL2Neo é uma proposta de solução automatizada para acesso a dados mantidos no Neo4j através de instruções SQL. O Neo4j é o SGBD NoSQL orientado a grafos escolhido devido a sua popularidade na indústria. Neste caso, instruções SQL são convertidas em instruções equivalentes na linguagem Cypher.

SQL2Neo considera as regras clássicas de mapeamento relacional-grafo mostradas na Tabela 1. No caso de chaves primárias, gera-se uma propriedade cujo valor é único para todos os nodos que possuem o mesmo rótulo. Este controle é realizado através de uma restrição de integrade implementada no Neo4j. No caso de chaves estrangeiras, o(s) nome(s) do(s) atributo(s) que compõe(m) a chave é(são) representado(s) (e concatenados, no caso de chave composta) como rótulo da aresta correspondente. A Figura 2 exemplifica um mapeamento relacional-grafo para 2 tabelas relacionais.

Cabe ressaltar que *SQL2Neo* é uma extensão de uma solução destinada ao acesso SQL a BDs NoSQL orientados a agregados (chave-valor, orientado a colunas, orientado a documentos), denominada *SQLToKeyNoSQL* [Schreiner et al. 2020]. *SQLToKeyNoSQL* converte as principais instruções SQL para instruções a serem executadas em SGBDs NoSQL destino. As próximas seções detalham a arquitetura da *SQL2Neo* e o mapeamento de instruções SQL DDL e DML para Cypher com exemplos sobre os dados da Figura 2.

4.1. Arquitetura

A arquitetura da *SQL2Neo* é mostrada na Figura 3. A *interface de acesso* recebe instruções SQL *ad hoc* ou de uma aplicação baseada em BDR que migra seus dados relacionais para um BD NoSQL orientado a grafos, como o Neo4j. Essas instruções são enviadas para um *parser SQL*, que realiza análises sintáticas e semânticas com o apoio de um *dicionário*. Caso a instrução seja um `SELECT`, `UPDATE` ou `DELETE`, ele a envia para o componente de *planejamento de acesso*, que define um plano que otimiza a execução da consulta através da execução antecipada de filtros sobre tabelas específicas no caso de instruções que requeiram junções de duas ou mais tabelas. Este componente gera um plano de consulta no formato de uma árvore de execução otimizada da consulta que é encaminhada ao componente de *tradução*.

O componente de *tradução* recebe uma instrução SQL DML ou um plano de consulta e o converte para instruções em Cypher. Essas instruções são remetidas ao compo-

Personagens		
Id	Nome	FamiliaId
1	Homer Jay Simpson	1
2	Marjorie Jacqueline Simpson	1
3	Bartholomew JoJo Simpson	1
4	Lisa Marie Simpson	1
5	Margaret Evelyn Lenny Simpson	1
6	Abraham Jebediah Simpson II	1
7	Mona Penelope Simpson	1

Familias	
Id	Nome
1	Simpson family

Figura 2. Exemplo de mapeamento relacional-grafo realizado pela *SQL2Neo*

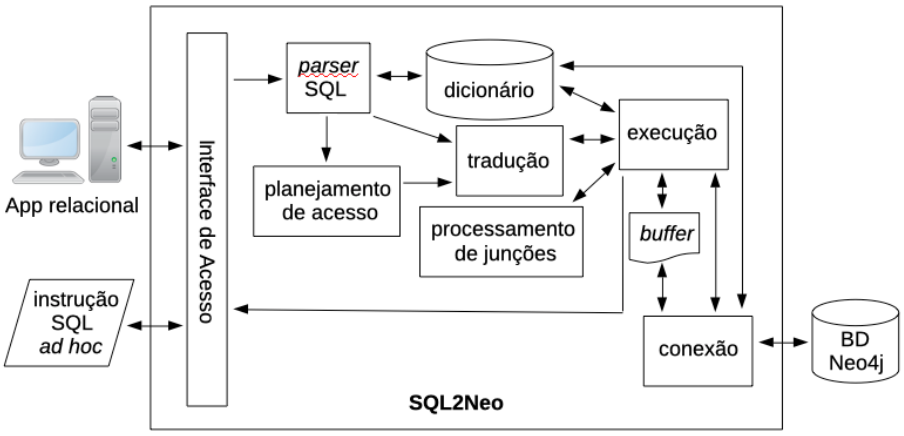

Figura 3. Arquitetura da *SQL2Neo*

nente de *execução*. Este componente central da *SQL2Neo* orquestra o envio das instruções para o componente de *conexão*, bem como o recebimento dos dados do Neo4j e posterior geração do resultado de uma consulta ou status das operações de atualização, que são então enviadas à *interface de acesso* para serem devolvidas ao usuário ou aplicação.

Caso uma consulta SQL possua junções, os dados recebidos do Neo4j são remetidos ao componente de *processamento de junções*, que executa um algoritmo de junção para produzir o resultado combinado desejado. Este processamento se faz necessário uma vez que BDs NoSQL geralmente não suportam junções. Junções são computadas pela *SQL2Neo* através de dois algoritmos: *Merge-Join* (quando todos os dados não podem ser

mantidos na memória principal) e *Hash-Join* (caso contrário). Por fim, o componente de *conexão* é responsável pela comunicação com o Neo4j. Ele submete instruções ao Neo4j, recebe dados ou status de execução de instruções e os remete ao componente de *execução*.

Dois repositórios auxiliares são também considerados na arquitetura da solução: *buffer* e o *dicionário*. O primeiro é utilizado quando o volume de dados manipulado não é comportado na memória principal. Neste caso, utiliza-se o disco para a persistência de dados. Já o dicionário mantém metadados relativos ao esquema relacional. Um registro de metadados de uma tabela relacional segue a Definição 1.

Definição 1 (Registro Dicionário). *Um registro de metadados de uma tabela t de um BDR no dicionário é uma tupla $(nome, AT, PK, FK, bd)$, onde $nome$ é o nome de t, AT é a lista de nomes de atributos de t, $PK \subseteq AT$ é o conjunto de atributos que compõem a chave primária de t, $FK = \{(A_1, tnome_1), ..., (A_n, tnome_n)\}$ é o conjunto de chaves estrangeiras de t, sendo que cada par $(A_i, tnome_i) \in FK$, $A_i \subseteq AT$, mantém o nome $tnome_i$ da tabela referenciada, e bd é o nome do BDR.*

O acesso aos metadados do dicionário é necessário para o *parsing* de instruções SQL, para a geração do esquema do resultado de consultas pelo componente de *execução*, e para o mapeamento de instruções SQL para instruções *Cypher* de acordo com as regras de mapeamento relacional-grafo.

4.2. Mapeamento SQL DDL

Instruções SQL DDL recebidas pela *SQL2Neo* manipulam dados presentes no dicionário e no Neo4j. A solução permite a execução das 3 principais instruções DDL: `CREATE TABLE`, `ALTER TABLE` e `DROP TABLE`. O mapeamento da instrução `CREATE TABLE` é dado pela Definição 2.

Definição 2 (`CREATE TABLE`). *Uma instrução CREATE TABLE tab ({$at_1 dt_1$[,...,$at_n dt_n$]} [,PRIMARY KEY (at_i,...,at_j)] [,FOREIGN KEY ({at_k,...,at_m}) REFERENCES tab_r [,...,FOREIGN KEY ({at_p,...,at_q}) REFERENCES tab_s]]) em um BDR com nome bd_1 gera um registro no dicionário com o formato $(tab, \{at_1, ..., at_n\}, \{at_i, ..., at_j\}, \{(\{at_k, ..., at_m\}, tab_r), ..., (\{at_p, ..., at_q\}, tab_s)\}, bd_1)$.*

Para cada instrução DDL um mapeamento também é definido para a linguagem Cypher para posterior execução no Neo4j. No caso da `CREATE TABLE`, uma restrição de unicidade é criada para o controle de chaves primárias conforme ilustra a Definição 3. Esta chave é definida pela propriedade sintética *NODE_KEY* gerada pela *SQL2Neo* para cada nodo no Neo4j.

Definição 3 (`CREATE TABLE`-Cypher). *Uma instrução CREATE TABLE tab (...) gera em Cypher uma restrição CREATE CONSTRAINT tab_NODE_KEY IF NOT EXISTS FOR (n:tab) REQUIRE (n.NODE_KEY) IS UNIQUE.*

No caso do `ALTER TABLE`, a adição de um atributo (cláusula `ADD COLUMN`) não gera uma alteração no Neo4j, pois seu esquema é flexível, não sendo necessária a presença de todas as propriedades em um nodo. Esse não é caso da exclusão de um atributo (cláusula `DROP COLUMN`), que requer a sua exclusão dos nodos que o possuem, bem como a exclusão de uma aresta quando o atributo atua como uma chave estrangeira, conforme mostra a Definição 4.

Definição 4 (`ALTER TABLE-DROP COLUMN`-Cypher). *Uma instrução ALTER TABLE tab ... DROP COLUMN a_1, ..., a_n ... gera em Cypher, para cada $a_i \in a_1, ..., a_n$, uma instrução MATCH (n:tab) REMOVE n.a_i caso a_1 não seja chave estrangeira, ou MATCH (n:tab)-[f1:a_i]->(f) DELETE f1, caso contrário.*

A Figura 4 exemplifica o mapeamento de uma instrução `ALTER TABLE` que remove o atributo *FamiliaId* da tabela *Personagens*. Inicialmente é feita a seleção dos nodos com rótulo *Personagens* que possuem uma aresta do tipo *FamiliaId*. A aresta que atua como chave estrangeira é então removida.

```
1 MATCH(n:Personagens)-[f1:FamiliaID]->(f)
2 DELETE f1
```

Figura 4. Exemplo de mapeamento `ALTER TABLE` para exclusão de atributo

No caso de renomeação de um atributo (cláusula `RENAME COLUMN`), propriedades não podem ser renomeadas no *Neo4j*. Neste caso, deve-se criar uma nova propriedade com o novo nome e mantendo o valor para, em seguida, excluir a propriedade antiga. Quando o atributo é uma chave estrangeira, cria-se uma aresta com o novo rótulo e exclui-se a aresta antiga, conforme rege a Definição 5. A Figura 5 exemplifica este mapeamento para a alteração de nome do atributo *Nome* da tabela *Personagens* para $Nome\_novo$.

Definição 5 (`ALTER TABLE-RENAME COLUMN`-Cypher). *Uma instrução ALTER TABLE tab ... RENAME COLUMN a_{old} TO a_{new} ... gera em Cypher uma instrução MATCH (n:tab) SET n.a_{new} = n.a_{old} REMOVE n.a_{old}, caso a_{old} não seja chave estrangeira, ou MATCH (n:tab)-[f1:a_{old}]->(f) CREATE (n)-[a_{new}]->(f) DELETE f1, caso contrário.*

```
1 MATCH(n:Personagens)
2 SET n.Nome_novo = n.Nome
3 REMOVE n.Nome
```

Figura 5. Exemplo de mapeamento `ALTER TABLE` para renomeação de atributo

Cabe ressaltar que todas estas variantes da instrução `ALTER TABLE` também modificam o registro da respectiva tabela no dicionário. Por fim, para a instrução `DROP TABLE`, a *SQL2Neo* exclui nodos que correspondem às tuplas da tabela tab excluída e suas arestas. Na sequência é feita a exclusão da restrição de unicidade para o rótulo tab, conforme descreve a Definição 6. A Figura 6 mostra a exclusão da tabela *Personagens*.

Definição 6 (`DROP TABLE`-Cypher). *Uma instrução DROP TABLE tab gera em Cypher as instruções MATCH (n:tab) OPTIONAL MATCH (n:tab)-[f1]->(f) DELETE f1,n; DROP CONSTRAINT tab_NODE_KEY IF EXISTS.*

```
1 MATCH(n:Personagens)
2 OPTIONAL MATCH(n:Personagens)-[f1]->(f)
3 DELETE f1, n;
4 DROP CONSTRAINT Personagens_NODE_KEY IF EXISTS
```

Figura 6. Exemplo de mapeamento `DROP TABLE`

4.3. Mapeamento SQL DML

SQL2Neo oferece suporte para as 4 instruções clássicas SQL de manipulação de dados: `INSERT`, `UPDATE`, `DELETE` e `SELECT`. Subconsultas não são permitidas e condições são compostas por predicados conectados por `AND` ou `OR`.

O mapeamento da instrução `INSERT` é descrito na Definição 7. Um nodo em Cypher é criado para a tupla a ser inserida e, caso a tupla possua chaves estrangeiras, arestas são geradas para os nodos referenciados, caso existam. O método auxiliar *value* obtém o valor concatenado de um conjunto de atributos. A Figura 7 exemplifica o mapeamento de uma inserção na tabela *Personagens*. Cabe salientar que, apesar da Definição 7 gerar propriedades para todos os atributos de uma tabela, atributos que atuam como chaves estrangeiras não geram propriedades e sim arestas, conforme ilustra a Figura 7. O mesmo vale para atributos que compõem a chave primária, que são substituídos pela propriedade *NODE_KEY*.

Definição 7 (`INSERT`-Cypher). *Uma instrução INSERT INTO T VALUES (v_1,...,v_m) para uma tabela T registrada no dicionário como uma tupla t_i gera em Cypher uma instrução CREATE (n:t_i.nome{NODE_KEY: value(PK), t_i.AT[1]:v_1,...,t_i.AT[m]:v_m}) e, para cada $fk_j \in t_i.FK$ gera-se instruções (fk:$t_i.fk_j.tnome_i$) WHERE fk.NODE_KEY = value($fk_j.A_i$) WITH fk WHERE fk IS NOT NULL CREATE (n)-[:value($fk_j.A_i$)]->(fk).*

```
1 //SQL
2 INSERT INTO Personagens VALUES (1,'Homer Jay Simpson',1);
3 // CYPHER
4 CREATE (n:Personagens{NODE_KEY:1,Nome:'Homer Jay Simpson'})
5 MATCH (fk:Familias) WHERE fk.NODE_KEY=1
6 WITH fk WHERE fk IS NOT NULL CREATE (n)-[:FamiliaID]->(fk)
```

Figura 7. Exemplo de mapeamento `INSERT`

O mapeamento da instrução `UPDATE` é apresentado na Definição 8. Cada nodo n_k correspondente a uma tupla a ser alterada é localizado no Neo4j (instrução `MATCH` do Cypher) e então atualizado (instrução `SET` do Cypher). Caso algum atributo fk_j que atue como chave estrangeira for alterado, então o nodo n_j referenciado pelo novo valor de fk_j é localizado e uma aresta é criada entre n_k e n_j. A aresta correspondente ao valor antigo de fk_j é também removida. A Figura 8 exemplifica o mapeamento de uma instrução `UPDATE` sobre a tabela *Personagens* que modifica um atributo que é uma chave estrangeira e um atributo que não é.

Definição 8 (`UPDATE`-Cypher). *Uma instrução UPDATE T SET a_1 = v_1,..., a_m = v_m [WHERE condição] para uma tabela T registrada no dicionário como uma tupla t_i gera em Cypher uma busca e posterior alteração de nodos através das instruções MATCH (t_i.nome) WHERE condição SET n += {a_1 = v_1,..., a_m = v_m} e, para cada $fk_j \in t_i.FK (fk_j \in \{a_1, ..., a_m\})$ gera-se instruções MATCH (fk:$t_i.fk_j.tnome_i$) WHERE $condicao_{fk_j}$ RETURN(fk) WITH n WHERE fk IS NOT NULL MATCH (n)-[fk_{old}:$t_i.fk_j.tnome_i$]->(f) DELETE fk_{old} CREATE (n)-[:$t_i.fk_j.tnome_i$]->(fk).*

O mapeamento da instrução `DELETE` é similar ao mapeamento da instrução `DROP TABLE` com exceção da exclusão da restrição de unicidade e da exclusão dos metadados da tabela. Por fim, uma consulta SQL, formada pelo bloco `SELECT-FROM-WHERE`, é convertida conforme descrito na Definição 9.

```
//SQL
UPDATE Personagens
SET FamiliaId = 2, Nome = 'Lisa Marie Simpson da Silva'
WHERE Id = 4;
//CYPHER
MATCH (n:Personagens)
WHERE n.NODE_KEY = 4
SET n +={Nome = 'Lisa Marie Simpson da Silva'}
MATCH (m:Familias)
WHERE m.NODE_KEY = 2
RETURN (m)
WITH n WHERE m IS NOT NULL
MATCH (n)-[f_old:FamiliaId]->(f)
DELETE f_old
CREATE (n)-[:FamiliaId]->(m)
```

Figura 8. Exemplo de mapeamento `UPDATE`

Definição 9 (`SELECT`-Cypher). *Uma instrução SELECT* a_1, ..., a_n *FROM* T_1 *[JOIN* T_2 *ON condição_junção_1 [...JOIN* T_m *ON condição_junção_m]] [WHERE condição] para um conjunto de tabelas* $T_1, ..., T_m$ *registradas no dicionário como um conjunto de tuplas* TU = *{*$t_1, ..., t_m$*}, respectivamente, gera em Cypher uma instrução MATCH (*n_i*:*$t_i.tnome_i$*) [WHERE condição_*t_i*] RETURN (*$n_i.a_p$*, ...,* $n_i.a_r$*) para o filtragem de dados de cada tabela* $t_1 \in TU$*, tal que condição_*$t_i \subseteq$ *condição.*

De acordo com a Definição 9, uma tabela declarada na consulta SQL gera uma instrução `MATCH-WHERE-RETURN` correspondente em Cypher. No caso de consultas com junções, mais de uma instrução Cypher é gerada, uma para cada tabela t_i a ser acessada. Cada instrução Cypher retorna um subconjunto de linhas e colunas de t_i necessário para o processamento da consulta SQL como um todo. Esses subconjuntos de dados mais as condições de junção são encaminhadas ao componente de *processamento de junções* da *SQL2Neo* que, posteriormente, gera filtros sobre arestas do grafo para combinar nodos correspondentes a tuplas de tabelas envolvidas na junção. Devido a restrições de espaço, esse processamento não é descrito formalmente aqui. A Figura 9 exemplifica o mapeamento de uma consulta SQL envolvendo as tabelas *Personagens* e *Famílias*.

```
//SQL
SELECT t1.Nome FROM Personagens t1 JOIN Familias t2
ON t1.FamiliaId=t2.Id
WHERE t1.Id<5 AND t2.Nome='Simpson family';
//CYPHER
MATCH(n1:Personagens) WHERE n1.NODE_KEY<5 RETURN (n1.Nome,n1.FamiliaId);
MATCH(n2:Familias) WHERE n2.Nome='Simpson family' RETURN (n2.NODE_KEY)
```

Figura 9. Exemplo de mapeamento `SELECT`

5. Experimentos

Esta seção descreve um conjunto de experimentos que visa analisar o *overhead* introduzido pela camada de acesso *SQL2Neo* para o processamento e execução de instruções *SQL* sobre o Neo4j. Para tanto, foi utilizado um BD composto por 17 tabelas variando de 2 a 21 atributos, a grande maioria com chaves estrangeiras[3].

[3] *https://github.com/lucasOliveira20/GerenciadorMatConstru.*

Os testes foram realizados em um computador com um processador Intel(R) Core(TM) i5-7300HQ, 2.50GHz, 24 GB de RAM 2.400 MHz, GPU Nvidia GTX 1050 4GB e Windows(R) 11 Education 22H2. Para a análise do *overhead* foram utilizadas 3 medições: *(i)* tempo medido entre o recebimento da instrução SQL e sua conversão para a instrução Cypher, incluindo consultas e atualizações no dicionário (*Camada*); *(ii)* tempo gasto para conexão com o Neo4j e envio/recebimento de dados (*Conector*) e; *(iii)* tempo gasto na execução da instrução pelo *Neo4j* (*Neo4j*).

Um gerador de dados sintéticos foi desenvolvido. Ele permite a parametrização do número de tuplas por tabela. Cinco cenários de teste foram construídos variando a quantidade de tuplas em cada tabela (1K, 3K, 5K, 7K e 10K), que produziu, respectivamente, a seguinte quantidade total de tuplas: 17K, 51K, 85K, 119K e 170K. O gerador produz réplicas de uma mesma tupla em todas as tabelas, variando apenas o Id e inserindo valores condizentes com os tipos de dados das colunas. Desta forma, consultas sempre retornam um cenário de pior caso, ou seja, todos os dados da tabela.

Para cada cenário foram executadas 20 vezes as seguintes instruções SQL nas 17 tabelas na seguinte ordem: *(i)* criação de todas as tabelas (`CREATE TABLE`); *(ii)* inserção de tuplas individuais (`INSERT`); *(iii)* busca de todas as tuplas (`SELECT`); *(iv)* atualização de todos os atributos (`UPDATE`); *(v)* exclusão de todas as tuplas (`DELETE`); *(vi)* inserção simultânea de todas as tuplas (`INSERTN`); *(vii)* renomeação de todos os atributos (`ALTER TABLE`); e *(viii)* exclusão das tabelas (`DROP TABLE`).

A Figura 10 apresenta a média dos resultados para a execução das instruções DDL. Como esperado, a instrução `CREATE TABLE` é a mais rápida pois a quantidade de tabelas a ser criada é pequena. Já os maiores tempos ficaram por conta da instrução `ALTER TABLE` pela maior quantidade de tarefas de mapeamento.

Como esperado também, o tempo é dominado pelo processamento no Neo4j. Este tempo aumenta linearmente em função da quantidade de tuplas. Para grandes quantidades de dados, o tempo do conector ficou competitivo com o tempo do Neo4j para a instrução `ALTER TABLE`. Apesar de não ser um bom resultado, argumenta-se que os testes estressaram a quantidade de modificações de atributos em todas as tabelas, sendo uma situação rara de ocorrer na prática. Por outro lado, o *overhead* introduzido pela camada em todos os testes se manteve praticamente o mesmo e bastente inferior aos tempos do conector e do Neo4j (0,079 segundos em média), ou seja, um comportamento escalável e satisfatório.

A Figura 11 apresenta a média dos resultados para a execução das instruções DML. Os maiores tempos para todas as 3 medições ficaram por conta das instruções de carga de dados (`INSERT` e `INSERTN`), com um crescimento linear. O maior crescimento, de caráter exponencial, ficou por conta da instrução `UPDATE` no Neo4j. O estresse da atualização de valor de todos os atributos em todas as tuplas contribuiu para esta situação.

As instruções `DELETE` e `SELECT` apresentaram variações bem menos significativas para a camada e o conector em comparação com o Neo4j. Mesmo assim, o conector apresentou um comportamento levemente superior ao linear para a instrução `DELETE` devido à grande quantidade de modificações a realizar no grafo. A instrução `SELECT` foi a que apresentou melhor desempenho de execução em todos os cenários avaliados. Um ponto bastante positivo a salientar é que a camada apresentou um comportamento estável,

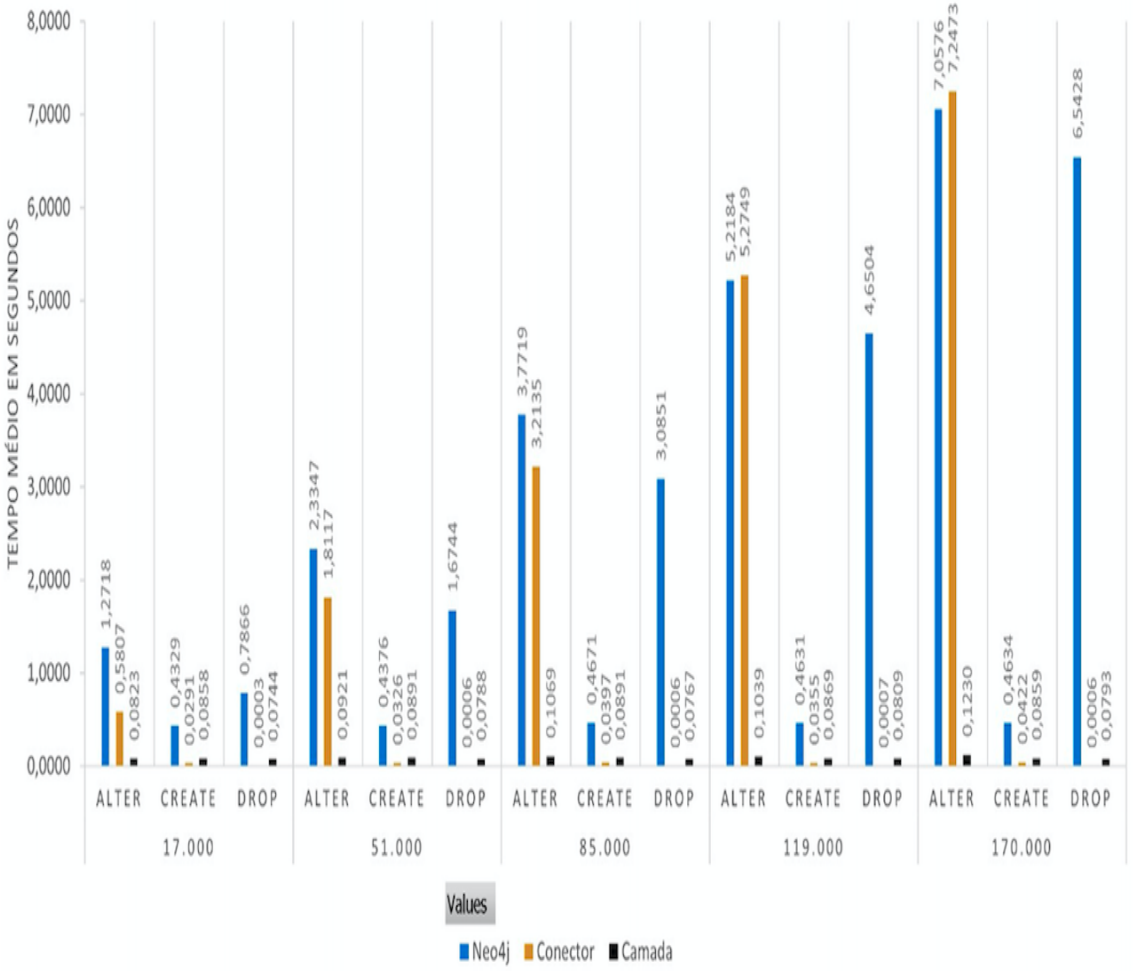

Figura 10. Resultados para as instruções DDL

com variação particamente nula em função do aumento no volume de dados processados.

6. Conclusão

Este trabalho apresenta uma solução pioneira em termos de interface de acesso SQL para o Neo4j. Poucos trabalhos relacionados lidam com essa questão, enfatizando apenas o mapeamento de consultas sem um detalhamento de regras e de um processo de mapeamento. A solução proposta neste artigo (*SQL2Neo*), por sua vez, oferece suporte às instruções DDL e DML mais significativas. O código fonte completo da *SQL2Neo* encontra-se disponível em um repositório do GitHub[4].

Outra contribuição importante é oferecer a aplicações baseadas em BDRs uma interface de acesso SQL para o Neo4j. Isso evita a curva de aprendizado para o uso de uma tecnologia NoSQL orientada a grafos ao mesmo tempo que favorece a migração de dados relacionais para um BD orientado a grafos em cenários onde manipulações e análises com foco em relacionamentos entre dados se faz necessária por ser mais eficiente.

A avaliação experimental gerou resultados que foram considerados bastante satis-

[4]*https://github.com/pablo-vicente/SQLToKeyNoSQL-Grafo.*

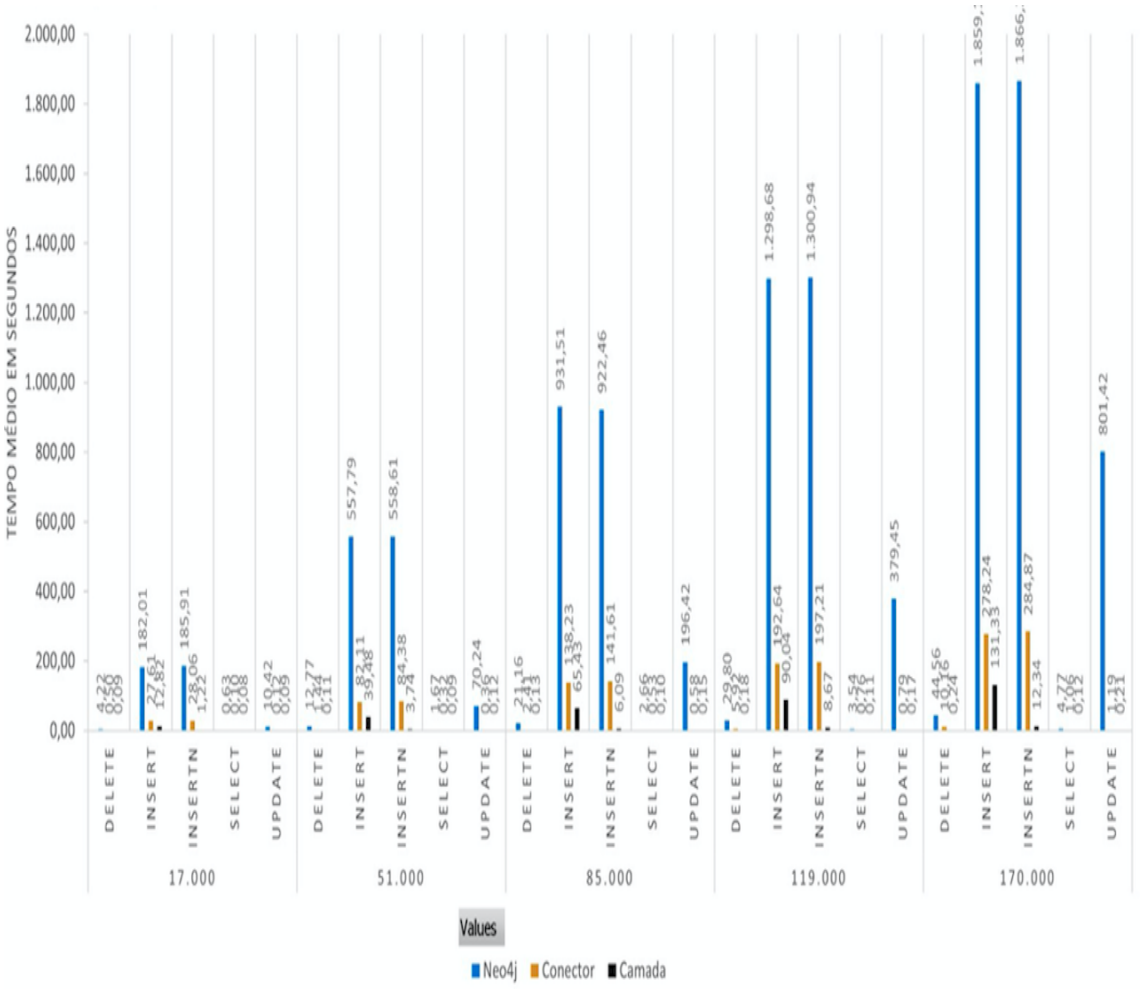

Figura 11. Resultados para as instruções DML

fatórios pois o *overhead* introduzido pela camada foi mínimo. Considera-se, assim, que a *SQL2Neo* é uma solução viável para o problema que se propõe a resolver. Trabalhos futuros incluem a comparação de desempenho da *SQL2Neo* com trabalhos relacionados, o aumento da expressividade da instrução `SELECT` para contemplar, por exemplo, subconsultas e agrupamentos, e a conexão com outros SGBDs NoSQL orientados a grafos.

Agradecimentos. Este trabalho conta com recursos financeiros do projeto *Big Data Analytics: Lançando Luz do Genes ao Cosmos* (CAPES/PRINT - Processo 88887.310782/2018-00) e do projeto *Ceos: Data Intelligence for the Society*, uma parceria de pesquisa entre a UFSC e o Ministério Público do Estado de Santa Catarina (MPSC) com suporte financeiro do MPSC.

Referências

Abadi, D. (2009). Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Eng. Bull.*, 32:3–12.

Alotaibi, O. and Pardede, E. (2019). Transformation of Schema from Relational Database (RDB) to NoSQL Databases. *Data*, 4(4):148.

Boudaoud, A. et al. (2022). Towards a Complete Direct Mapping from Relational Databases to Property Graphs. In *11th Int. Conference on Model and Data Engineering (MEDI)*, volume 13761 of *LNCS*, pages 222–235. Springer.

Gueidi, A. et al. (2021). Towards Unified Modeling for NoSQL Solution Based on Mapping Approach. In *Knowledge-Based and Intelligent Inf. & Eng. Systems (KES)*, volume 192 of *Procedia Computer Science*, pages 3637–3646. Elsevier.

Li, S. et al. (2021). SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS. In *22nd Int. Conference on Web Information Systems Engineering (WISE), Proceedings, Part II*, volume 13081 of *LNCS*, pages 510–517. Springer.

Megid, Y. A. et al. (2018). Using Functional Dependencies in Conversion of Relational Databases to Graph Databases. In *29th Int. Conf. on DB and Expert Systems Applications (DEXA), Proceedings, Part II*, volume 11030 of *LNCS*, pages 350–357. Springer.

Orel, O. et al. (2017). Property Oriented Relational-To-Graph Database Conversion. *Automatika*, 57(3):836–845.

Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education.

Sayeb, Y. et al. (2017). From Relational Database to Big Data: Converting Relational to Graph Database, MOOC Database as Example. *J. Ubiquitous Syst. Pervasive Networks*, 8(2):15–20.

Schreiner, G. A., Duarte, D., and dos Santos Mello, R. (2020). Bringing SQL Databases to Key-based NoSQL Databases: A Canonical Approach. *Computing*, 102(1):221–246.

Stoica, R. et al. (2019). On Directly Mapping Relational Databases to Property Graphs. In *13th Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW)*, volume 2369 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Virgilio, R. D. et al. (2013). Converting Relational to Graph Databases. In *1st Int. SIGMOD/PODS Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 1–6. ACM.