UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

Patrick Valencio Leguizamon

Uma Proposta para Mapeamento de Documentos JSON para Bancos de Dados de Grafos de Propriedades

Florianópolis
2022

Patrick Valencio Leguizamon

Uma Proposta para Mapeamento de Documentos JSON para Bancos de Dados de Grafos de Propriedades

Trabalho de Conclusão de Curso do Curso de Sistemas de Informação do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Sistemas de Informação.
Orientador: Prof. Ronaldo dos Santos Mello, Dr.

Florianópolis
2022

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

http://portalbu.ufsc.br/ficha

Patrick Valencio Leguizamon

Uma Proposta para Mapeamento de Documentos JSON para Bancos de Dados de Grafos de Propriedades

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Sistemas de Informação" e aprovado em sua forma final pelo Curso de Sistemas de Informação.

Florianópolis, [dia] de [mês] de [ano].

Prof. Álvaro Junio Pereira Franco, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Ronaldo dos Santos Mello, Dr.
Orientador

Prof.(a) Carina Friedrich Dorneles, Dr(a).
Avaliador(a)
Instituição Universidade Federal de Santa Catarina

Prof.(a) Patricia Della Mea Plentz, Dr(a).
Avaliador(a)
Instituição Universidade Federal de Santa Catarina

RESUMO

Nos últimos anos houveram diversos avanços no ramo da tecnologia da informação, tanto no meio acadêmico como no industrial. Isso se dá pela constante necessidade de tecnologias que buscam resolver problemas de forma simples e efetiva, muitas vezes voltadas para uso específicos. Bancos de dados *NoSQL* existem há muito tempo, porém recentemente obtiveram bastante popularidade por conta da sua flexibilidade, facilidade de escalonamento, velocidade, simplicidade, disponibilidade e outros benefícios. Suas características permitem que diferentes formatos de dados possam ser armazenados de forma eficiente para consumo, como documentos *JSON* ou *CSV*, e dados baseados em grafos. Este trabalho apresenta uma solução para o mapeamento automático de dados no formato de documento *JSON* para o modelo de grafo, considerando uma aplicação que deseja migrar dados JSON, amplamente utilizados por bancos de dados NoSQL orientados a documentos, para bancos de dados NoSQL orientados a grafos. O foco deste trabalho é a persistência destes dados no sistema de gerência de bancos de dados Neo4j, que é o principal representante de bancos de dados NoSQL orientados a grafos na indústria. Uma avaliação preliminar demonstra que a solução é escalável.

Palavras-chave: Mapeamento. *JSON*. *NoSQL*. Modelagem em grafos. *Neo4j*.

ABSTRACT

In recent years, there have been several advancements in the field of information technology, both in the academic and industrial sectors. This is driven by the constant need for technologies that aim to solve problems in a simple and effective manner, often tailored for specific purposes. NoSQL databases have existed for a long time, however, they have recently gained significant popularity due to their flexibility, scalability, speed, simplicity, availability, and other benefits. Their features enable efficient storage of various data formats, such as JSON or CSV documents, and graphic data. This work presents a solution for the automatic mapping of JSON document data to the graph model, considering an application that aims to migrate JSON data, widely used by document-oriented NoSQL databases, to graph-oriented NoSQL databases. The focus is on persisting these data in the Neo4j database management system, which is the leading graph-oriented NoSQL database in the industry. Preliminary evaluation demonstrates that the solution is scalable.

Keywords: *Mapping. JSON. NoSQL. Graph modelling. Neo4j.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BASH	Bourne-Again SHell
BSON	Binary JSON
IoT	Internet das Coisas
JSON	JavaScript Object Notation
NoSQL	Not only SQL

SUMÁRIO

1 INTRODUÇÃO

Em junho de 1970 foi publicado um artigo de Edgar F. Codd chamado "A Relational Model of Data for Large Shared Banks". Neste artigo ele introduziu uma nova forma de modelar dados, sendo sua principal preocupação evitar que o usuário tenha que saber como o dado está armazenado logicamente.

Segundo ele, os usuários devem ser protegidos de ter que saber como os dados estão armazenados internamente, sendo que caso haja mudanças em alguns aspectos do armazenamento as aplicações devem continuar com seu funcionamento inalterado (CODD, 1970). Além disso, a estrutura que ele propôs tinha o potencial de responder questões baseadas nos dados e o uso de disco seria mais eficiente.

Com base nessas ideias se deu início ao desenvolvimento de diversas tecnologias para armazenamento de dados que poderiam ter relações entre si, sendo esse modelo chamado de relacional. Os sistemas de gerência de bancos de dados relacionais, baseados no modelo relacional, se caracterizam por uma abordagem de gerenciar dados usando o sistema lógico de primeira ordem onde os dados são representados como tuplas agrupadas com relações entre si. A popularidade que esse modelo ganhou tornou possível que diversos avanços pudessem ser realizados, como transações *ACID* para assegurar integridade e consistência dos dados, a linguagem *SQL*, estudos de performance e segurança, entre outros, sendo ele bastante utilizado até os dias de hoje.

Os bancos de dados relacionais tiveram como arquitetura inicial a escalabilidade vertical apenas, já que foram criados antes da internet ganhar a popularidade que tem hoje. Tradicionalmente, eles foram pensados para estar em apenas um servidor que pode ser escalado adicionando mais recursos, como memória, poder de processamento e armazenamento (POKORNY, 2011).

Porém aos poucos a internet se tornou cada vez mais acessível e carregada de informações. Isso se dá por conta do surgimento de sites agregadores de conhecimento, como a Wikipédia, motores de busca, redes sociais, sites de vídeos e imagens, entre outros. Ainda, as empresas perceberam que era possível conseguir uma infinidade de tipos de informações relevantes aos negócios com base no comportamento dos usuários e assim aumentar as chances de venda.

Devido a esses fatores o número de dados gerados e armazenados se tornou muito grande, principalmente em empresas que tinham como foco a análise dos dados, que agora se viam diante de diversos desafios. Os bancos de dados utilizados enfrentavam problemas de performance quando o volume de dados crescia para estar de acordo com a demanda.

Além disso, durante o desenvolvimento e evolução de aplicações, os bancos de dados relacionais não podiam ter seu esquema modificado com o tempo e também não lidavam bem com diferentes tipos de dados. Por conta disso, novas tecnologias surgiram para solucionar esses problemas, sendo uma delas os bancos de dados NoSQL (SAHATQIJA

et al., 2018).

Em 1998, Carlo Strozzi nomeou seu banco de dados relacional que não usava SQL como NoSQL. Atualmente o termo pode ser usado para descrever tanto sistemas que não usam SQL ou, mais comumente, para descrever bancos de dados não relacionais que podem suportar SQL ou linguagens semelhantes (Not Only SQL).

NoSQL é uma família de bancos de dados que se caracterizam pela flexibilidade para o armazenamento de dados, sendo amplamente utilizados no contexto de *Big Data* por oferecer uma boa escalabilidade (ABDELHEDI *et al.*, 2017). Ele armazena dados de forma diferente da abordagem relacional, permitindo uma persistência e recuperação independente da estrutura e do conteúdo dos dados (MUS, 2019).

Um dos desafios do grande volume de dados é a variedade, já que eles podem se apresentar de diversos tipos e formatos, como estruturados, semi-estruturados e não-estruturados (HOLUBOVA; CONTOS; SVOBODA, 2021). Assim sendo, como mencionado anteriormente, existem diversos tipos de bancos de dados NoSQL, sendo a escolha da tecnologia ligada à natureza de dado e a forma que ele deve ser utilizado. Diversas soluções sacrificam alguns aspectos considerados essenciais nos bancos de dados relacionais para serem capazes de suprir as necessidades associadas ao uso. Dados nesses bancos de dados podem ser armazenados em pares chave-valor, conjuntos de colunas, documentos ou grafos. Além disso, existe a preocupação com a alta disponibilidade, que é provida geralmente com o relaxamento da consistência dos dados e/ou por meio de diversas formas de redundância (MEIER; KAUFMANN, 2019).

Dois importantes representantes da família de bancos de dados NoSQL são os *bancos de dados de documento* e os *bancos de dados orientados a grafos*. O primeiro é capaz de representar dados complexos, armazenando documentos no formato JSON (principalmente) ou XML, e permitindo operações semelhantes ao que é possível em SQL com o benefício de não ter necessariamente um esquema definido. Já os *bancos de dados baseados em grafos* se caracterizam por conter nós e relações conectando diferentes nós, sendo que cada um deles pode apresentar rótulos e propriedades (MEIER; KAUFMANN, 2019).

Os bancos de dados de documento, apesar de nem sempre terem esquemas definidos, utilizam algumas convenções para armazenar os dados e assim conseguir realizar análises. Por meio dessas convenções é possível perceber que alguns dados apresentam estruturas que definem diversos tipos de relacionamentos entre os dados, e que seria mais natural uma representação dessas estruturas em um formato de grafos (DE VIRGILIO; MACCIONI; TORLONE, 2013), e consequentemente, a adoção de um banco de dados orientado a grafos.

Entretanto, alguns desafios estão presentes para o caso de uma aplicação que deseja migrar seus dados de uma tecnologia de banco de dados de documento para um banco de dados orientado a grafos. Um exemplo está na etapa de migração, já que se houver muitos

dados seriam necessárias muitas horas de trabalho manual para identificar todos os tipos de relacionamentos entre os dados. Outro problema é a dificuldade em aprender novos paradigmas e linguagens, já que bancos de dados de documentos possuem seus próprios mecanismos de acesso a dados, que geralmente diferem dos mecanismos presentes nos bancos de dados orientados a grafos.

Por conta disso, esse trabalho propõe um processo automatizado de mapeamento e migração de dados de bancos de dados de documento para bancos de dados orientados a grafos, com foco nos sistemas de gerência de bancos de dados NoSQL MongoDB e Neo4j, que são os principais representantes de persistência de documentos e de grafos na indústria, respectivamente, e pela possibilidade de utilização de ambos sem custos adicionais por serem de código aberto. No caso do MongoDB, dados são mantidos em documentos em formato JSON.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem como objetivo o desenvolvimento de uma aplicação que realiza o mapeamento e migração de dados em formato JSON para o banco de dados orientado a grafos Neo4j.

1.1.2 Objetivos Específicos

Para que o objetivo geral seja possível, os seguintes objetivos específicos são definidos:

- Desenvolver um componente capaz de ler dados JSON presentes no banco de dados MongoDB;
- Desenvolver um componente capaz de escrever dados no Neo4j;
- Criar regras de mapeamento de dados JSON para o modelo de grafo de propriedades;
- Desenvolver uma solução eficiente para o mapeamento e migração de dados JSON para dados no formato de grafos aplicando técnicas de processamento paralelo;

1.2 METODOLOGIA

Para o desenvolvimento deste trabalho é necessário o estudo de diversas tecnologias existentes NoSQL com foco na solução de problemas de mapeamento de dados, formas de armazenamento, convenções acerca dos formatos de dados em documento e regras para construção de relacionamentos entre diferentes arquivos JSON. Além disso, deve-se realizar pesquisas acerca das tecnologias de bancos de dados NoSQL MongoDB e Neo4j.

Além disso, deve-se proceder uma pesquisa sobre o estado da arte e quais são os métodos atuais para resolver os problemas de mapeamento e migração entre tecnologias NoSQL. Esta etapa deve ser realizada através da leitura de artigos, livros, apresentações, códigos fonte e qualquer tipo de trabalho que possa ser relevante.

Posterior a esta etapa é o projeto e desenvolvimento da aplicação, considerando alternativas disponíveis atualmente e como esta aplicação irá se destacar dos outros trabalhos. Por fim, deve-se apresentar os resultados obtidos com a avaliação da aplicação.

1.3 CONTEÚDO DA MONOGRAFIA

Além desta introdução, esta monografia apresenta mais seis capítulos. O capítulo 2 apresenta a fundamentação teórica, sendo ela composta pelos conceitos de bancos de dados relacionais, transações, bancos de dados *NoSql*, alguns de seus formatos e usos e por fim uma introdução ao *Modelo de atores* para processamento paralelo. O capítulo 3 apresenta trabalhos que se propõem a realizar migrações de dados de forma semelhante e mapeia algumas das suas características para ser alvo de comparações com a ferramenta proposta. O capitulo 4 apresenta a ferramenta proposta, sendo dividido entre a solução do problema original e a evolução para o modelo de processamento paralelo e distribuído, com o final dedicado a validação dos dados. O capítulo 5 demonstra a utilização da ferramenta para migração de dados entre os bancos junto com as análises de qualidade e variação das configurações para obter melhor performance, sendo a velocidade comparada com uma ferramenta disponibilizada pelo próprio *Neo4j*. O capitulo 6 apresenta a conclusão do trabalho, sendo analisado o cumprimento dos objetivos e as possibilidades de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos essenciais para este trabalho: bancos de dados relacionais, bancos de dados NoSQL, MongoDB, Neo4j e o Modelo de Atores.

2.1 BANCOS DE DADOS RELACIONAIS

Bancos de dados relacionais são um tipo de banco de dados que utiliza o modelo relacional para armazenamento. Eles são projetados para armazenar informações estruturadas e permitir que os usuários realizem consultas complexas e análises.

A Figura 1 é um exemplo de como são representados os dados em um banco de dados relacional. Temos as tabelas *User*, *Follower*, *Tag*, *Blog* e *Comment*. Pode-se perceber que cada tabela tem colunas em sua primeira linha e valores nas demais. É possível notar que cada coluna tem um tipo específico e que existem relações entre tabelas. Por exemplo, a tabela *Follower* possui uma referencia para a tabela *User* na coluna *fuser*, sendo *fuser* um id de usuário na tabela *User*.

User (US)

	uid	uname
t_1	u01	Date
t_2	u02	Hunt

Follower (FR)

	fuser	fblog
t_3	u01	b01
t_4	u01	b02
t_5	u01	b03
t_6	u02	b01

Tag (TG)

	tuser	tcomment
t_7	u02	c01

Blog (BG)

	bid	bname	admin
t_8	b01	Information Systems	u02
t_9	b02	Database	u01
t_{10}	b03	Computer Science	u02

Comment (CT)

	cid	cblog	cuser	msg	date
t_{11}	c01	b01	u01	Exactly what I was looking for!	25/02/2013

Figura 1 – Exemplo de banco de dados relacional

Fonte: (DE VIRGILIO; MACCIONI; TORLONE, 2013)

2.1.1 Modelo relacional

O modelo relacional é a base dos bancos de dados relacionais. Ele define a estrutura dos dados e a forma como os dados são organizados em relações com tuplas e atributos. Uma relação no modelo relacional é materializada como uma tabela em um banco de dados relacional composto de linhas e colunas. Tabelas são interconectadas por meio de chaves para permitir a consulta e manipulação de dados de várias tabelas ao mesmo tempo.

Ele é baseado na álgebra relacional, que fornece um conjunto de operações para manipulação de dados, como seleção, projeção, união e junção. Segundo (SILBERSCHATZ;

KORTH; SUDARSHAN, 2020) a álgebra relacional consiste em um conjunto de operações que recebe uma ou mais relações como entrada e produz uma nova relação como resultado.

2.2 BANCOS DE DADOS NOSQL

Bancos de dados Not only SQL (NoSQL) são uma família de banco de dados que não seguem o modelo relacional. Eles são projetados para armazenar e processar grandes quantidades de dados estruturados, não estruturados ou semiestruturados.

Eles são amplamente utilizados em aplicações web, *Big Data*, Internet das Coisas (IoT) e outros. Apesar de comumente estes bancos de dados não utilizarem *SQL* como sua linguagem de consulta, o termo *NoSQL* não é interpretado como *no SQL*, mas sim como *not only SQL* (POKORNY, 2011).

Os bancos de dados *NoSQL* são mais flexíveis do que os bancos de dados relacionais, já que apresentam um esquema dinâmico e que não necessariamente precisa ser pré-definido (SAHATQIJA *et al.*, 2018). Estas características permitem que os desenvolvedores adicionem novos campos ou tipos de dados à medida que surgem novos requisitos. Além disso, os bancos de dados *NoSQL* são escaláveis horizontalmente, o que significa que é possível adicionar mais servidores para lidar com um maior volume de dados e tráfego de usuários.

A família de bancos de dados NoSQL é composta por bancos de dados chave-valor, colunar, documento e grafos. Alguns desses modelos de dados são mostrados na Figura 2. Esse trabalho enfatiza os bancos de dados de documentos e de grafos. Eles são detalhados a seguir.

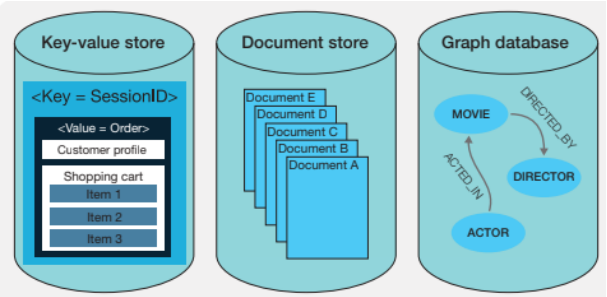


Figura 2 – Alguns exemplos de modelos de dados NoSQL

Fonte: (MEIER; KAUFMANN, 2019)

2.2.1 Bancos de dados de documento

Bancos de dados de documentos armazenam dados geralmente no formato *JavaScript Object Notation (JSON)*, sendo projetados para oferecer uma estrutura de dados flexível e escalável. JSON é um formato textual usado para representar dados complexos,

sendo muito utilizado para transmitir dados entre aplicações e armazenar informações complexas. Esse formato suporta alguns tipos de dados (inteiros, reais e *strings*), além de *arrays* e objetos, que são coleções de pares nome-valor que definem atributos. Arrays em JSON são representados entre colchetes e funcionam como mapas de inteiros para valores e objetos são delimitados entre chaves (SILBERSCHATZ; KORTH; SUDARSHAN, 2020). A Figura 3 mostra um exemplo de um dado JSON.

```
{
    "ID": "22222",
    "name": {
        "firstname: "Albert",
        "lastname: "Einstein"
    },
    "deptname": "Physics",
    "children": [
        {"firstname": "Hans", "lastname": "Einstein" },
        {"firstname": "Eduard", "lastname": "Einstein" }
    ]
}
```

Figura 3 – Exemplo de um dado JSON

Fonte: (SILBERSCHATZ; KORTH; SUDARSHAN, 2020)

O *MongoDB* é o banco de dados NoSQL de documento mais popular na indústria. Ele é de código aberto e projetado para ser escalável e flexível. Ele permite que os usuários armazenem grandes quantidades de dados e executem consultas complexas de forma rápida. Ainda, ele armazena dados em coleções de documentos em um formato Binary JSON (BSON), que é um formato binário para representar documentos *JSON*.

O acesso e modificação de dados no MongoDB é possível através de um conjunto de métodos de acesso. O método *find*, por exemplo, permite a filtragem e recuperação de atributos de documentos. Já o método *updateMany* é similar ao comando *Update* da SQL, permitindo a atualização de determinados atributos de documentos.

2.2.2 Bancos de dados de grafo

Bancos de dados baseados em grafo utilizam o modelo de grafos de propriedades para o armazenamento dos dados. Este modelo consiste de nós (fatos do mundo real) conectados a arestas (relacionamentos entre esse fatos). Para ambos é possível definir rótulos e propriedades. As propriedades seguem o padrão atributo-valor, com o nome do atributo e seu respectivo valor (MEIER; KAUFMANN, 2019). A Figura 4 exemplifica um grafo de propriedade. Pode-se ver os nós em azul e ligados a eles estão as arestas representando suas relações.

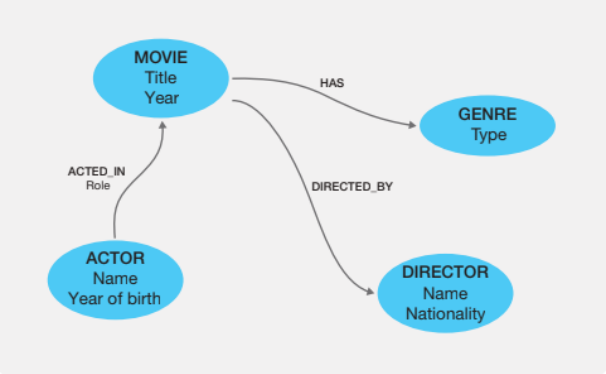


Figura 4 – Exemplo de um grafo de propriedade

Fonte: (MEIER; KAUFMANN, 2019)

Os bancos de dados de grafo apresentam alguns benefícios em relação aos bancos de dados relacionais, incluindo a capacidade de escalar horizontalmente para lidar com grandes volumes de dados, desempenho mais rápido em consultas com vários relacionamentos entre dados, e maior flexibilidade no esquema de dados.

Como comentado anteriormente, um dos principais benefícios dos bancos de dados baseados em grafo é a capacidade de navegar rapidamente através de conexões complexas e visualizar relacionamentos em um formato amigável. Essa visualização de dados em formato de grafo pode ser particularmente útil em áreas como redes sociais, detecção de fraudes e sistemas de recomendação. Eles são especialmente adequados para representar e analisar dados que possuem muitas relações complexas entre as entidades, permitindo consultas avançadas e análises de dados em tempo real.

O *Neo4j* é o mais popular de banco de dados de grafo. Ele utiliza a *Cypher* como linguagem de consulta. Cypher é uma linguagem de consulta declarativa para extrair padrões de bancos de dados de grafo. Os usuários definem suas consultas especificando nós, arestas e padrões de percorrimento. O sistema analisa esses padrões e define uma estratégia de caminhamento mais eficiente no grafo. Além disso, é possível criar, alterar e excluir nós e arestas, ordenar o resultado de consultas, dentre outras facilidades.

A Figura 5 exemplifica uma consulta escrita em Cypher. Ela retorna nós (pessoas) que possuem um relacionamento rotulado como *LOVES* com a pessoa *Dan*.

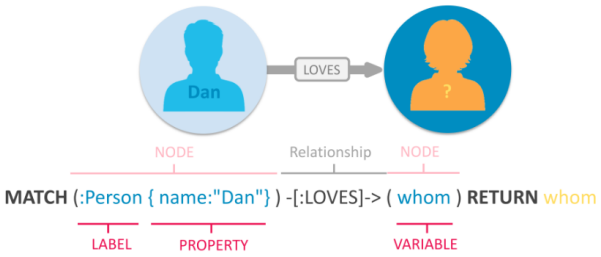


Figura 5 – Exemplo de uma consulta com Cypher

Fonte: (NEO4J, s.d.)

2.3 MODELO DE ATORES

O modelo de atores é uma teoria da computação que define os atores como elementos básicos da computação concorrente, sendo utilizado para a compreensão da teoria de concorrência e como base teórica em sistemas distribuídos (HEWITT, 2010). Cada ator é independente e encapsula estado e comportamento, podendo se comunicar com outros atores através de mensagens assíncronas. Essa abordagem permite que sistemas sejam escaláveis e tolerantes a falhas em ambientes distribuídos.

O Modelo de Atores foi criado para integrar informações de forma robusta, mesmo com inconsistências contínuas. Ele não tenta eliminar essas inconsistências, abordando uma estratégia de aceitar e lidar com elas como uma característica observada e desejada.

Um ator pode ter diferentes formas e estar presente em diferentes lugares de uma aplicação. Qualquer parte de um sistema que precise realizar computação pode ser um ator (MICHAEL NASH, s.d.). Isso é demonstrado na Figura 6, onde pode-se criar um ator que expõe métodos de uma entidade *Person*, sendo que esse ator pode se comunicar com outros atores. No exemplo essa comunicação é utilizada para realizar buscas de datas (atores *Date*) na agenda (ator *Schedule*) de forma concorrente. Neste caso, *Date* são atores filhos de *Schedule* e possuem estados independentes para a realização de trabalho.

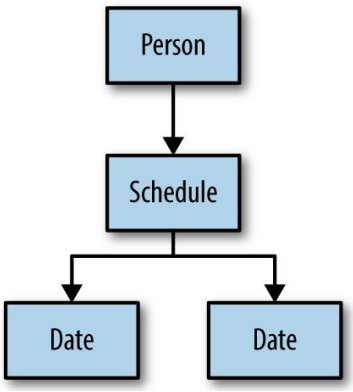


Figura 6 – Exemplo de aplicação para o Modelo de Atores

Fonte: (MICHAEL NASH, s.d.)

Um exemplo de solução baseada em atores é o *framework Ray*. Ele utiliza o modelo de atores para disponibilizar aplicações distribuídas e paralelas. Para ele, um ator representa uma computação com estado, sendo que os atores disponibilizam métodos que retornam *futuros*, que indicam a pendencia da computação, e a execução acontece em um trabalhador que possui estado (MORITZ *et al.*, 2018).

3 TRABALHOS CORRELATOS

Este capítulo apresenta a revisão sistemática e compara as propostas de três trabalhos que realizam a conversão de dados estruturados para grafos, demonstrando as regras de conversão e estratégias para detecção de relações. Estas propostas foram utilizadas como base para a realização deste trabalho e ao final são comparadas a ele.

3.1 REVISÃO SISTEMÁTICA

A Revisão Sistemática é uma metodologia comumente utilizada em pesquisa acadêmica para responder perguntas específicas e informar decisões baseadas em evidências (SAMPAIO; MANCINI, 2007). Os principais passos de uma Revisão Sistemática são:

1. Formulação da Pergunta de Pesquisa;
2. Desenvolvimento do protocolo: detalhamento de critérios de inclusão e exclusão, métodos de busca, critérios de avaliação da qualidade dos estudos incluídos, e procedimentos para a síntese dos resultados;
3. Busca de evidências: definição de termos ou palavras-chave, seguida das bases de dados e outras fontes de informação a serem pesquisadas;
4. Seleção dos estudos: avaliação de títulos, resumos e até mesmo o texto completo, seguindo de forma rigorosa os critérios de inclusão e exclusão para selecionar os estudos relevantes;
5. Análise de qualidade: uso de diferentes escalas, como lista de Delphi, PEDro e outras, visando obter um índice de qualidade metodológica e por meio dela eventualmente excluir mais estudos;
6. Extração de dados: recuperação de dados relevantes dos estudos selecionados, como métodos utilizados e resultados;
7. Síntese e Análise dos Resultados;
8. Interpretação dos Resultados: cruzamento dos resultados com a pergunta de pesquisa, considerando a qualidade dos estudos e possíveis fontes de viés.

Devido a especificidades deste trabalho e a dificuldade em encontrar artigos que fossem próximos à proposta apresentada, foi decidido considerar estudos que tivessem alguma similaridade ao trabalho proposto. Isso se reflete no critério de inclusão, que inclui dados estruturados ao invés de apenas semiestruturados, que é o caso do formato JSON.

As seguintes bases de dados bibliográficas foram pesquisadas utilizando as seguintes *strings* de busca:

- **IEEE:** *((("Document Title":json OR "Document Title":document OR "Document Title":mongodb) AND ("Document Title":graph OR "Document Title":neo4j)) AND ("Document Title":mapping OR "Document Title":conversion OR "Document Title":transformation OR "Document Title":modeling OR "Document Title":modelling OR "Document Title":persistence OR "Document Title":storage)) AND (("All Metadata":json OR "All Metadata":mongodb OR "All Metadata":document OR "All Metadata":couchdb) AND ("All Metadata":Neo4j OR "All Metadata":graph))*

- **Wiley:** *"((json OR document OR mongoDB) AND (graph OR neo4j)) AND (mapping OR conversion OR transformation OR modeling OR modelling OR persistence OR storage)"in Title and "((json OR mongodb OR document) AND (Neo4j OR graph))"anywhere*

- **ACM:** *Title:(json OR document OR mongodb) AND Title:(graph OR neo4j) AND Title:(mapping OR conversion OR transformation OR modeling OR modelling OR persistence OR storage) AND (Fulltext:(json OR mongodb OR document) AND Fulltext:(neo4j OR graph))]*

- **Scopus:** *((TITLE("mapping") OR TITLE("conversion") OR TITLE("transformation") OR TITLE("modeling") OR TITLE("modelling") OR TITLE("persistence") OR TITLE("storage")) AND((TITLE("json") OR TITLE("mongodb") OR TITLE("document")) AND (TITLE("neo4j") OR TITLE("graph")))) AND((ALL(json) OR ALL(mongoDB) OR ALL(document) OR ALL(couchdb))AND (ALL(neo4j) OR ALL(graph)))*

- **DBLP:** *(json | mongodb | document | couchdb) (neo4j | graph) (mapping | conversion | transformation | modeling | modelling | persistence | storage)*

Na sequência, critérios de inclusão e exclusão foram definidos. Critérios de inclusão:

- Redigido em inglês ou português;

- Apresenta formas de mapear dados estruturados ou semiestruturados para bancos de dados de grafo.

Critérios de exclusão:

- Não detalha um método para mapeamento de dados para bancos de dados de grafo;

- Considera apenas dados não estruturados para realizar o mapeamento.

Os resultados obtidos com a aplicação da revisão sistemática foram os seguintes:

- **IEEE**: 15 trabalhos resultantes da pesquisa, 3 respeitavam os critérios, sendo 0 relacionados ao mapeamento JSON para grafo;

- **Wiley**: 3 trabalhos resultantes da pesquisa, 0 respeitavam os critérios;
- **ACM**: 15 trabalhos resultantes da pesquisa, 2 respeitavam os critérios, sendo 0 relacionados ao mapeamento JSON para grafo;
- **scopus**: 24 trabalhos resultantes da pesquisa, 3 respeitavam os critérios, sendo 0 relacionados ao mapeamento JSON para grafo;
- **DBLP** 35 trabalhos resultantes da pesquisa, 0 respeitavam os critérios.

O resultado revelou que não foi encontrado nenhum estudo que realizasse a conversão de dados JSON (semiestruturado) para dados no formato de grafo, demonstrando o ineditismo da proposta deste trabalho. Assim sendo, foram selecionados estudos que realizam mapeamento do modelo relacional para o modelo de grafo. Como existem trabalhos que se propõem a converter dados semiestruturados para relacionais, pode-se assumir que teoricamente é possível a conversão para o formato de grafo, mesmo que seja necessário utilizar um intermediário (JSON para estruturado e depois para grafo). Estes estudos são detalhados a seguir.

3.2 *MIGRATION OF DATA FROM RELATIONAL DATABASE TO GRAPH DATABASE*

Este trabalho discute a migração de dados jurídicos, que possuem relacionamentos complexos entre diferentes tipos e partes de dados, de um banco de dados relacional (*MySQL)* para grafo (*Neo4j*) (ALOTAIBI; PARDEDE, 2019). Os dados e seus relacionamentos estão apresentados na Figura 7.

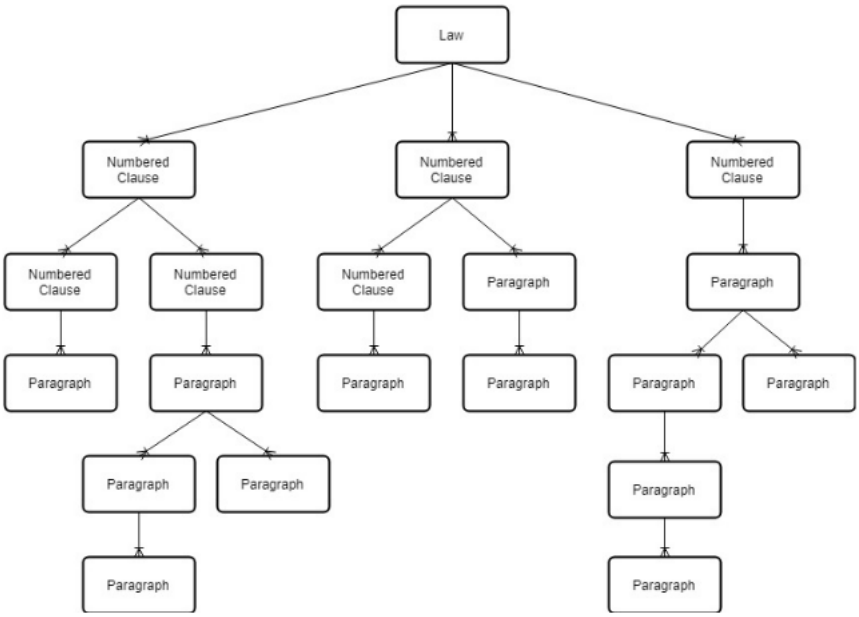


Figura 7 – Entidades do projeto

Fonte: (ÜNAL; OĞUZTÜZÜN, 2018)

Já a Figura 8 mostra de que forma os autores originalmente projetaram o sistema como um modelo relacional, mas isso causou problemas de desempenho devido à grande quantidade de dados e operações de junção complexas necessárias para consultar e acessar os dados. Para solucionar esse problema, a modelagem dos dados foi redesenhada no formato de grafo (demonstrado na Figura 9), o que melhorou o desempenho ao permitir uma fácil navegação pelos dados por meio da busca por rótulos, relacionamentos e navegação entre nós.

O trabalho descreve regras de transformação usadas para migrar os dados. Cada tabela é representada por um rótulo em nós, cada linha em uma tabela é um nó, e as colunas das tabelas se tornam propriedades de nó. Chaves estrangeiras são substituídas por arestas, e as tabelas de junção são transformadas em arestas, sendo que as colunas destas tabelas se tornam propriedades das arestas.

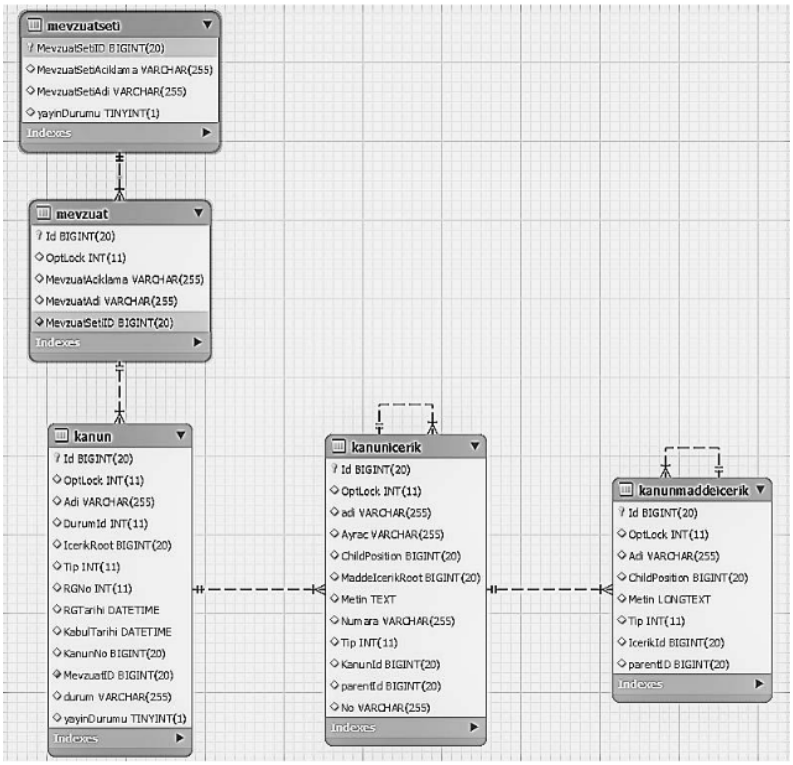


Figura 8 – Esquema relacional

Fonte: (ÜNAL; OĞUZTÜZÜN, 2018)

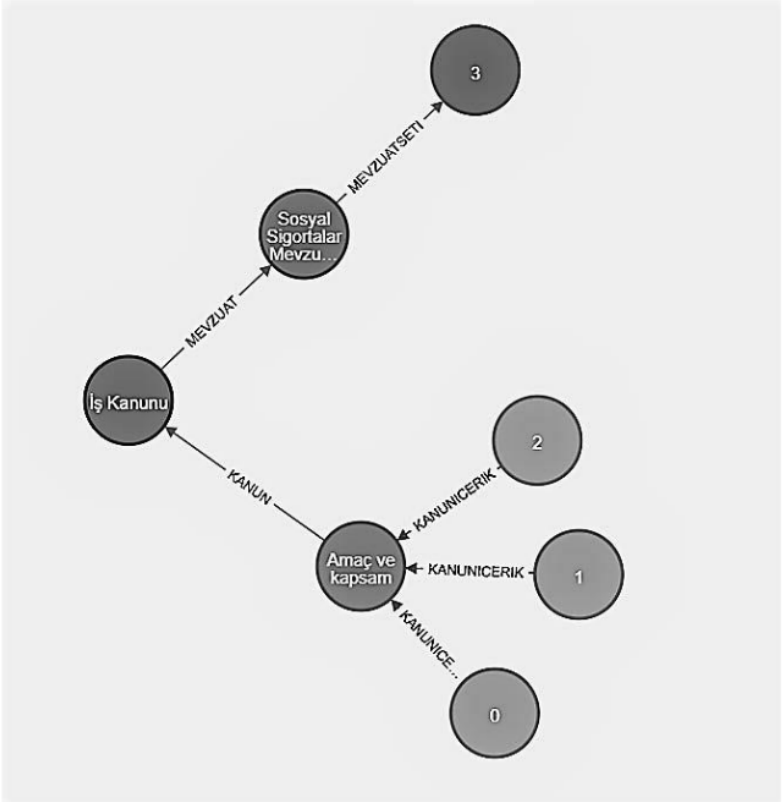


Figura 9 – Esquema de grafo

Fonte: (ÜNAL; OĞUZTÜZÜN, 2018)

Os autores compararam o desempenho de acesso de uma consulta complexa aos dados dos esquemas de banco de dados relacional e de grafo, sendo a execução da consulta no banco de dados Neo4j 10 vezes mais rápida que no banco de dados MySQL (relacional). Segundo os autores, a execução de consultas no esquema de grafo é mais direta por que a persistência dos dados do domínio imitam melhor os objetos do mundo real e seus relacionamentos.

3.3 *AMANDA: A MIDDLEWARE FOR AUTOMATIC MIGRATION BETWEEN DIFFERENT DATABASE PARADIGMS*

Neste trabalho os autores afirmam que a maioria dos estudos anteriores apresenta um foco na avaliação dos metadados do banco de dados, rastreamento de esquema e conversão de diagrama entidade-relacionamento, não permitindo aos usuários selecionar um subconjunto de banco de dados ou atributos de tabela para migrar. Outros estudos dependem de ferramentas de terceiros, como *Kafka* e *Apache Phoenix* (QUEIROZ *et al.*, 2022).

Além disso, os trabalhos relacionados analisados não podem ser estendidos para oferecer suporte a mais de um banco de dados de origem e destino, sendo um diferencial

desse trabalho a possibilidade de novos módulos serem desenvolvidos e integrados à ferramenta de migração proposta.

Como demonstrado na Figura 10, para fazer a migração a ferramenta proposta, denominada *AMANDA*, lê um arquivo de configuração contendo as características da migração, como quais dados devem ser migrados e quais são suas propriedades e relações, chamado de *schema.json*.

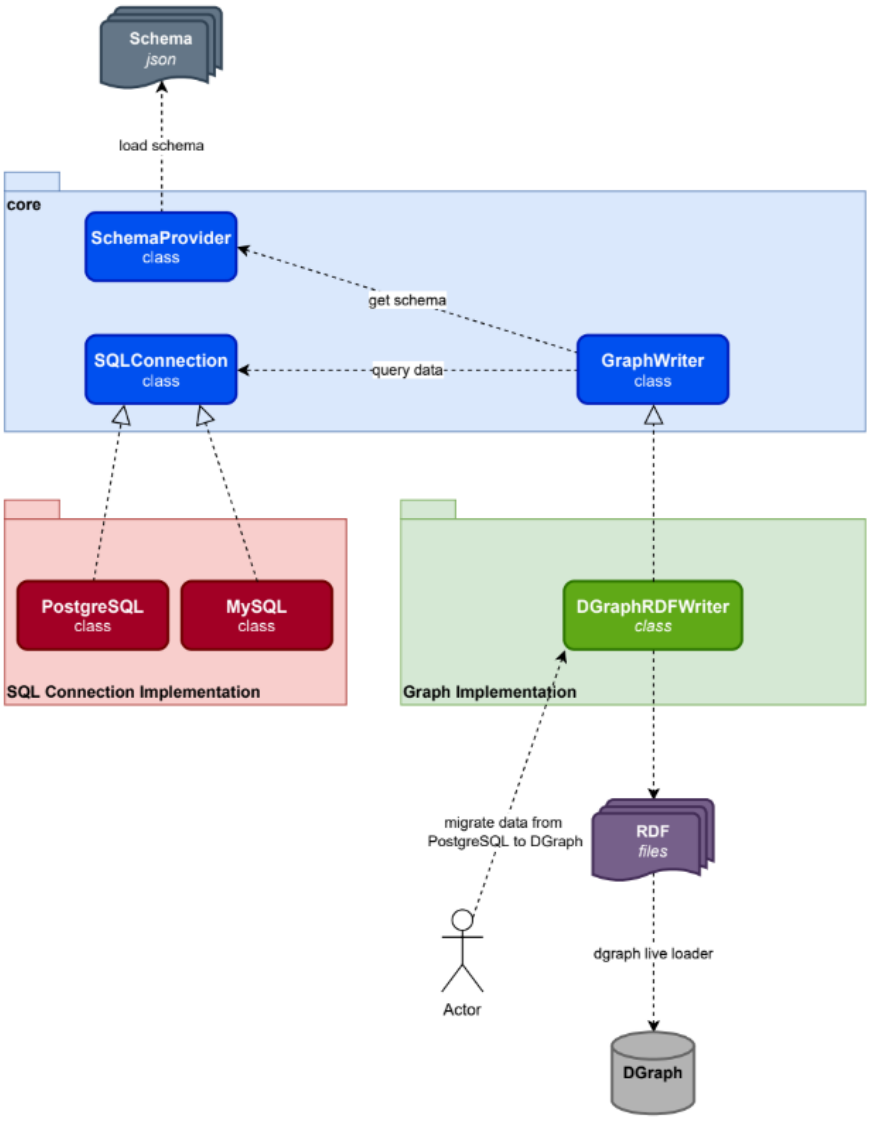


Figura 10 – Arquitetura do middleware AMANDA

Fonte: (QUEIROZ *et al.*, 2022)

A Figura 11 demonstra a configuração presente no arquivo para a criação de vértices. Depois disso, por meio das tabelas e atributos definidos, os dados são buscados do banco de dados de origem por meio de consultas *SQL*.

```
{
  "vertices": [{
      "source": {
        // Table's name in source DB
        "tableName": "customers",
        "orderField": "CustomerID"
      },
      "target": {
        // Vertex name in target DB
        "vertexName": "Customer",
        "vertexIDs": [
          // In Dgraph, each entity must have an unique ID
          "CustomerID"
        ]
      },
      "fields": [
        "CustomerID",
        "CompanyName",
        "ContactName",
        "ContactTitle",
        "Address",
        "City",
        "Region",
        "PostalCode",
        "Country",
        "Phone",
        "Fax"
      ]
    }
    ...
  ]
}
```

Figura 11 – Seção de vértices do schema.json

Fonte: (QUEIROZ *et al.*, 2022)

De acordo com a Figura 12, *AMANDA* inicialmente busca as informações nas tabelas (**A**), posteriormente cria os nós (**B**) e por fim cria os relacionamentos (**C**), sendo a tabela *order_details* transformada em aresta.

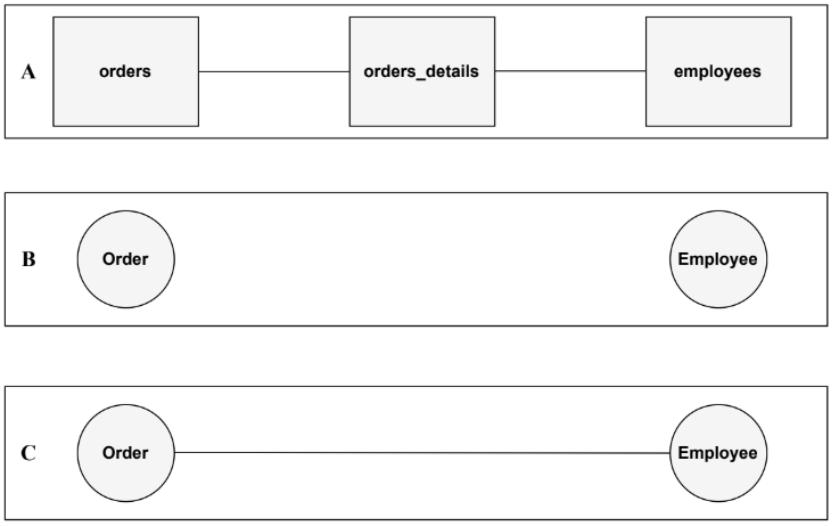


Figura 12 – Migração das tabelas orders, orders_details e employees para o banco de dados de grafo

Fonte: (QUEIROZ *et al.*, 2022)

Depois que os nós são definidos, a ferramenta lê a seção de arestas no *schema.json*, conforme demonstra a Figura 13 e usa as chaves primárias e estrangeiras para criar as arestas no banco de dados de destino. As definições de nós e arestas são salvas no formato *RDF* e o resultado é convertido em nós do banco de dados de grafo.

```
{
    "edges": [{
      // Source table on Releational DB
      "source": {
        "tableName": "products",
        "orderField": "ProductID",
        "primaryKey": "ProductID",
        "foreignKey": "SupplierID"
      },
      // Target vertex on Graph DB
      // (Product)--[Product.supplier]-->(Supplier)
      "target": {
        "leftVertexName": "Product",
        "edgeName": "Product.supplier",
        "rightVertexName": "Supplier"
      }
    }
    ...
    ]
}
```

Figura 13 – Seção de arestas do schema.json

Fonte: (QUEIROZ *et al.*, 2022)

O trabalho também avalia o desempenho da migração em termos de tempo de processamento. Todo o processo de migração levou apenas 1 segundo para ser concluído, com ambos os bancos de dados sendo executados na mesma máquina. A quantidade de RAM usada foi de 19,47 MB, e o tempo de *CPU* foi de 0,54 segundos para a migração de

Postgres para *Dgraph*. O tempo de CPU para a migração de *MySQL* para *Dgraph* foi de 0,57 segundos.

O ótimo desempenho do AMANDA demonstra que, por depender de um *schema.json* fornecido antecipadamente contendo todas as informações de vértices e arestas a serem migrados, ela é superior a uma abordagem baseada em algoritmos automáticos de migração que busca os metadados do banco de dados ou os trabalhos que utilizam um diagrama ER para identificar os vértices e arestas para serem migrados, conforme realizado por trabalhos relacionados.

3.4 *TRANSFORMATION OF SCHEMA FROM RELATIONAL DATABASE (RDB) TO NOSQL DATABASES*

Este trabalho descreve diversas regras de relacionamento para transformação de dados em bancos relacionais para NoSQL, abordando três das quatro formas convencionais de armazenamento (baseado em colunas, documentos e grafos), sendo a chave-valor indicada para um trabalho futuro (ALOTAIBI; PARDEDE, 2019). As regras referentes à transformação relacional-grafo são descritas a seguir.

A Figura 14 mostra a transformação de relacionamentos *um-para-um* e *um-para-muitos*. Nestes casos é criado um nó para cada tabela relacional A e B (nó inicial e nó final). O id do nó inicial é incluído como uma propriedade no nó final, e uma aresta é definida do nó inicial para o final. Se o relacionamento for do tipo um-para-muitos, a tabela do lado um se torna o nó inicial e a tabela do lado muitos se torna o nó final. Caso contrário, qualquer tabela pode se tornar o nó inicial.

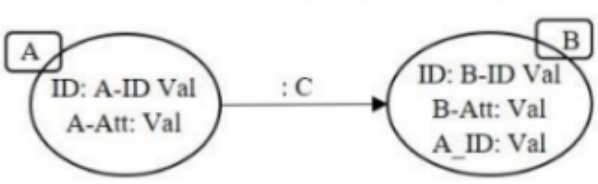


Figura 14 – Transformação de relacionamentos um-para-um e um-para-muitos

No caso de relacionamentos *muitos-para-muitos*, é criado um nó para cada tabela A e B, e a tabela associativa AB se torna uma aresta cujas propriedades são os atributos de AB. Dessa forma, é criado um nó para os elementos de A, de B e a associação entre A e B, junto com suas propriedades, se torna uma aresta. Essa regra é aplicável também para relações que possuem tabelas associativas que relacionam mais de duas tabelas.

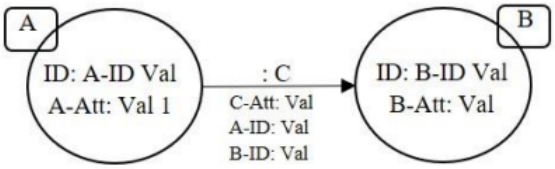


Figura 15 – Transformação de relações muitos para muitos

Para relacionamentos de especialização, onde uma tabela A representa uma entidade genérica e as demais tabelas (B e C, por exemplo) representam suas especializações, é criado um nó para cada tabela que participa do relacionamento, como mostra a Figura 16. O nó A se tornando o nó inicial e os nós B e C os nós finais das arestas conectando A-B e A-C. O ID do nó A se torna propriedade nos nós B e C.

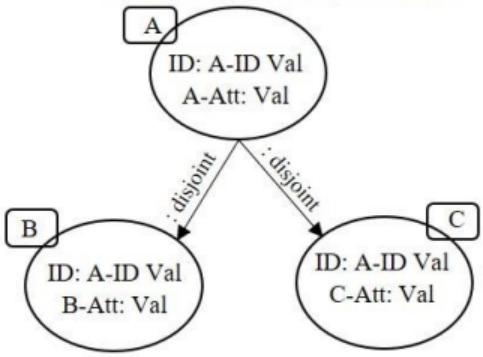


Figura 16 – Transformação de especializações

Para relacionamentos derivados de uma união de tipos, onde uma tabela A representa um tipo união e as demais tabelas B e C representam suas subclasses, é criado também um nó para cada tabela que participa do relacionamento, como mostra a Figura 17. A diferença está nas arestas geradas, que agora partem dos nós B e C para o nó A.

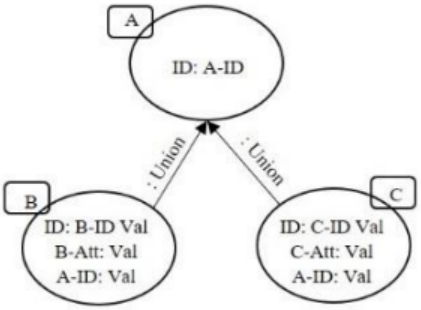


Figura 17 – Transformação de tipos união

Por fim, para relacionamentos de agregação, onde uma tabela B é um agregado (tabela dependente) de uma tabela A, é criado um nó para cada tabela é uma aresta que parte do nó A para o nó B. O ID do nó A é incluído como propriedade no nó B. A Figura 18 exemplifica essa regra.

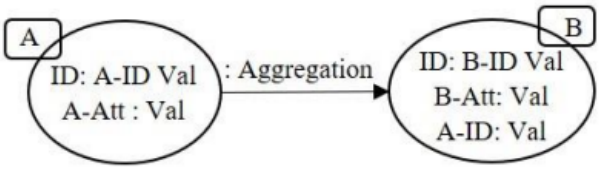


Figura 18 – Transformação de agregações

3.5 DISCUSSÃO

A Tabela 1 a seguir apresenta um comparativo das principais características observadas nos trabalhos relacionados ao mapeamento de modelos de bancos de dados para o modelo de grafo de propriedades. As características consideradas são as seguintes: *(i) origem; (ii) estrutura; (iii) linguagem de programação; (iv) destino; (v) modo de definição de esquema; (vi) modo de execução;* e *(vii) forma de execução.*

Tabela 1 – Comparação entre trabalhos correlatos

Trabalho	(ÜNAL; OĞUZTÜZÜN, 2018)	(QUEIROZ *et al.*, 2022)	(ALOTAIBI; PARDEDE, 2019)	**Este trabalho**
Origem	MySQL	Postgres	Oracle Live	**MongoDB**
Estrutura	Estruturado	Estruturado	Estruturado	**Semiestruturado**
Linguagem de programação	Java	Python	SQL (execução manual de query)	**Python**
Destino	Neo4J	Dgraph	Neo4J	**Neo4J**
Modo de definição de esquema	Automatizado (Crawler)	Manual (Arquivo de configuração)	Manual	**Automático (evolução de esquema em *streaming*)**
Modo de execução	Sem informação	Local	Manual	**Local ou Distribuído**
Forma de execução	Em lote	Em lote	Manual	***Streaming***

Como é possível notar na Tabela 1, este trabalho apresenta características bem diferentes em relação aos estudos encontrados que haviam algum tipo de mapeamento para o banco de dados de grafo, sendo seu principal diferencial lidar com dados de entrada semiestruturados no formato JSON, ser completamente automático, construir o esquema dos dados durante a migração (evolução de esquema em v), a possibilidade de rodar localmente ou de forma distribuída, sendo a distribuição gerenciada pelo *framework Ray*[1], e a possibilidade de rodar como *streaming* ao invés de apenas em lote. A proposta deste trabalho, desenvolvida na forma de uma ferramenta, é detalhada no próximo capítulo.

[1] https://www.ray.io/

4 ABORDAGEM PROPOSTA

A abordagem proposta como solução para a problemática tratada neste trabalho é composta pela estratégia de mapeamento, que descreve as regras de mapeamento e como elas podem ser aplicadas em um *pipeline* separada por componentes, seguida pela implementação da ferramenta, que descreve como foi implementada a ferramenta *Json2Node*, tendo como base a estratégia de mapeamento e que resulta em um *script* em *Python* que realiza a migração dos dados utilizando uma estratégia de *streaming* de dados e distribuição de processamento por meio do modelo de atores, sendo por fim seguida da demonstração da aplicação web para gerenciamento das migrações.

4.1 ESTRATÉGIA DE MAPEAMENTO

A estratégia de mapeamento utiliza uma arquitetura base e algoritmos para demonstrar como o *script* irá realizar o mapeamento dos dados JSON para o modelo de grafos de propriedades seguindo as regras de mapeamento. A arquitetura que serve de base para a ferramenta é composta pelos módulos de conexão com o *MongoDB*, mapeamento dos objetos *JSON* para uma estrutura interna chamada *Node* (*JsonToGraph*) e o cliente do *Neo4j*, conforme ilustra a Figura 19.

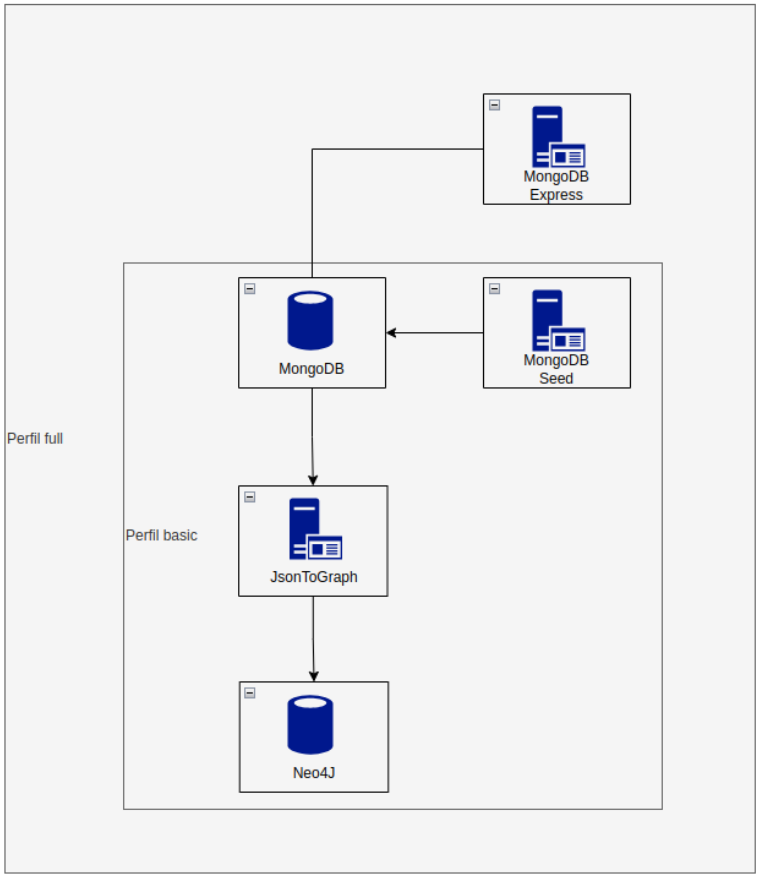


Figura 19 – Arquitetura da ferramenta

A composição desses módulos segue uma estratégia de desenvolvimento onde há a possibilidade deles serem abstraídos para possibilitar a criação de novos módulos que tratem outros tipos de dados e bancos de dados. Essa abstração se dá pelos tipos aceitos, já que o módulo *JsonToGraph* não necessita que os dados tenham como origem o *MongoDB*. Ele consegue lidar com qualquer tipo de dado que consiga ser representado como um dicionário da linguagem *Python*. Um dicionário Python é uma estrutura que armazena dados no formato de pares chave-valor, semelhante à estrutura do documento JSON. Ele é uma tabela de *hash* e é usado para mapear uma chave única a um valor, sendo que as chaves devem ser de um tipo imutável, como *string*, enquanto os valores podem ser de qualquer tipo. Isso torna a representação do dicionário similar a um documento JSON.

Já a versão denominada *ferramenta completa* (perfil *full* na Figura 19) necessita de uma instancia do *MongoDB*, *Neo4j* e do *JsonToGraph*. Tudo o que é necessário para o funcionamento da ferramenta está configurado em um *docker-compose*, sendo ele composto por *containers* para teste do *MongoDB* junto com a ferramenta de administração do *MongoDB* (*MongoDB Express*). Para alimentar o banco de dados com os dados de exemplo foi desenvolvido um *script* em *Bourne-Again SHell (BASH)* chamado de *Mongo seed*. Ele utiliza a interface de comando do *MongoDB* para adicionar linhas conforme os exemplos

desejados. *BASH* é um interpretador de comandos e linguagem de *script* que pode ser utilizado em sistemas Unix e Linux e foi amplamente utilizado neste trabalho para o gerenciamento da infraestrutura.

O *script* foi desenvolvido inicialmente por meio de um *pipeline* ilustrado na Figura 20. Neste *pipeline* todos os objetos são processados de forma síncrona e ordenada, sendo o resultado acumulado para posteriormente ser escrito em lote no Neo4j.

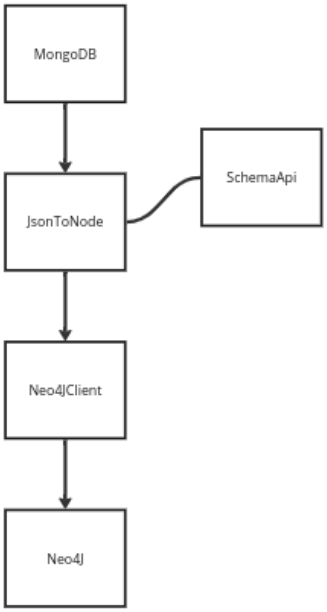


Figura 20 – *Pipeline* de processamento da ferramenta

Este *pipeline* executa o processamento de mapeamento e migração de dados JSON para o Neo4j respeitando a arquitetura da ferramenta. Seus componentes são detalhados a seguir.

4.1.1 MongoDB Reader

O *MongoDB Reader* é responsável por se conectar a um banco de dados *MongoDB*, recebendo as configurações necessárias por meio de variáveis de ambiente. Para a conexão, a ferramenta utiliza a biblioteca *pymongo*, instanciando um cliente.

Uma vez estabelecida uma conexão, ele retorna um *cursor* que itera sobre a coleção de documentos JSON até receber todos os dados. O retorno dos dados se dá no formato de dicionário do *Python*, sendo ele uma tradução do formato armazenado originalmente no banco de dados *MongoDB* no formato BSON. Neste contexto, iteração é o processo de percorrer sequencialmente a coleção de objetos sem expor todos os elementos, assim economizando recurso de memória e computação.

Para fim de exemplificação e simulação do algoritmo, a Figura 21 mostra um dado que representa uma **questão** fictícia guardada em uma coleção chamada de **questões**, sendo que essas questões possuem dados aninhados, sendo eles **respostas** e dados aninhados em **respostas** (**usuário**). Esse dado (**questão**) é um elemento da coleção que está sendo iterada, e ele é transformado em uma entrada no dicionário e irá alimentar o próximo componente.

Coleção questões

```
{
    "respostas": [
        {
            "usuario": {
                "nome": "__snoopyDog",
                "respostas": 10
            }
        },
        {
            "usuario": {
                "nome": "elchatgpt",
                "respostas": 7
            }
        }
    ]
}
```

Figura 21 – Representação da coleção questões em JSON

4.1.2 JsonToNode

O componente *JsonToNode* é responsável por receber os dados do cursor do *MongoDB Reader*, sendo ele tratado como um *stream* de dicionário *Python*. Neste contexto, *stream de dados* se refere a forma de acesso aos dados: eles são tratados de maneira sequencial e contínua, sem a necessidade de carregar todo o conjunto de dados na memória.

Após receber o *stream de dados* ele deve iterar sobre cada um. Cada dado recebe um número identificador único para a aplicação, sendo ele utilizado internamente antes e depois da criação do dado no banco de dados destino. Nessa iteração ele identifica os atributos do objeto e por meio deles cria sua identidade interna, chamada pela ferramenta de **Schema** (esquema), junto com as informações de origem (como o nome do atributo pai), chamada de **Table** (tabela), e envia essas informações junto com os dados do objeto para o componente de escrita no *Neo4j* por meio de um objeto chamado de **Node**. O Algoritmo 1 demonstra de que forma acontece todo o processamento dentro do componente.

Algorithm 1 Algoritmo de mapeamento dicionário para Node

```
1:  procedure MigrarDicionario(dicionario, nivel, campoPai, schemaApi)
2:      identificadorUnicoDoNo ← pegarProximoIndentificador()
3:      if nivel == 0 then
4:          if tabelaRaiznaoexiste then
5:              tabelaRaiz = CriarTabela(nomeColecao, schemaApi.novoSchemaVazio())
6:          end if
7:          schemaApi.atualizarSchema(tabelaRaiz.schema, dicionario)
8:          node ← Node(identificadorUnicoDoNo, tabelaRaiz)
9:      else
10:         schema ← schemaApi.identificarSchema(dicionario)
11:         tabela ← pegarTabelaOuCriar(campoPai, schema)
12:         node ← Node(identificadorUnicoDoNo, tabela)
13:     end if
14:     for all campo ∈ dicionario do
15:         if Tipo(campo) == dicionario ou Tipo(campo) == lista[dicionario] then
16:             node_relacao ← MigrarDicionario(campo, nivel + 1, campo.nome)
17:             node.relations ← node_relacao
18:         else
19:             node.common_values[campo.nome] ← campo
20:         end if
21:     end for
22:     return node
23: end procedure
```

Para migrar dados, conforme mostra a linha 1, o componente recebe como parâmetros da função de migração a representação do objeto JSON em *Python*, sendo ele um dicionário, em conjunto com o nível do dado, que inicialmente é 0, mas que muda conforme a função realiza a recursão, o nome do campo pai, sendo ele presente apenas nos objetos aninhados e uma instância de uma classe que gerencia as informações de identidade dos dados, os esquemas, sendo essa classe chamada de *SchemaAPI*. Os *Schemas* armazenam informações de atributos dos dados já processados. O nome do campo pai é utilizado para compor a instância de *Table*, que armazena as informações de origem do objeto. Conforme mostra a Figura 22, a classe *Table* é composta pelos campos:

- **id**: Identificador interno único;
- **name**: Nome do atributo de origem;
- **schema**: Informações de identidade (esquema);

```
@dataclass
class Table:
    id: int
    name: str
    schema: Schema
```

Figura 22 – Classe Table

A linha 2 indica que o resultado do processamento do dicionário recebido é transformado na classe *Node*, recebe um identificador único, sendo ele sequencial. Esse identificador é gerenciado pelo próprio componente, que garante que não há repetição.

Já na linha 3 há a separação entre os níveis. O objeto patriarca da coleção recebe o nível 0. P conta disso ele executa as instruções das linhas 4, 5 e 6. Já os demais objetos, que estão todos aninhados, recebem níveis conforme a função realiza recursões.

As linhas 4, 5 e 6 são responsáveis por verificar se o componente já criou uma tabela chamada de raiz, caso contrário ela é criada. Essa tabela é especial, sendo criada para guardar as informações de origem da coleção e criar uma ligação entre todos os dados presentes no nível 0, ou seja, todos os objetos patriarcas, juntando todos eles em uma só origem e identidade. Essa origem recebe o nome da própria coleção.

Dessa forma, a Figura 21 mostra como um objeto de nível 0 é armazenado em um documento JSON. Temos que o nível 0 se refere a **questões**, o nível 1 a **respostas** e o nível 2 a **usuário**. O nível muda conforme a função realiza recursões, e ela sempre realiza uma recursão para processar um objeto aninhado.

O nível 0, que são os objetos **questões**, estão todos presentes em uma mesma tabela, sendo essa tabela chamada internamente de raiz. Essa tabela raiz tem como nome a coleção, portanto se chamará **questões**, e para ela as mudanças nos metadados são tratados pela descoberta de esquema de forma incremental, ou seja, todas as colunas são acumuladas e nenhuma pode ser removida, já que isso poderia invalidar um objeto migrado. Isso é visível na linha 7, que executa a atualização do esquema adicionando novos atributos que podem ter aparecido neste objeto. A identidade do objeto, chamado de *Schema* (esquema), conforme mostra a Figura 23, é composta pelos seguintes campos:

- **id**: Identificador interno único;
- **column_information**: Um dicionário que contém as informações de nomes de atributos e suas informações, por meio da classe **Column**.

```python
@dataclass
class Schema:
    id: int
    column_information: Dict[str, Column]
```

Figura 23 – Entidade Schema

Já as informações de coluna (*Column*), conforme mostra a Figura 24, são as seguintes:

- **name**: Nome do atributo do objeto;
- **is_optional**: Booleano que marca se a coluna é opcional ou obrigatória;
- **is_relationship**: Booleano que marca se uma coluna é utilizada para criar uma relação entre nós.

```python
@dataclass
class Column:
    name: str
    is_optional: bool
    is_relationship: bool
```

Figura 24 – Entidade Column

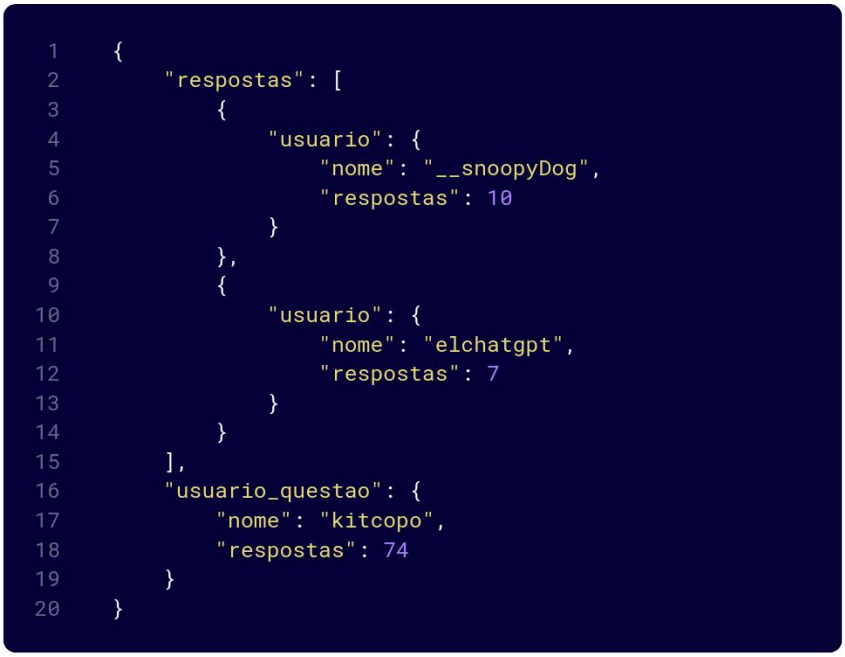
Na linha 8, a variável *node* recebe um objeto *Node* contendo todas as informações processadas do dicionário, sendo elas as informações de origem presentes na classe *Table* e as informações de identidade presentes na classe *Schema*, que para o nível 0 são todas referentes a raiz. A Figura 25 demonstra como o objeto *questoes* está representado na linha 8, sendo a representação uma instância de *Node* com o atributo de tabela contendo a *Table* raiz, e na instância de *Table* o atributo de esquema contendo o *Schema* raiz.

```
1   Node(
2       id=1,
3       table=Table(
4           id=1,
5           name="questoes",
6           schema=Schema(
7               id=1,
8               column_information={
9                   "respostas": Column(
10                      name="respostas",
11                      is_optional=True,
12                      is_relationship=True,
13                  )
14              },
15          ),
16      )
17  )
```

Figura 25 – Resultado do Node para questoes

Nas linhas 10 e 11 é utilizada a *SchemaAPI* para identificar a identidade do dicionário. Após isso, o componente tenta obter a ligação entre o nome do campo pai com o esquema obtido, sendo que caso haja divergências um novo objeto de origem é criado. Caso haja conflito entre objetos de origem, ou seja, caso os esquemas sejam divergentes, é criado uma nova *Table* com o novo esquema. A preferência sempre é a de identificar o objeto pelo *Schema* e depois pelo campo pai, já que o esquema é a identidade do objeto.

A Figura 26 exemplifica um caso onde há conflito de nomes de origem para um objeto. Nela, o objeto patriarca *questoes* possui um objeto aninhado novo chamado de *usuario_questao*. Esse objeto novo tem 0% de diferença com o objeto *usuario*, que se localiza dentro do objeto *respostas*. Dessa forma, tanto *usuario_questao* como *usuario* terão o mesmo rótulo, tabela e esquema, sendo que o nome é dado pelo primeiro objeto a ser processado.



```
{
    "respostas": [
        {
            "usuario": {
                "nome": "__snoopyDog",
                "respostas": 10
            }
        },
        {
            "usuario": {
                "nome": "elchatgpt",
                "respostas": 7
            }
        }
    ],
    "usuario_questao": {
        "nome": "kitcopo",
        "respostas": 74
    }
}
```

Figura 26 – Objeto usuario com campo pai usuario e usuario_questao

Por fim, nas linha 12 e 13 é encerrada a etapa de identificação do objeto com a criação de uma instância de *Node* contendo os dados de tabela e esquema identificados. Esta instância será utilizada na segunda etapa do algoritmo.

Na linha 12 a classe *Node* recebe as informações obtidas por meio da descoberta de esquema. Para compor a classe *Node* é necessário considerar os atributos obrigatórios e opcionais, sendo os atributos não obrigatórios preenchidos durante a fase de separação entre os atributos simples e os objetos aninhados. Como demonstra a Figura 27, a classe é composta pelos campos:

- **id**: Identificador interno do nó;
- **table**: Informações da tabela;
- **common_values**: um dicionário contendo os dados de atributo e valor originais, sem os atributos de relacionamento. Não é obrigatório;
- **relationship_references**: um dicionário contendo os dados de nome do atributo e referência a outro nó formando uma relação. A referência em memória aponta para uma classe instanciada na lista **relations**. Não é obrigatório;
- **relations**: lista contendo todos os nós um (1) nível acima que formam uma relação com esse nó. Não é obrigatório.

```python
@dataclass
class Node():
    id: int
    table: Table
    common_values: dict[str, Any] = field(default_factory=lambda: {})
    relationship_references: dict[str, Any] = field(default_factory=lambda: {})
    relations: list[Node] = field(default_factory=lambda: [])
```

Figura 27 – Entidade Node

A linha 14 marca o inicio da segunda etapa do algoritmo. Nessa etapa são realizadas as recursões que criam novos níveis, sendo eles compostos pelos objetos aninhados e seus atributos pai. Para isso, é feita uma iteração sobre todas as chaves e valores do dicionário, sendo que o valor está representado como a variável *campo* no Algoritmo 1 e o seu nome está representado como atributo (*campo.nome*).

Na linha 15 é feita a verificação se o valor do campo que está sendo iterado é do tipo dicionário ou lista de dicionários, o que significa que ele é formado por objetos aninhados. Caso seja, na linha 16 é feita uma recursão no procedimento, sendo fornecido o valor do campo, o incremento de um nível e o nome do campo pai. O resultado, conforme mostra a linha 17, é incluído no atributo *relations* da instância de *Node* que está sendo modificada.

No exemplo das questões do *StackOverflow*, o atributo **respostas** seria identificado como lista de objetos aninhados e os objetos contidos seriam retroalimentados no procedimento, recebendo o nível 1 e *campoPai respostas*. O resultado do algoritmo é demonstrado na Figura 28.

```
Node(
    id=2,
    table=Table(
        id=2,
        name="respostas",
        schema=Schema(
            id=2,
            column_information={
                "usuario": Column(
                    name="usuario",
                    is_optional=True,
                    is_relationship=True
                )
            }
        )
    )
)
```

Figura 28 – Algoritmo aplicado no objeto aninhado respostas

Ainda nessa etapa, o atributo **usuario** seria identificado como objeto aninhado. Dessa forma, ele seria retroalimentado para o procedimento recebendo o nível 2 e o

campoPai usuario. O resultado é motrado na Figura 29.

```
Node(
    id=3,
    table=Table(
        id=3,
        name="usuario",
        schema=Schema(
            id=3,
            column_information={
                "nome": Column(
                    name="nome",
                    is_optional=True,
                    is_relationship=False
                ),
                "respostas": Column(
                    name="respostas",
                    is_optional=True,
                    is_relationship=False
                )
            }
        )
    ),
    common_values={
        "nome": "__snoopyDog",
        "respostas": 10
    },
    relations=[],
    relationship_references={}
)
```

Figura 29 – Algoritmo aplicado no objeto aninhado usuario

Como o objeto **usuario** não possui objetos aninhados, o procedimento se encerraria retornando ele como resposta, assim continuando o processamento do objeto de nível 1 a qual ele está aninhado. O resultado é apresentado na Figura 30.

As linhas 18 e 19 encerram a segunda etapa do algoritmo, sendo responsável por armazenar os dados simples no campo *common_values* da instância de *Node*, que é o resultado final, conforme mostram as linhas 20, 21, 22 e 23.

Nesta etapa, o procedimento retorna ao objeto inicial, que recebe o resultado do objeto aninhado. O resultado é mostrado na Figura 31, com a observação que os dados de usuário foram truncados pelo tamanho da imagem.

```
Node(
    id=2,
    table=Table(
        id=2,
        name="respostas",
        schema=Schema(
            id=2,
            column_information={
                "usuario": Column(
                    name="usuario",
                    is_optional=True,
                    is_relationship=True
                )
            }
        )
    ),
    common_values={},
    relationship_references={"usuario": RelationshipReference(table_name="usuario", node_id=3)},
    relations=[
        Node(
            id=3,
            table=Table(
                id=3,
                name="usuario",
                schema=Schema(
                    id=3,
                    column_information={
                        "nome": Column(
                            name="nome",
                            is_optional=True,
                            is_relationship=False
                        ),
                        "respostas": Column(
                            name="respostas",
                            is_optional=True,
                            is_relationship=False
                        )
                    }
                )
            ),
            common_values={
                "nome": "__snoopyDog",
                "respostas": 10
            },
            relations=[],
            relationship_references={}
        )
    ]
)
```

Figura 30 – Resultado do algoritmo aplicado no objeto respostas

```
Node(
    id=1,
    table=Table(
        id=1,
        name="questões",
        schema=Schema(
            id=1,
            column_information={
                "respostas": Column(
                    name="respostas",
                    is_optional=True,
                    is_relationship=True
                )
            }
        )
    ),
    relationship_references={"respostas": RelationshipReference(table_name="respostas", node_id=2)},
    relations=Node(
        id=2,
        table=Table(
            id=2,
            name="respostas",
            schema=Schema(
                id=2,
                column_information={
                    "usuario": Column(
                        name="usuario",
                        is_optional=True,
                        is_relationship=True
                    )
                }
            )
        ),
        common_values={},
        relationship_references={"usuario": RelationshipReference(table_name="usuario", node_id=3)},
        relations=[
            Node(
                id=3,
                table=Table(…
                common_values={
                    "nome": "__snoopyDog",
                    "respostas": 10
                },
                relations=[],
                relationship_references={}
            )
        ]
    )
)
```

Figura 31 – Resultado do algoritmo aplicado no objeto raiz

4.1.3 Schema API

A *Schema API* é uma *API* utilizada para gerenciamento de todos os esquemas identificados, além de ser responsável por sua criação e atualização de forma incremental. Sua principal utilidade é identificar o objeto dado seus metadados. Os dados JSON que são recebidos não possuem identificação, ou seja, não é possível saber quais campos eles devem possuir e quais campos diferenciam ele de outros objetos.

No exemplo proposto na Figura 21, inicialmente não há como saber que **respostas** e **usuario** são objetos diferentes e não devem receber o mesmo rótulo. Por conta disso, conforme mostra a linha 10 do Algoritmo 1, quando um objeto de nível diferente de 0 é recebido a *Schema API* olha para todos os *Schemas* que possui e diz se esse objeto se parece ou não com algum deles. Se não parece, ela cria um novo *Schema* para ele contendo todos os atributos que foram identificados. A Figura 32 mostra os *Schemas* identificados para todos os objetos.

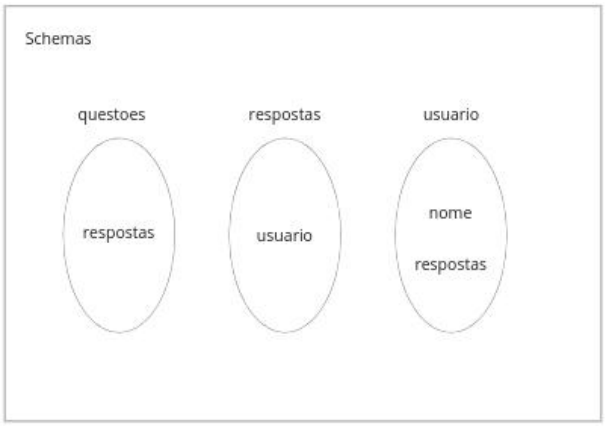


Figura 32 – Esquemas identificados para todos os objetos

Para fazer a identificação de esquema para um objeto, essa *API* utiliza um cálculo de pontuação de diferença entre todos os *Schemas* que possui e o objeto. Esse cálculo de pontuação é feito por meio da diferença entre os campos de todos os *Schemas* e os campos do objeto. Caso as pontuações sejam menores que um limite dado, o *Schema* de menor pontuação de diferença é atribuído ao objeto. O Algoritmo 2 detalha como isso ocorre.

Algorithm 2 Algoritmo de pontuacao de diferença

```
1:  procedure IdentificarSchema(atributosDicionario)
2:      schemaMaisSimilar ← None
3:      for all schema ∈ self.schemas do
4:          attr_do_schema ← set(schema.column_information.keys())
5:          attr_do_dicionario ← set(atributosDicionario)
6:          attr_unicos ← attr_do_schema.union(attr_do_dicionario)
7:          diff_sch_dict ← attr_do_schema − attr_do_dicionario
8:          diff_dict_sch ← attr_do_dicionario − attr_do_schema
9:          attr_diferentes ← diff_sch_dict.union(diff_dict_sch)
10:         pontuacao ← attr_diferentes.tamanho/attr_unicos.tamanho
11:         if pontuacao <= pontuacao_maxima e (schemaMaisSimilar! = None ou
    pontuacao <= schemaMaisSimilar[1]) then
12:             schemaMaisSimilar ← (schema, pontuacao)
13:         end if
14:     end for
15:     if schemaMaisSimilar! = None then
16:         return schemaMaisSimilar[0]
17:     else
18:         return None
19:     end if
20: end procedure
```

Para dar inicio ao cálculo de pontuação, é necessário que sejam fornecidos os atributos do dicionário para a função de cálculo, conforme mostra a linha 1. Na linha 2, é

criada uma variável para armazenar o *Schema* mais similar, que pode ser identificado no próximo passo.

Das linhas 3 a 14 é feito o cálculo da pontuação. Para isso, a *SchemaAPI* possui um atributo *schemas*, que armazena todos os *Schemas* criados, sendo eles representados na Figura 32. Na linha 3 ele itera sobre todos esses *Schemas* que ela possui e na linha 4 ele coleta as informações de atributos desse esquema e colocar em um *set*. No Python, *set* é uma coleção desordenada de elementos únicos e é utilizada por suportar operações matemáticas como união, interseção e diferença. Na linha 5, as informações de atributos do dicionário também são adicionadas em um *set*.

A linha 6 cria uma união dos atributos do *set* do *Schema* e do *set* dos atributos do dicionário. Dessa forma, são obtidos todos os atributos únicos entre ambos os *sets*. Nas linhas 7 e 8 é calculada a diferença, sendo a primeira dos atributos de esquema e dicionário e depois do dicionário e esquema. Na linha 9 é feita a união das diferenças para que seja possível calcular a quantidade de atributos que não estão presentes entre o esquema e o dicionário.

Na linha 10 é calculada a pontuação, sendo ela o tamanho das diferenças dividido pelo tamanho de atributos únicos entre os *sets*. Na linha 11 essa pontuação é utilizada para se comparar com o esquema presente na variável de *Schema* mais similar, sendo que na ausência de um esquema ou se a pontuação calculada for menor que a presente na variável, este esquema é atribuído na variável junto com a sua pontuação. As demais linhas apenas retornam como resultado *None*, caso não haja nenhum esquema que obteve pontuação menor que a máxima, ou o esquema que obteve menor pontuação.

No caso do exemplo das **questoes**, **questoes** possui o campo *respostas*, **respostas** possui como campos apenas *usuario*, enquanto que **usuario** possui como campos *nome* e *respostas*. Quando a *API* recebe um dicionário **respostas**, ela analisa todos os *Schemas* que possui e verifica as diferenças entre os campos. A Figura 33 mostra o algoritmo de pontuação aplicado ao dicionário. Como o dicionário fornecido à *API* possui a pontuação de 0 (0% de diferença) com o *Schema respostas*, a sua identificação será de **respostas**.

Figura 33 – Pontuação para identificar um dicionário respostas

Na Figura 34 está a pontuação resultante do dicionário **usuario**. Como o esquema **usuario** possui a pontuação de 0, enquanto que **questoes** tem a pontuação de 0,5, o *Schema* será **usuario**.

O *Schema* **questoes** recebe 0,5 de diferença porque a diferença entre **questoes** e o objeto é de 0 (o atributo respostas está em ambos), a diferença entre o objeto e **questoes** é de 1 (o atributo nome não está no *set* de atributos de **questoes**) e a quantidade de atributos únicos entre ambos é de 2 (nome e respostas). Assim, a pontuação se dá pelo cálculo $(0 + 1)/2$.

Por fim, a descoberta de esquema é dada pela pontuação da *Schema API*. Caso a pontuação de diferença seja suficiente, os metadados são analisados para identificar quais são suas diferenças. As diferenças são mescladas com o esquema atual de forma incremental, ou seja, colunas que não estão presentes são tratadas como de valor nulo e novas colunas são criadas.

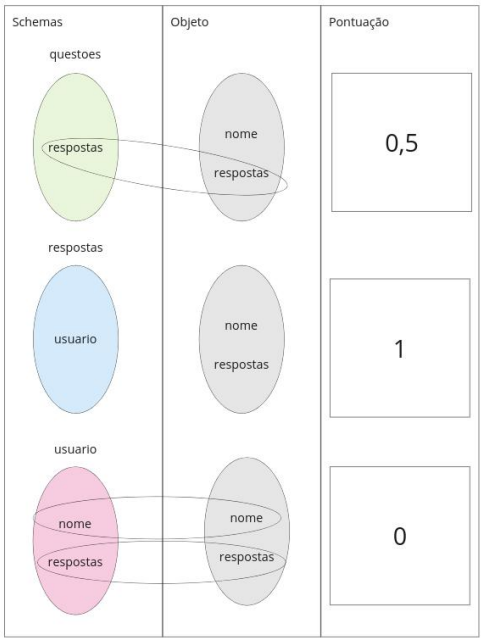


Figura 34 – Pontuação para identificar um objeto usuario

4.1.4 Neo4JClient

O componente cliente do Neo4J realiza as escritas necessárias no banco de dados. Ele utiliza a estrutura de nó interna (*Node*), já que nela já existem todas as informações necessárias para a criação dos nós e de suas relações. O Algoritmo 3 demonstra de que forma a escrita é feita no Neo4j.

Algorithm 3 Algoritmo de escrita de nós no Neo4j

1: $labels \leftarrow emptyDict$
2: **for all** $node \in nodes$ **do**
3: $\quad labels[node.table.name] \leftarrow node$
4: **end for**
5: **for all** $label, nodes \in labels.items()$ **do**
6: $\quad yield\ gerarQueryParaRotulo(label, nodes)$
7: **end for**

Conforme demonstram as linhas 1, 2, 3 e 4, os *Nodes* são primeiros separados por rótulos, sendo eles o nome da *Table* atribuído. Depois, para todos eles são criadas consultas de escrita no *Neo4j*, assim permitindo que sejam escritos múltiplos nós com a mesma consulta, já que o rótulo é inserido na consulta por meio de interpolação de *string*. Interpolação de *string* neste contexto é uma técnica que permite incluir variáveis ou expressões dentro de uma *string*, sendo um exemplo demonstrado na Figura 35.

Como é necessário criar primeiro o nó para depois estabelecer suas relações, foi adotada a estratégia de criar todos os nós primeiro. Dessa forma é garantido que sempre existem os nós de origem e destino.

A criação de nós em lote utiliza os mecanismos do Neo4J para que tudo seja criado de uma só vez. Para isso foi desenvolvido um *template*, conforme mostra a Figura 35, onde o rótulo do nó é o nome da tabela e os valores são fornecidos de uma só vez por meio de uma lista de dicionário, sendo a chave o nome da coluna e o valor é o seu valor. Um exemplo de código gerado para o Neo4j a partir deste *template* é mostrado na Figura 36.

```
return (
f"""
    WITH $nodes AS nodes
    UNWIND nodes AS node
    MERGE (n:{self.scape_label(label)} {{__jtg_id: node.id}})
    ON CREATE\n {row_query}
    """,
    [{"id": node.id, **node.common_values} for node in nodes]
)
```

Figura 35 – Template da consulta de criação de nós em lote

```
WITH $nodes AS nodes
UNWIND nodes AS node
MERGE (n:headline {__jtg_id: node.id})
ON CREATE
  SET n.main = node.main,
    n.kicker = node.kicker,
    n.content_kicker = node.content_kicker,
    n.print_headline = node.print_headline,
    n.name = node.name,
    n.seo = node.seo,
    n.sub = node.sub
```

Figura 36 – Exemplo de consulta de criação de nós em lote

Conforme mostra a Figura 37, para a criação das relações é utilizada uma consulta com os comandos *MATCH* e *MERGE*, sendo o *MATCH* realizado com os identificadores internos (chamados de ___jtg_id), *n* o nome do nó origem e *n1* o nome do nó destino, seguido do *MERGE* entre eles. No caso do comando *MERGE*, a função entre chaves que é utilizada para especificar o rótulo, chamada de *self.scape_label* realiza uma interpolação de *string*, já que é uma função executada dentro de uma *string*. Um exemplo de resultado é mostrado na Figura 38.

```
for node in nodes:
    for relation in node.relations:
        yield f"""
        MATCH (n :{node.table.name} {{__jtg_id: {node.id}}})
        MATCH (n1 :{relation.table.name} {{__jtg_id: {relation.id}}})
        MERGE (n)-[:{self.scape_label(relation.table.name)}]->(n1)
    """
```

Figura 37 – Template da consulta de criação de relações entre os nós

```
MATCH (n :nyt {__jtg_id: 630})
MATCH (n1 :byline {__jtg_id: 712})
MERGE (n)-[:byline]->(n1)
```

Figura 38 – Exemplo de consulta de criação de relações entre os nós

4.1.4.1 Regras de Mapeamento

O processo de mapeamento de objetos JSON para o esquema BD de grafo foi explicado no Algoritmo 1 e culminou no processo descrito no Algoritmo 3. Este processo aplica o seguinte conjunto de regras de mapeamento, que é uma adaptação das regras apresentadas em (ALOTAIBI; PARDEDE, 2019):

- Um objeto JSON mantido em uma coleção de nome C_x se torna um nó cujo rótulo é C_x;
- Um atributo simples de um objeto JSON O_i se torna uma propriedade do nó correspondente à O_i;
- Um atributo a_n de um objeto JSON O_i que mantém um objeto aninhado gera:
  - Um nó O_j cujo rótulo é o nome de a_n;
  - Uma aresta direcionada de O_i e O_j.

Um exemplo de mapeamento é mostrado na Figura 39 para o documento *JSON* de *questoes* mostrado na Figura 21.

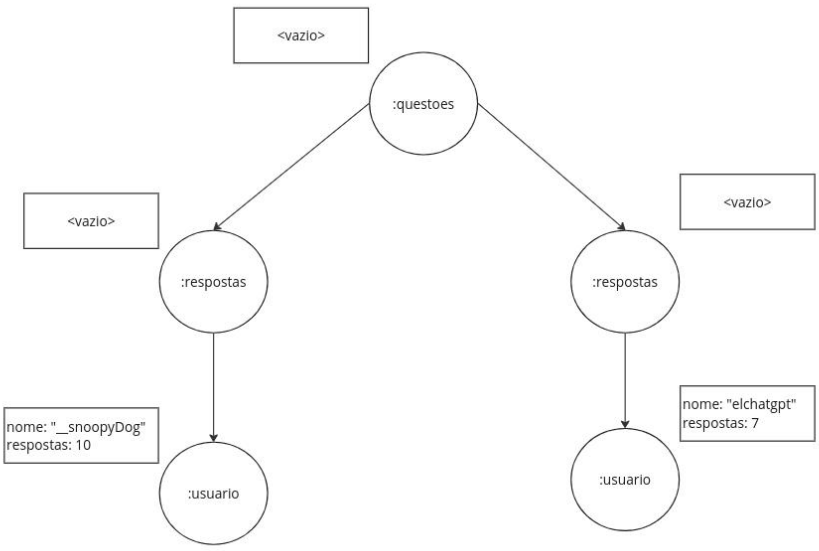


Figura 39 – Mapeamento do documento JSON para nós do Neo4J

Esta solução realiza a escrita de nós por meio da árvore criada a partir dos objetos JSON aninhados, conforme descrito na Seção 4.1.2. O Algoritmo 4 aplica as regras de mapeamento. Para os dados de *questoes* (Figura 21), o nó patriarca é o primeiro a ser escrito no *Neo4j*. Na Figura 31 estão todos os *Nodes* resultantes do processamento do objeto JSON, sendo o patriarca (**questoes**) detentor do id 1, as **respostas** id 2 e o **usuario** id 3. Essa árvore de *Node* é utilizada como ponto de partida para a escrita dos nós e arestas.

Algorithm 4 Algoritmo para escrita de nós da árvore de Nodes

```
1: ClienteNeo4j = Neo4j()
2: procedure EscreverNos(nodePai)
3:     ClienteNeo4j.escreverNo(nodePai)
4:     nosAninhados ← nodePai.relations
5:     for all noAninhado ∈ nosAninhados do
6:         EscreverNos(noAninhado)
7:         ClienteNeo4j.criarArestas(nodePai, noAninhado)
8:     end for
9: end procedure
```

Pela linha 3 do Algoritmo 4, as **questões** são transformadas em um nó sem nenhuma propriedade, já que não existe nenhum atributo simples no objeto. Após isso, conforme mostram as linhas 4 e 5, todos os *Nodes* aninhados no atributo *relations*, que no exemplo são as **respostas**, são transformados em nós e tem suas arestas geradas. O algoritmo, por ser recursivo (linha 6), percorre toda a estrutura aninhada JSON gerando os demais nós e arestas do grafo. O resultado é o apresentado na Figura 39.

4.2 IMPLEMENTAÇÃO DA FERRAMENTA

A implementação da ferramenta utiliza dos mesmos princípios e componentes descritos anteriormente para a arquitetura de base. A principal diferença está na forma como as execuções acontecem e na separação dos componentes de escrita no Neo4j e criação de arestas. Esta arquitetura é orientada a eventos, utilizando estratégias para execução paralela dos componentes por meio de outros processos gerenciados pelo *framework Ray*, e utilizando filas para o balanceamento do trabalho, conforme mostra a Figura 40.

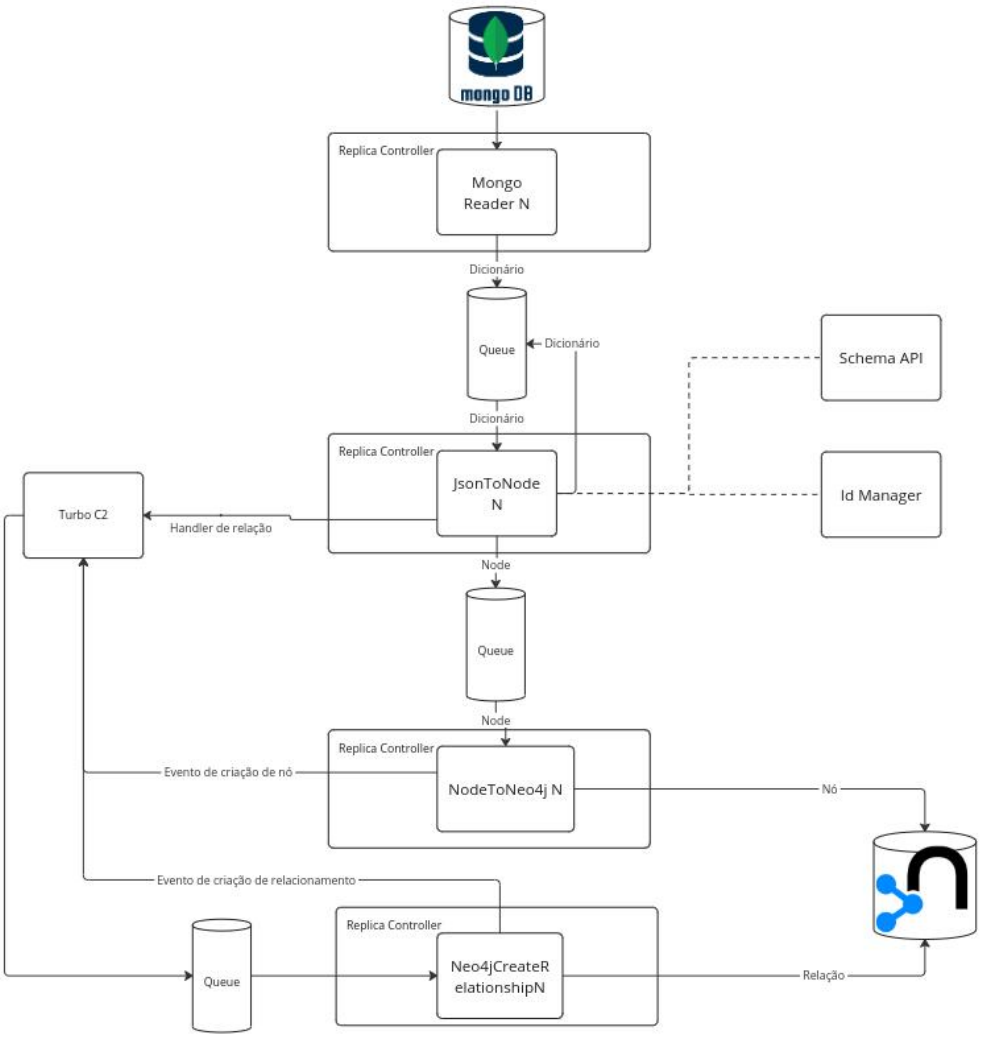


Figura 40 – Arquitetura distribuída e escalável baseada em eventos

Para viabilizar essa arquitetura foi desenvolvido, por intermédio do *framework Ray*, um *framework* de orquestração distribuído que executa códigos em Python de forma assíncrona, paralela e concorrente, com fácil configuração de escalabilidade. Este *framework*, chamado de *Turbo C2*[1], é utilizado pela solução de mapeamento proposta neste trabalho, sendo disponibilizado como código aberto.

4.2.1 Turbo C2

O *framework Turbo C2* utiliza o modelo de atores para distribuição de trabalho e assim auxiliar a criação de algoritmos escaláveis. Para isso, ele disponibiliza uma *API* baseada em decoradores da linguagem *Python* para sinalizar a existência de um ator, que pode realizar computações e gerenciar estado interno. Esse ator recebe o nome de *job*. O

[1] https://github.com/gorgonun/turbo-c2

decorador neste contexto é uma função em *Python* que estende o comportamento de outra função sem alterar seu código.

Ele também disponibiliza uma *API* para declaração de *endpoints HTTP* com o auxilio das bibliotecas *FastApi* e *Pydantic*, além da biblioteca *Ray Serve* para a distribuição e controle de réplicas, conforme mostra a Figura 41. Para a coleta de métricas de recursos utilizados, como CPU e memória, e para alimentar os *dashboards* com a contagem de nós e arestas, foi utilizada a ferramenta *Prometheus*[2] devido a sua integração nativa com o *framework Ray* e a sua facilidade de uso. *Prometheus* é uma ferramenta de monitoramento e alertas de código aberto que é bastante utilizada em ambientes de microsserviços e *Kubernetes*[3]. Já *Kubernetes* é uma plataforma de código aberto para automação de implantação, dimensionamento e gerenciamento de aplicações em contêineres, sendo que o *framework Ray* também oferece suporte para sua utilização, caso necessário.

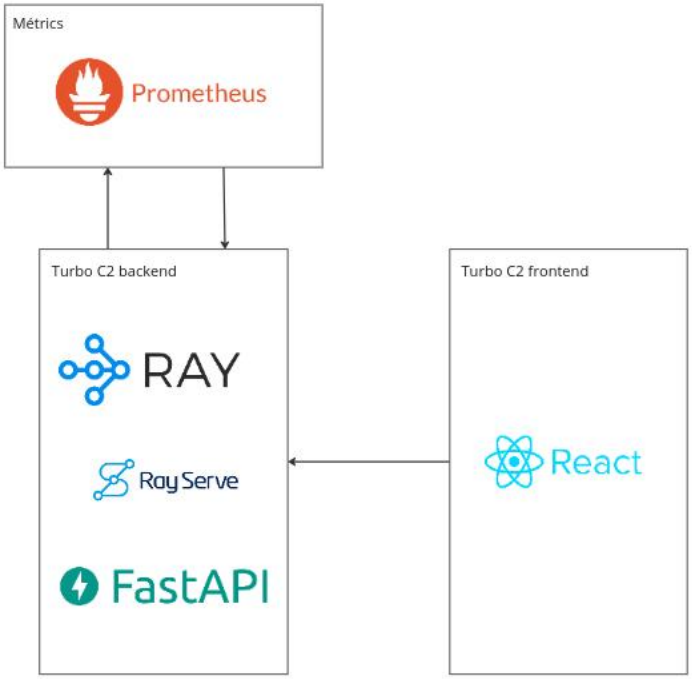


Figura 41 – Arquitetura de alto nível do Turbo C2

A seguir são explicados alguns conceitos criados pelo *framework* para possibilitar a execução dos componentes do *Json2Node* de forma paralela e distribuída utilizando o modelo de atores, sendo eles os **jobs**, **filas** e **handlers**. Para simplificar a explicação, é descrito um exemplo de uso que mostra uma aplicação mais familiar das ferramentas e posteriormente é explicado como elas foram utilizadas para o contexto deste trabalho.

O exemplo de uso segue o seguinte contexto: uma faculdade deseja implementar um sistema que notifica os alunos do curso *TCC 1* sobre a chegada do prazo de postagem

[2] https://prometheus.io/
[3] https://kubernetes.io/pt-br/

do resumo de TCC. Para isso, é desejado que haja uma verificação diária do prazo e se for analisado que falta uma semana para o final, deve-se enviar um e-mail para o grupo *alunos_de_tcc_1* todos os dias até a chegada do prazo.

A Figura 42 demonstra um *job* criado para o exemplo, sendo o papel deste *job* verificar diariamente se chegou o período de envio de e-mails, sendo este período de 7 a 1 dia antes do prazo final. Caso o período tenha chegado, ele deve avisar que os e-mails devem ser enviados.

```
1   # ------------------------------------- Job que monitora data -------------------------------------
2   @job(
3       wait_time=24 * 60 * 60,
4       replicas=1,
5       meta={"dispatches": [PrazoParaPostagemDeRelatorioDeTCC1Chegando]},
6       prazo_final=datetime.datetime.fromisoformat("2024-05-01T23:59:59"),
7       data_para_enviar_email=datetime.datetime.fromisoformat("2024-04-24T00:00:00"),
8   )
9   def monitorar_prazo_para_postagem_de_relatorio_de_tcc_1(
10      prazo_final: datetime.datetime,
11      data_para_enviar_email: datetime.datetime,
12  ):
13      agora = datetime.datetime.now()
14
15      if agora >= data_para_enviar_email and agora < prazo_final:
16          return PrazoParaPostagemDeRelatorioDeTCC1Chegando(prazo_final)
17
```

Figura 42 – Job que realiza o monitoramento do prazo

Um *job* é criado por meio de um decorador, chamado *job*, aplicado em uma função *Python*. Esse decorador contém diversas informações referentes ao contexto de execução da função e o nome da fila de entrada de informação (se houver) e as filas por onde vai ocorrer a saída da informação (se houver). Ele é um *ator* que executa constantemente de forma paralela, sendo que isso se traduz em outro processo *Python*.

Para o exemplo, serão necessários dois *jobs*, sendo um deles presente na Figura 42. Conforme é visível na linha 2, temos o decorador *@job*, que se localiza acima da função que irá realizar o monitoramento.

As linhas 3, 4 e 5 apresentam algumas configurações disponibilizadas pelo *job*, sendo elas o **wait_time**, que indica que o *job* deve executar 1 vez por dia (a cada 86400 segundos), em uma réplica (uma execução paralela a outros *jobs*), conforme mostra a linha 4, e a linha 5 indica que o *job* despacha um evento, sendo ele chamado de **PrazoParaPostagemDeRelatorioDeTCC1Chegando**. Este evento indica que o prazo para a postagem do relatório está chegando, e assim deve-se enviar o e-mail para os alunos.

As linhas 6 e 7 são parâmetros customizados da função. Tanto **prazo_final** quanto **data_para_enviar_email** são declarados nessas linhas e serão repassados para a função durante a execução, conforme mostram as linhas 10 e 11. O prazo final foi definido como 01/05/2024 às 23:59:59 enquanto que a data para começar a enviar os e-mails está como 24/04/2024 às 00:00:00.

Por fim, nas linhas 13, 14, 15 e 16 é feita a verificação se o dia de hoje é maior ou

igual a data para iniciar o envio dos e-mails. Se for e se não houver chegado o prazo final, na linha 16 ele despacha o evento **PrazoParaPostagemDeRelatorioDeTCC1Chegando** junto com o prazo final. Em um *job* é possível despachar um evento apenas o retornando como resultado da função. Este evento pode ser observado na Figura 43.

```
1   # ------------------------ Evento ------------------------
2   @dataclass(frozen=True)
3   class PrazoParaPostagemDeRelatorioDeTCC1Chegando(Event):
4       prazo: datetime.datetime
```

Figura 43 – Evento PrazoParaPostagemDeRelatorioDeTCC1Chegando

Uma fila, por sua vez, pode ser criada de forma natural, quando referenciada em uma instancia ou no encadeamento de *jobs*, ou por meio do decorador *@queue*. As filas estão abstraídas em uma *API* própria utilizando o conceito de *atores*, sendo cada fila um *ator*, que pode executar em um processo separado ou até mesmo em outra máquina. A Figura 44 mostra como ocorre a relação entre filas e *jobs*, sendo que um *job* pode enviar dados para uma fila e também consumir dela.

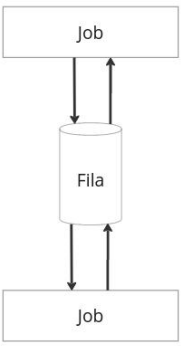


Figura 44 – Relação entre filas e jobs

Já um *handler* pode ser criado por meio de um decorador chamado *@when*. Ele é composto por uma sentença lógica, sendo ela composta por operadores de eventos ou um predicado (caso seja atômica).

Ele pode estar associado a comportamentos quando a sentença é verdadeira ou falsa. Para que a sentença seja avaliada, é necessário que um evento que possua dados iguais aos definidos de forma explícita na referencia aconteça. Caso todos os eventos aconteçam, a sentença é avaliada como verdadeira e as instruções definidas são executadas.

A Figura 45 mostra como uma condição booleana pode ser transformada em um *handler*. Dado um evento **A**, um evento **B** e uma função **x**, pode-se colocar as condições

A e **B** no decorador *@when* e definir o comportamento na função logo abaixo para quando os eventos *A* e *B* acontecerem. O símbolo **&** representa **AND**.

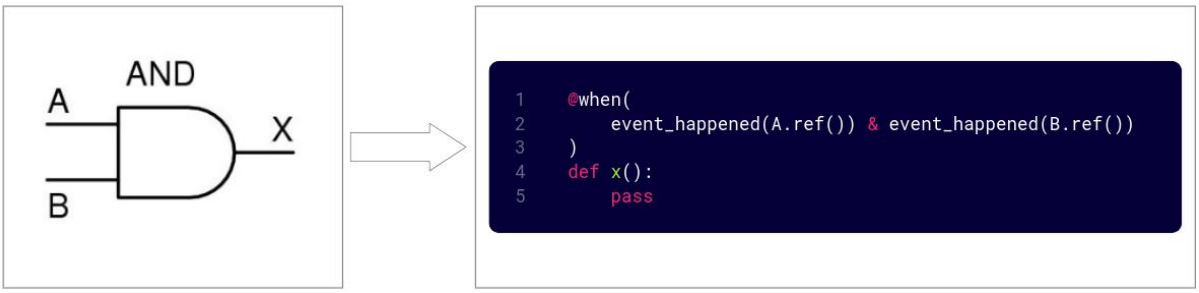


Figura 45 – Tradução de uma condição booleana para um *handler*

Voltando para o exemplo, a Figura 46 mostra um *handler* criado para lidar com o evento despachado do primeiro *job*, que monitora a data para postagem final. Depois que o evento **PrazoParaPostagemDeRelatorioDeTCC1Chegando** é despachado, conforme mostra a linha 20 da Figura 42, ele aciona o *handler* **enviar_emails_para_alunos_de_tcc_1** para que seja executada as instruções definidas.

```
1  # ------------------------------------ Handler ------------------------------------
2  @when(
3      event_happened(PrazoParaPostagemDeRelatorioDeTCC1Chegando.ref()),
4      outputs=["fila_de_envio_de_notificacao_para_alunos_de_tcc_1"],
5  )
6  def enviar_emails_para_alunos_de_tcc_1(
7      ultimo_evento: PrazoParaPostagemDeRelatorioDeTCC1Chegando,
8      eventos: list[PrazoParaPostagemDeRelatorioDeTCC1Chegando],
9  ):
10     return EmailParaAlunos(
11         grupo="alunos_de_tcc_1",
12         assunto=f"Prazo para postagem de relatório de TCC 1 chegando, fique atento! O prazo final é dia
               {ultimo_evento.prazo.isoformat()}",
13     )
```

Figura 46 – *Handler enviar_emails_para_alunos_de_tcc_1*

Na linha 2 da Figura 46 temos o decorador *@when*, que é utilizado para declarar um *handler*. Na linha 3 está presente um predicado, que deve ser interpretado como: *quando o evento PrazoParaPostagemDeRelatorioDeTCC1Chegando acontecer, execute*. A linha 4 indica que o valor resultante da execução da instrução deve ser enviado para a fila *fila_de_envio_de_notificacao_para_alunos_de_tcc_1*. Esta fila, por estar sendo referenciada, é criada automaticamente com as definições padrão.

As linhas 6, 7 e 8 declaram a instrução a ser executada caso o predicado seja verdadeiro. Essa instrução recebe de forma obrigatória o último evento que aconteceu e todos os eventos ligados ao predicado. Por fim, nas linhas 10, 11, 12 e 13 a instrução retorna

um objeto **EmailParaAlunos**, que é demonstrado na Figura 47. Este objeto contém os dados para envio do e-mail, sendo eles o grupo **alunos_de_tcc_1** e o assunto.

```
1    # ---- Instruções para email ----
2    @dataclass
3    class EmailParaAlunos:
4        grupo: str
5        assunto: str
```

Figura 47 – Objeto EmailParaAlunos

Por fim, o *job* demonstrado na Figura 48 é um pouco mais simples que o primeiro, já que sua função é apenas ler da fila de e-mails a serem enviados e realizar o envio por meio de um cliente de e-mail.

```
1   # ---------------------------------------- Job que envia email ----------------------------------------
2   @job(
3       input_queue_reference="fila_de_envio_de_notificacao_para_alunos_de_tcc_1",
4   )
5   async def enviar_email_para_alunos(fself, content: EmailParaAlunos, on_first_run):
6       @on_first_run
7       async def __on_first_run(fself, content: EmailParaAlunos, on_first_run):
8           fself["cliente_de_email"] = ClienteDeEmail()
9
10      await __on_first_run()
11
12      fself["cliente_de_email"].enviar_email(content.grupo, content.assunto)
```

Figura 48 – Job que faz o envio do e-mail para os alunos

A linha 2 possui o decorador *job*, que indica que a função é um ator *job*. A linha 3 indica que este *job* consome de uma fila chamada de **fila_de_envio_de_notificacao_para_alunos_de_tcc_1**, sendo ela a mesma fila presente no *handler*, que irá alimentar ela, conforme mostra a Figura 46. Sempre que a fila for alimentada, este *job* recebe o seu conteúdo por meio do parâmetro *content*, conforme mostra a linha 5. Além deste parâmetro, as linhas 6, 7 e 8 mostram os preparativos do estado do ator, sendo que é utilizado o decorador *on_first_run* para indicar uma função que será executada apenas na primeira vez que o *job* executar e ela cria uma instância do cliente de e-mail e salvar em seu estado interno. Por fim, na linha 12, o *job* encerra sua execução realizando o envio do e-mail para o grupo contido no objeto **EmailParaAlunos** criado no *handler* que alimentou a fila, junto com o assunto.

A solução completa para o exemplo propõe o *job* **monitorar_prazo_para_postagem_de_relatorio_de_tcc_1**, demonstrado na Figura 42, que diariamente verifica se a data está entre 1 a 7 dias antes do prazo final da entrega do relatório. Se estiver, ele dispara o evento **PrazoParaPostagemDeRelatorioDeTCC1Chegando** contendo

o prazo final, que é recebido pelo *handler* **enviar_emails_para_alunos_de_tcc_1**, onde a informação do prazo é adicionada no assunto do objeto **EmailParaAlunos** junto com o grupo de e-mail *alunos_de_tcc_1*. Esse objeto é enviado para a fila *fila_de_envio_de_notificacao_para_alunos_de_tcc_1* que o job **enviar_email_para_alunos** consome, que por meio do grupo de e-mail e assunto irá enviar o e-mail para os alunos do grupo. A Figura 49 mostra a arquitetura final do exemplo.

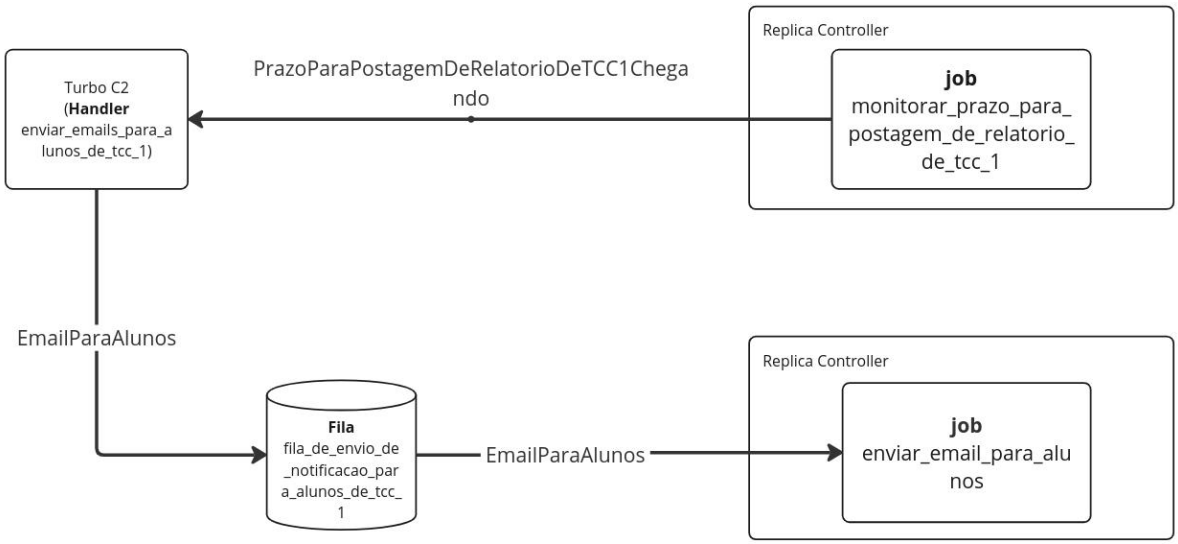


Figura 49 – Arquitetura do exemplo

Na próxima seção é explicado como esses três conceitos serão utilizados no trabalho para possibilitar a execução do processamento de forma paralela e em *streaming* mantendo o formato de *pipeline*.

4.2.2 Execução baseada em eventos

Por meio dos conceitos apresentados na seção anterior foi criada uma estratégia para processar os documentos JSON de forma paralela, com *streaming* dos dados em um *pipeline*. Essa estratégia foi implementada transformando os componentes originais em atores, seguindo o modelo de atores.

Conforme descrito anteriormente, pode-se utilizar um *job* para executar computação e armazenar estado. Assim, os componentes originais **MongoDB Reader** e **JsonToNode** se transformaram em um *job*, enquanto que o **Neo4jClient** foi utilizado por meio de dois outros componentes, no formato de *jobs*, chamados de **NodeToNeo4j**, que realiza a escrita de nós no *Neo4j*, e **Neo4jCreateRelationship**, que realiza a escrita de arestas no *Neo4j*. Essa separação entre o tempo de escrita de nós e arestas acontece para possibilitar o processamento de objetos de forma assíncrona, sendo criado um *handler* que coordena a escrita assíncrona de arestas.

Na Figura 40 estão presentes todos esses componentes, sendo que entre cada *job* está presente uma fila para balancear a carga de trabalho e possibilitar a execução paralela de computação entre os dados. Além disso, existem os componentes **Schema API**, que se transformou em um ator, e um novo componente chamado de **Id Manager**, que também é um ator. Por fim, o **Turbo C2** está presente para realizar o gerenciamento de criação de *handlers* e processamento de eventos, que são despachados no **NodeToNeo4j** e **Neo4jCreateRelationship**.

Para possibilitar a escrita assíncrona de nós e posteriormente a criação de arestas, o Algoritmo 1 foi modificado. Anteriormente ele executava de forma recursiva até que todos os objetos aninhados do documento JSON em formato de dicionário fossem processados, assim criando uma árvore de *Nodes*. Este comportamento pode ser perigoso no caso de objetos JSON gigantes, que podem ocupar completamente o processamento dos *jobs* disponíveis e até provocar falhas devido ao uso excessivo de memória. O novo Algoritmo 5, mostrado a seguir, substitui a recursividade por uma retroalimentação de todos os objetos aninhados, sendo que em conjunto com a retroalimentação é criado um *handler* que liga o objeto pai ao filho.

Algorithm 5 Algoritmo de mapeamento dicionário para Node baseado em eventos

```
1: procedure MigrarDicionario(dicionario, nivel, campoPai, schemaApi)
2:     identificadorUnicoDoNo ← pegarProximoIndentificador()
3:     if nivel == 0 then
4:         if tabelaRaiznaoexiste then
5:             tabelaRaiz = CriarTabela(nomeColecao, schemaApi.novoSchemaVazio())
6:         end if
7:         schemaApi.atualizarSchema(tabelaRaiz.schema, dicionario)
8:         node ← Node(identificadorUnicoDoNo, tabelaRaiz)
9:     else
10:        schema ← schemaApi.identificarSchema(dicionario)
11:        tabela ← pegarTabelaOuCriar(campoPai, schema)
12:        node ← Node(identificadorUnicoDoNo, tabela)
13:    end if
14:    for all campo ∈ dicionario do
15:        if Tipo(campo) == dicionario ou Tipo(campo) == lista[dicionario] then
16:            node_referencia_de_relacao ← retroalimentarECriarHandler(campo,
   nivel + 1, campo.nome)
17:            node.relationship_references ← node_referencia_de_relacao
18:        else
19:            node.common_values[campo.nome] ← campo
20:        end if
21:    end for
22:    return node
23: end procedure
```

O Algoritmo 5 não possui muitas diferenças em relação ao Algoritmo 1, sendo

apenas as linhas 16 e 17 modificadas. Antes, o algoritmo recebia um *Node* como resultado das recursões sobre os objetos aninhados. Agora, ele recebe uma referência de relação, sendo esta referência um objeto *AsyncRelationshipReference*. Este objeto possui apenas um atributo, sendo ele um *UUID* que é utilizado para criar um novo *handler*, conforme mostra a Figura 50.

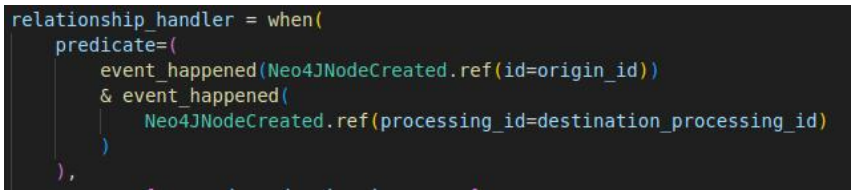


Figura 50 – Configuração de gatilho para criação de relações de forma assíncrona

O *handler* que está presente na Figura 50 estabelece uma ligação entre eventos de *Neo4JNodeCreated*. Este evento é despachado pelo componente **NodeToNeo4j** após ser realizada a escrita do nó, sendo ele demonstrado na Figura 51.

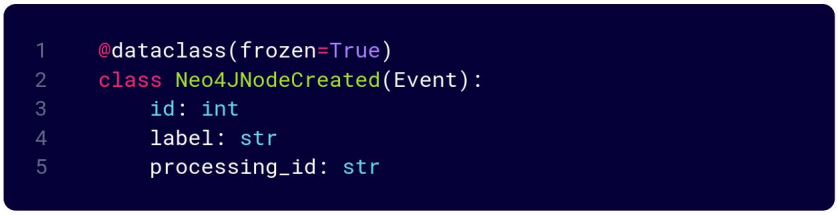


Figura 51 – Evento de criação de nó

Esta ligação dos eventos se dá por um evento identificado pelo campo **id** e o outro pelo campo **processing_id**. Ela é necessária porque o *Node* pai não sabe o **id** do *Node* filho, já que ele foi retroalimentado na fila de entrada do *JsonToNode* e só é processado posteriormente. Ele apenas possui a sua *AsyncRelationshipReference*, conforme mostra a linha 17. Com a posse dela, ele pode criar uma ligação entre seu próprio **id** e a *AsyncRelationshipReference*, que será utilizada no campo **processing_id**. O campo **id** é o identificador único do nó, que é criado na linha 2 do Algoritmo 5. Já o **processing_id** é o *UUID* dado na linha 16 do algoritmo.

A Figura 31 demonstra o resultado antes das modificações e a Figura 52 mostra o resultado depois das modificações. A principal diferença está na linha 23, onde a lista de *relations* está vazia, e as linhas 17 a 22 demonstram que o objeto aninhado **respostas** está referenciado com uma lista de *AsyncRelationshipReference*, sendo que cada um dos itens possui um *UUID* diferente.

```
Node(
    id=1,
    table=Table(
        id=1,
        name="questoes",
        schema=Schema(
            id=1,
            column_information={
                "respostas": Column(
                    name="respostas",
                    is_optional=True,
                    is_relationship=True,
                )
            },
        ),
    ),
    relationship_references={
        "respostas": [
            AsyncRelationshipReference("6d2560f31a654f32b25b64c7ef11398c"),
            AsyncRelationshipReference("a2e41e4860bd405f874f42798c3d698c")
        ]
    },
    relations=[],
)
```

Figura 52 – Resultado da substituição de recursividade por retroalimentação

Dessa forma, o *handler* deve ser interpretado dessa forma: "quando o evento *Neo4JNodeCreated* for despachado com o *id* referente ao nó de origem (pai) da aresta **e** quando o evento de *Neo4JNodeCreated* for despachado com o *processing_id* referente ao *UUID* do nó de destino (filho), execute as seguintes instruções.".

Tendo como base a Figura 52, esta sentença se tornaria (para a primeira resposta): "quando o evento *Neo4JNodeCreated* for despachado com o *id* 1 (questoes) **e** quando o evento de *Neo4JNodeCreated* for despachado com o *processing_id* 6d2560f31a654f32b25b64c7ef11398c (resposta 1), execute as seguintes instruções."

A instrução que é executada quando o *handler* for verdadeiro é a de enviar os dados de aresta entre os dois *Nodes* para o componente *Neo4jCreateRelationship*, sendo este comportamento ilustrado na Figura 53.

```
def crie_um_novo_relacionamento_entre_os_nos(
    last_event: Neo4JNodeCreated, events:
        list[Neo4JNodeCreated]
):
    return Neo4JRelationshipData(
        origin_id=origin_id,
        origin_label=origin_label,
        destination_id=last_event.id,
        destination_label=last_event.label,
        relationship_name=relationship_name,
    )

```

Figura 53 – Instruções executadas quando o handler é verdadeiro

Dessa forma, como podemos notar na Figura 40, não existe uma fila onde o componente **Neo4jCreateRelationship** recebe os dados do componente anterior, chamado de **NodeToNeo4j**, mas quando o *handler* executa as instruções, ele envia todas as informações necessárias para a criação de arestas por meio do *Turbo C2*. Quando as informações forem enviadas, o **Neo4jCreateRelationship** cria a aresta, que no exemplo estabelece uma ligação entre **questoes** e **respostas**.

Para exemplificar a forma como o *handler* deve se comportar e de que forma são criados os dados de arestas, na Figura 54 está uma representação visual do *handler* criado para esperar pelos eventos de criação de nó de **questoes** e **respostas**.

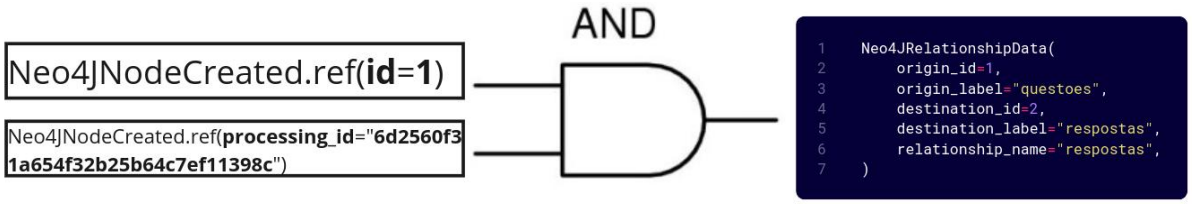


Figura 54 – Handler criado que espera pelos eventos de questoes e respostas

Para que haja o acionamento do *handler*, os eventos da Figura 55 e 56 devem ocorrer, sendo eles criados pelo componente **NodeToNeo4j**. O evento da Figura 55 é criado quando ocorrer a escrita do nó de *questoes*, sendo os seus dados descritos na Figura 52. Já o evento da Figura 56 ocorre quando o objeto aninhado *respostas* for obtido da fila e processado, da mesma forma que ocorreu com *questoes*.

```
Neo4JNodeCreated(
    id=1,
    label="questoes",
    processing_id=None
)
```

Figura 55 – Evento de criação do nó de questoes

```
Neo4JNodeCreated(
    id=2,
    label="respostas",
    processing_id="6d2560f31a654f32b25b64c7ef11398c"
)
```

Figura 56 – Evento de criação do nó de respostas

Além do componente **JsonToNode**, o componente *Schema API* sofreu algumas mudanças para se adequar à execução concorrente. Por conta de sua natureza de descoberta de novas informações e constante adaptabilidade, um esquema de uma tabela pode ser modificado por diferentes réplicas do componente *JsonToNode*. Assim, é necessário que exista um controle de acesso ao recurso para evitar problemas de sobrescrição.

Por conta disso, esse componente foi transformado em um ator. Assim, todas as réplicas do *JsonToNode* podem acessar os *Schemas* definidos e fazer modificações sem que as informações sejam perdidas. A Figura 57 demonstra como ela se transformou em um ator. Com o decorador *@ator* ela sofre as transformações necessárias, sendo que o componente não precisou ser modificado.

```python
@actor(
    name="schema_api",
    api=RemoteSchemaApiApi,
    api_identifier=JsonToGraphEnum.API_ID.value,
)
async def create_schema_api():
    return ActorDefinition(
        actor_class=SchemaAPI, kwargs={"max_difference_between_schemas": 0.2}
    )
```

Figura 57 – Transformação da instância de Schema API para ator

Por fim, para lidar com os identificadores únicos sequenciais, foi criado um componente chamado de **Id Manager** (ver Figura 40). Ele gerencia o contador de identificadores para cada objeto, assegurando que nunca haja identificadores repetidos.

4.3 APLICAÇÃO WEB PARA VISUALIZAÇÃO E GERENCIAMENTO DE MIGRAÇÕES

Para o gerenciamento de mapeamentos, migrações e análise das métricas, foi disponibilizada uma aplicação Web que se comunica com uma *API* presente na aplicação responsável por fazer as migrações. Essa aplicação é composta por duas páginas, uma contendo diversos painéis referente às métricas coletadas durante as migrações e a outra contendo ferramentas para visualização e gerenciamento de migrações.

4.3.1 Painéis

A página chamada de *dashboards* possui diversos painéis para visualização das métricas coletadas pelas migrações. Ela é composta por duas seções principais, sendo a primeira composta por um painel de métricas agregadas desde o inicio da coleta e a segunda composta por painéis de resumo das migrações por dia.

Os painéis disponíveis na aplicação Web recebem dados de métricas que são criados nos componentes *NodeToNeo4j* e *Neo4jCreateRelationship*. Estas métricas são referentes à escrita de nós no *Neo4j* e a criação de arestas.

A seção chamada de *All time* possui a subseção *Status*, que mostra as métricas de migração referentes ao momento atual. Conforme demonstra a Figura 58, o painel de *Status* da aplicação no momento do registro indicava que 12 nós estavam sendo criados no *Neo4j* por segundo e 2 arestas estavam também sendo criadas por segundo.

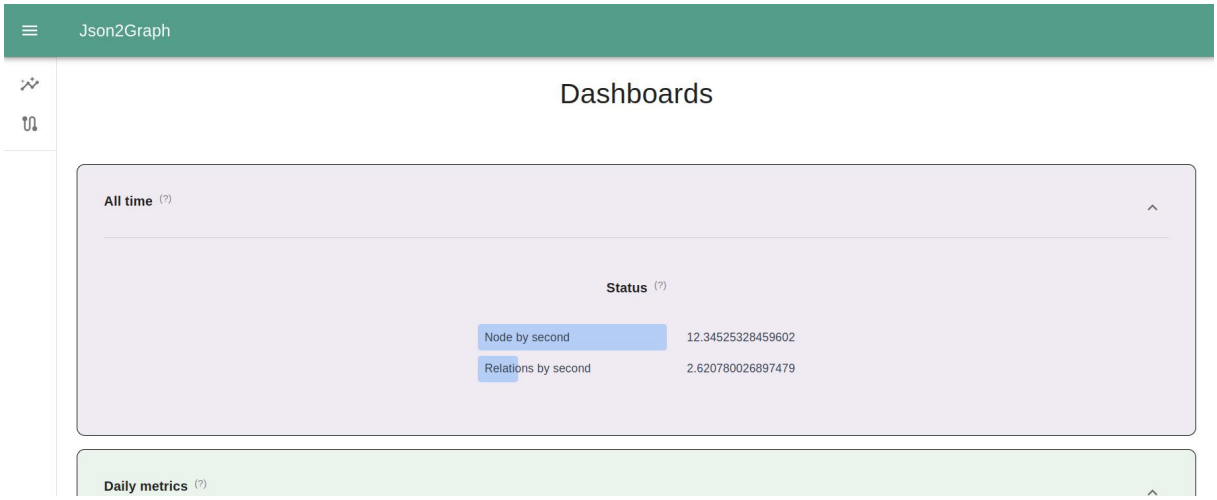


Figura 58 – Primeiro painel de métricas

A segunda seção, chamada de *Daily metrics*, possui painéis com métricas voltadas para o acompanhamento das migrações passadas e atuais. A subseção *Totals* possui o número total de métricas agregadas por migração no período de um dia e a subseção *Daily records migrated* apresenta um gráfico de linha representando o número de migrações finalizadas ao longo do tempo, com uma hora de intervalo entre os pontos durante uma semana.

Conforme mostra a Figura 59, a aplicação no momento do registro possuía diversas migrações acontecendo ao mesmo tempo, sendo os cinco primeiros cartões indicando o número de nós migrados e os três últimos indicando o número de arestas criadas. Para ilustração, o primeiro cartão indica que a migração *Default Migration2* escreveu 714 nós no *Neo4j* nas últimas 24 horas, enquanto que o sexto cartão indica que a migração *Default Migration0* escreveu 303 arestas nesse mesmo período.

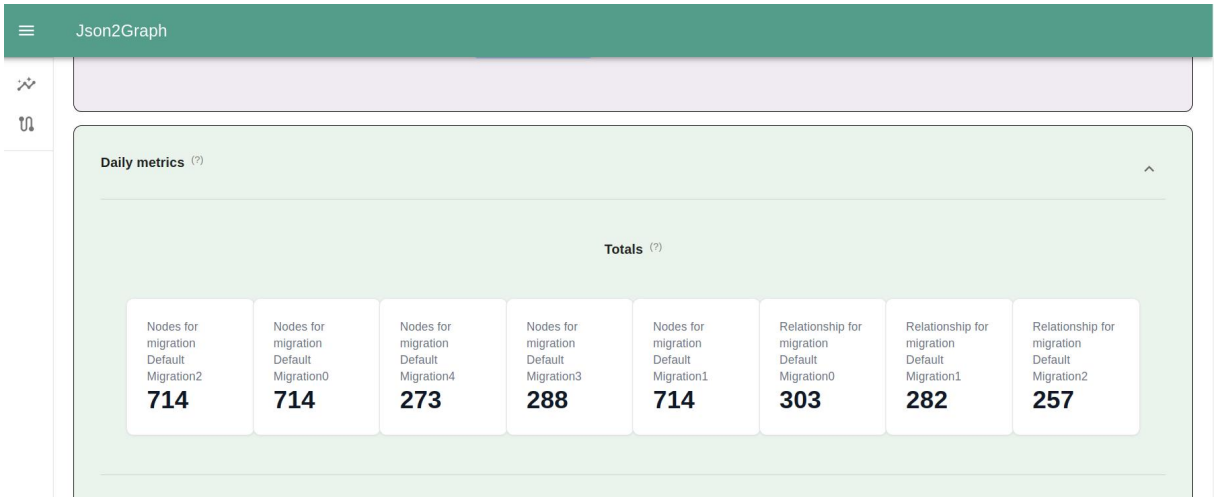


Figura 59 – Segundo painel de métricas

Conforme mostra a Figura 60, a aplicação possuía dados de 7 migrações ao longo de uma semana. Na Figura 61 é possível notar que no dia 5 de junho haviam 3 migrações e no intervalo de 1 hora elas possuíam os valores de 582, 553 e 530 migrações finalizadas.

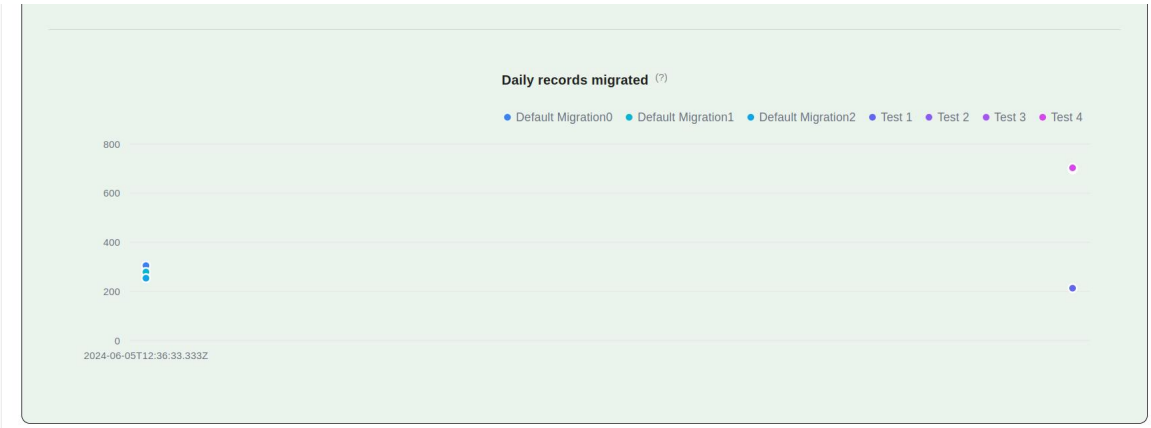


Figura 60 – Terceiro painel de métricas

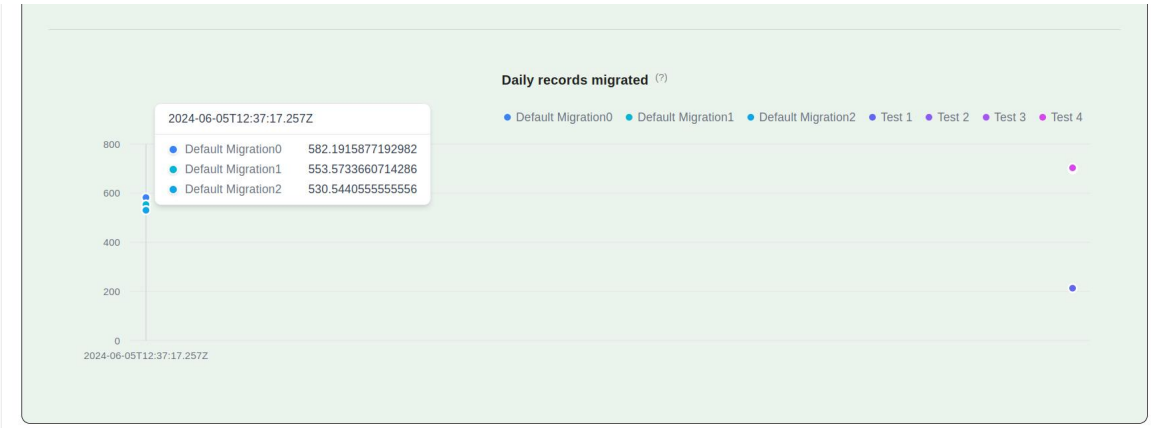


Figura 61 – Detalhes do terceiro painel de métricas

4.3.2 Migrações

A pagina de migrações contém uma listagem de todas as migrações criadas, que podem ser oriundas da aplicação Web ou do código. Além disso, é possível obter informações dos nomes das migrações, o seu tipo, o número total de réplicas em todas as instâncias dos *jobs*, fazer o gerenciamento por meios dos botões de *play*, *pause* e *stop* e, por fim, criar novas migrações.

Cada botão possui um comportamento único referente ao estado da migração. O botão *play* está disponível caso a migração não possua réplicas, não possua o mínimo de réplicas necessárias para uma migração completa, que seria pelo menos uma réplica para cada *job*, ou esteja no estado pausado. O botão *pause* impede que o *job* consuma itens da fila. Por fim, o botão de *stop* elimina todas as réplicas dos *jobs*.

A Figura 62 mostra a tela de migrações contendo todos os recursos criados desde o inicio da aplicação. Na listagem estão presente 6 migrações, todas sem réplicas no momento. Quando arrastamos o mouse para o botão de *play*, ele muda de cor para o verde, sinalizando que os pré-requisitos para sua execução estão cumpridos, conforme demonstra a Figura 63. Neste contexto, o botão de *play* cria uma réplica para cada *job* necessário e começa a migração.

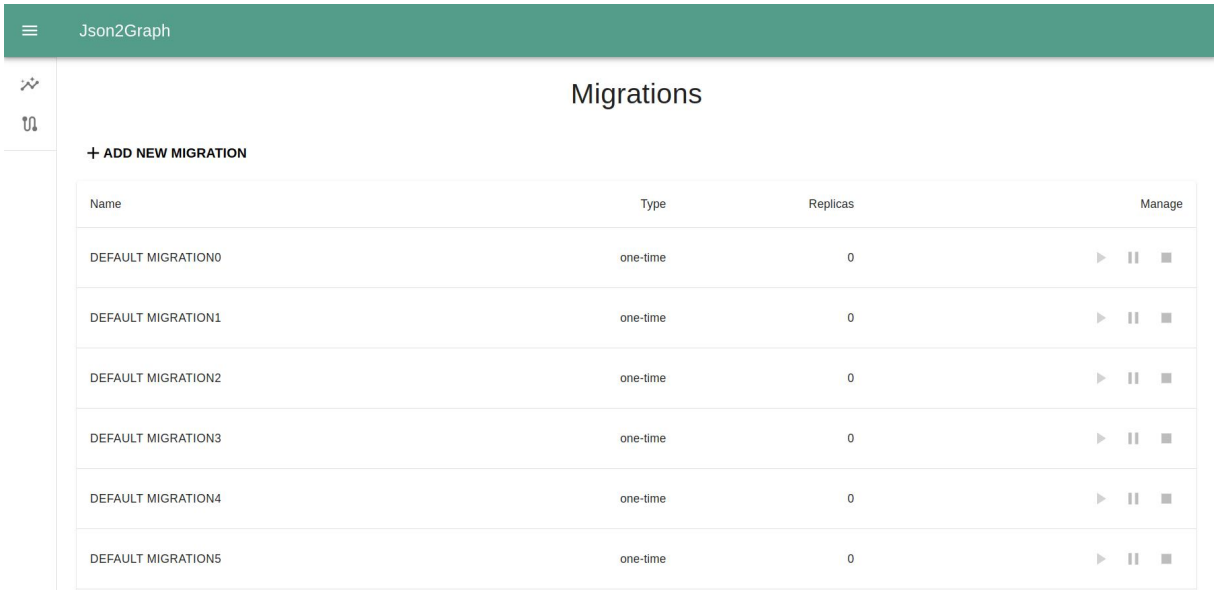


Figura 62 – Página de migrações

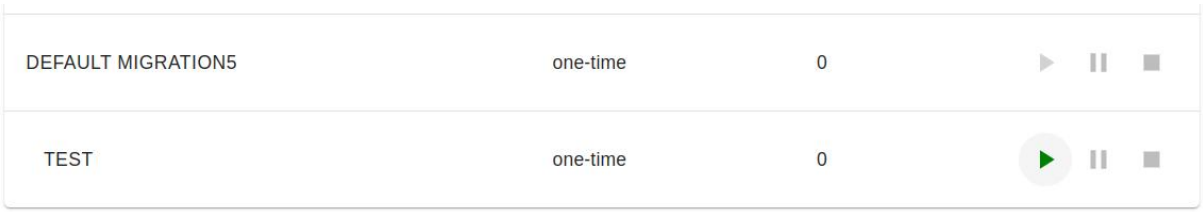


Figura 63 – Migração com o botão de *play* habilitado

Para criar uma nova migração, deve-se apertar o botão *Add new migration*. Ele direciona o usuário para uma tela de criação de migrações, que espera receber todas as informações necessárias. A Figura 64 demonstra a tela que o usuário se depara ao apertar no botão. Nela, é possível notar que existem três seções (*database*, *migration* e *advanced settings*), sendo que o usuário está presente na primeira seção (*database*).

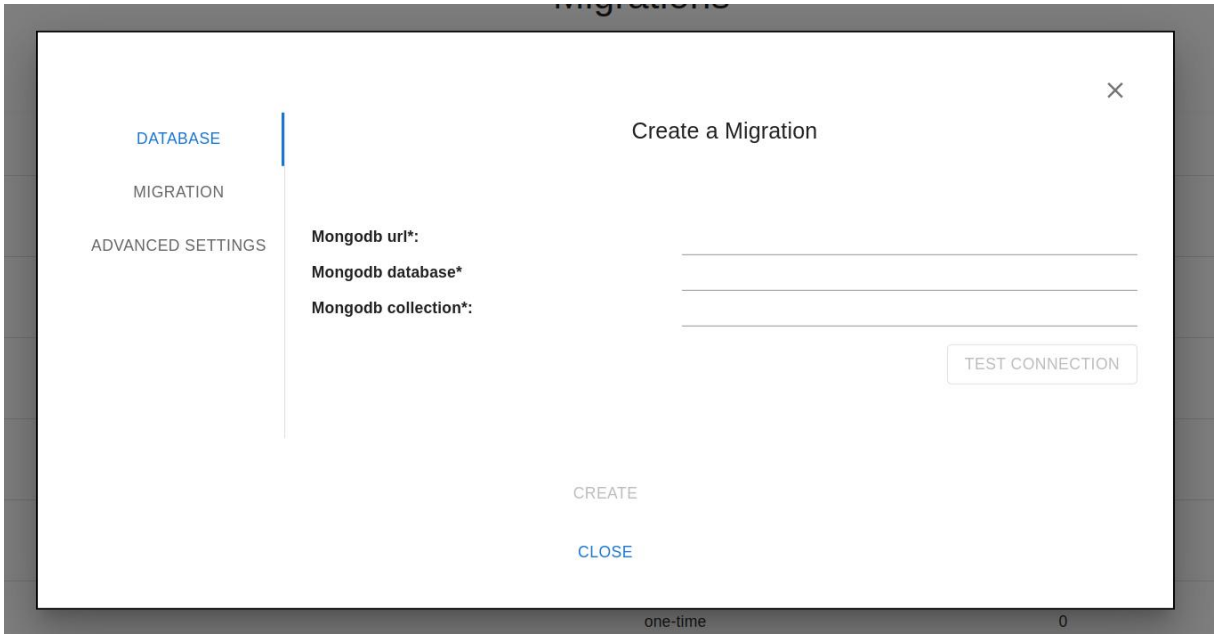


Figura 64 – Tela para criação de uma nova migração

A seção *database* possui três campos obrigatórios para serem preenchidos, além de um botão para teste de conexão. O primeiro campo espera receber uma url com todas as informações de conexão para o *Mongo DB*. Já o campo *Mongodb database* espera receber o *database* do banco de dados onde está presente a coleção a ser migrada. Por fim, o campo *Mongodb collection* espera receber a coleção que possui os objetos *json* a serem migrados.

Uma vez informada a *url* do banco de dados é possível utilizar o botão de teste de conexão para listar todas as *databases* e coleções disponíveis. A Figura 65 demonstra o comportamento do botão de teste de conexão, que modifica o campo de *Mongodb database* disponibilizando uma lista de todas as *databases* disponíveis para aquela conexão. Além disso, uma nova seção chamada de *logs* aparece contendo o resultado do teste de conexão.

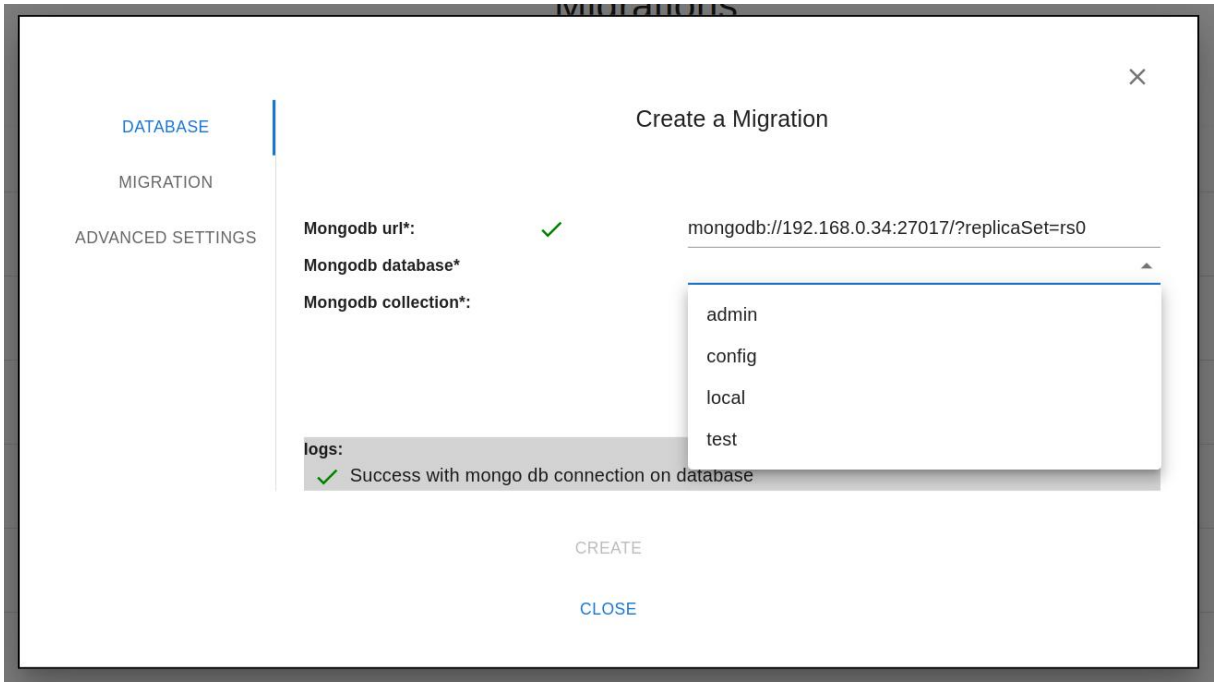


Figura 65 – Tela para criação de uma nova migração com url preenchida

Após a seleção da *database*, pode-se ver na Figura 66 que o campo *Mongodb collection* também é modificado para mostrar todas as coleções disponíveis naquele *database*.

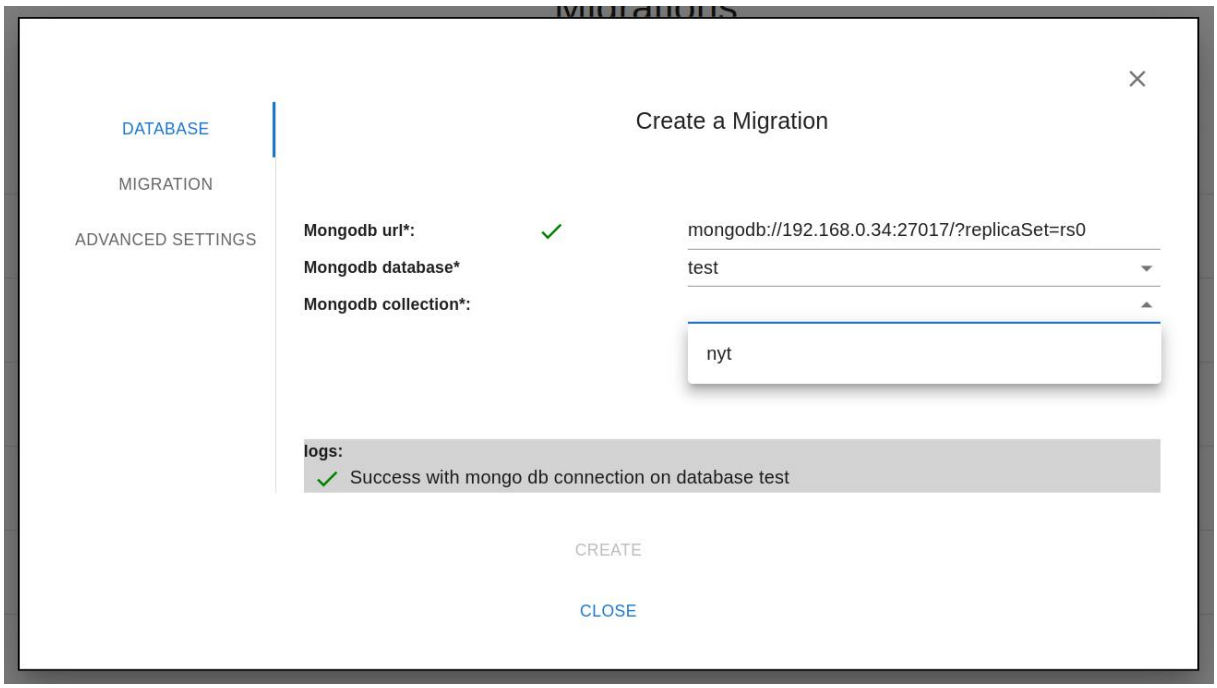


Figura 66 – Tela para criação de uma nova migração com *database* preenchida

Após configurada a conexão com o banco de dados é necessária apenas mais uma informação obrigatória antes do botão de criação ser disponibilizado. Essa informação é o

nome da migração, disponibilizada na seção *migration* e demonstrada na Figura 67.

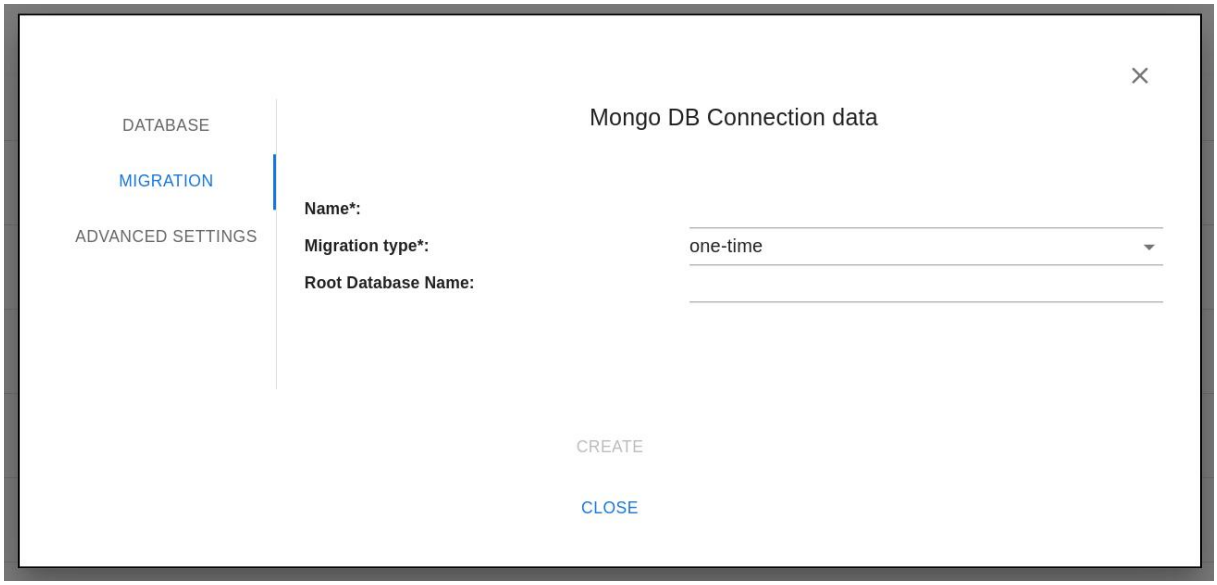


Figura 67 – Seção migration

Após preencher todos os campos obrigatórios pode-se criar uma nova migração. Na Figura 68 foi inserido o nome *Test* para satisfazer o requisito de nome, além do valor padrão de tipo de migração.

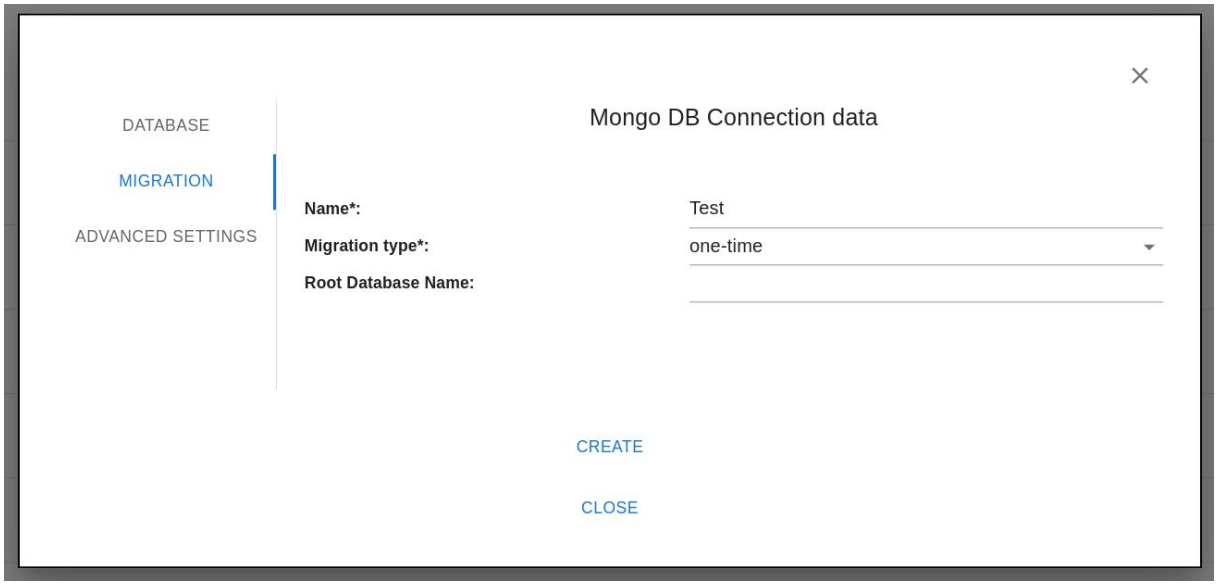


Figura 68 – Botão de criação habilitado

Caso necessário, pode-se utilizar a seção de configurações avançadas para modificar o número de réplicas iniciais de cada instância. Conforme demonstrado na Figura 69, pode-se informar o número de réplicas para os componentes *MongoDBProducer*, *JsonToNode*, *NodeToNeo4j* e *Neo4jCreateRelationship*. O padrão são 0 réplicas iniciais.

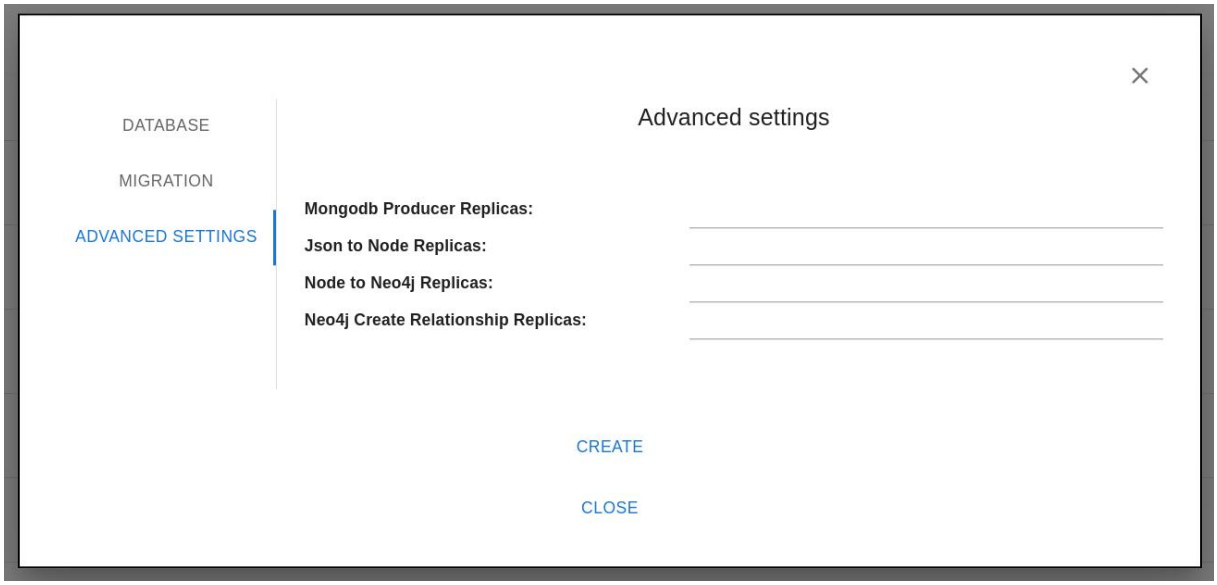


Figura 69 – Configurações avançadas

5 AVALIAÇÃO

Os critérios para avaliar a ferramenta se dividem em dois, sendo eles: a validação dos dados migrados e a avaliação da migração de dados. A validação dos dados migrados busca se certificar que os dados do banco de dados de origem chegaram no banco de dados de destino sem nenhuma perda por meio do teste contínuo com objetos JSON aleatórios. Já a avaliação da migração de dados busca avaliar uma migração por meio de dados obtidos para teste, sendo considerado a quantidade de dados migrados e relacionamentos criados comparados com o esperado e métricas de desempenho com diferentes variantes de ambiente e configurações.

5.1 VALIDAÇÃO DOS DADOS MIGRADOS

Para validar os dados migrados foi criado um *job* que cria modelos randômicos com outros modelos aninhados que injeta eventos próprios durante a criação para conseguir validar os dados assim que são escritos no *Neo4j*. A validação ocorre comparando os dados simples escritos por meio de um evento que estabelece uma ligação entre o *identificador* interno dado ao objeto e um *UUID* criado durante a geração dos dados. Assim, é possível obter o dado por meio do seu *id* e comparar se todos os campos estão no *Neo4j*.

Para validar as relações, o validador adiciona um campo de metadados nos objetos criados indicando que um campo contém objetos aninhados. Assim, após a escrita da aresta, é disparado um evento que também liga o *UUID* do objeto ao seu *id* e o validador verifica se foi estabelecido o mesmo número de relações que o número de objetos aninhados, podendo ser um objeto simples ou uma lista de objetos. Ao final, é escrito um relatório contendo o *JSON Schema* do modelo randômico criado, o seu estado original, o dado escrito no *Neo4j*, encontrado por meio de seu *id*, e qual é o resultado da validação.

As validações foram executadas por diversas vezes e não houve nenhum erro relacionado a dados faltando ou arestas não criadas. A Figura 70 mostra o resultado da validação de um modelo aleatório, sendo seu esquema e exemplo de dados disponibilizados nos Apêndices. Já a Figura 71 mostra os resultados de validação condensados por status, sendo a primeira linha correspondente a todas as validações, a linha 3 referente aos nós criados e a validação de propriedades presentes no banco de destino e a linha 6 referente às arestas criadas.

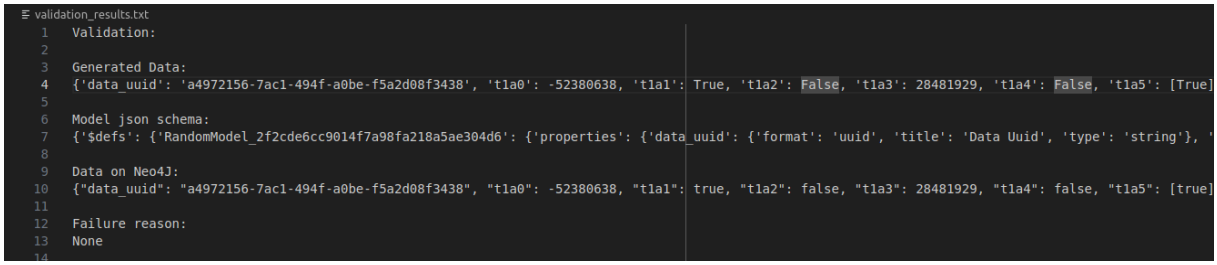


Figura 70 – Arquivo de log de resultados

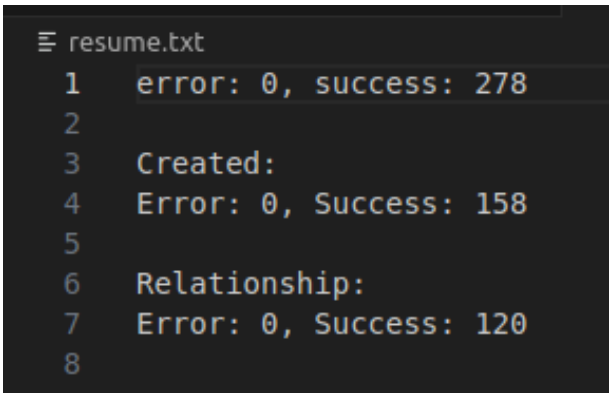


Figura 71 – Arquivo de resultado de validação

5.2 AVALIAÇÃO DA MIGRAÇÃO DE DADOS

Para avaliar a ferramenta, foram obtidos dados JSON aninhados de uma *API* pública. Esses dados foram escritos no *MongoDB* em instância criada por meio do *docker-compose*, sendo também criada uma instância do Neo4j. Após isso, foram definidas migrações com configurações variadas de réplicas para comparação dos resultados.

5.2.1 Coleta de dados

Foi efetuada a captura de questões criadas no site *StackOverflow* por meio da *API* do site *StackExchange* com uma chave de aplicação criada exclusivamente para esse trabalho. Após a captura, os dados foram salvos no *MongoDB*, dentro de uma *database* específica, chamada de *stackoverflow*, e uma coleção chamada de *questions*.

Os dados do *StackOverflow* possuem 3 objetos, sendo eles a raiz, chamada de *questions*, que é composta por dados de questões da entidade *Questions*, os dados de resposta, que são chamados de *Answers*, e os dados de quem fez a resposta, chamado de *Owners*. A Figura 72 mostra o esquema do objeto *Questions* obtido da *API* e seus objetos aninhados *Answers* e *Owners*.

Ao todo, foram obtidas 100000 questões com 58983 respostas dadas por 58983 usuários. Como o campo *answers* dentro do objeto *Question* é um *array* de objetos, existem algumas questões que possuem mais de uma resposta, enquanto outras não possuem nenhuma resposta.

```
type Owner = {
    user_id: number;
    display_name: string;
}

type Answer = {
    owner: Owner;
    answer_id: number;
}

type Question = {
    _id: string;
    tags: string[];
    answers: Answer[];
    question_id: number;
    share_link: string;
    title: string;
}
```

Figura 72 – Esquema do objeto Questions

Fonte: Elaboração própria

5.2.2 Configuração do ambiente

Para a migração dos dados, foi utilizado um notebook Intel i7 1260P 4.70 GHz e 16 GB de memória RAM 5200 MHz. Toda a infraestrutura foi criada na mesma máquina, por meio do *docker-compose*, e disponibilizada na rede local.

Para o teste distribuído, foi criado um cluster *Ray* na rede local contendo mais um notebook 10a geração do Intel Core i5-10200H 4,1 GHz e 32GB de memória RAM 2666MHZ. Esse notebook foi preparado para receber a migração, sendo feita a instalação das bibliotecas necessárias conforme especifica o gerenciador de dependências utilizado para este trabalho, chamado de *Poetry*[1].

5.2.3 Métricas de avaliação

Espera-se que a ferramenta consiga realizar a escrita de todos os objetos, sendo eles mapeados para nós no banco de dados Neo4j, e todas as informações de aninhamento,

[1] https://python-poetry.org/

sendo elas mapeadas para arestas entre os nós. Para identificar todos os nós e arestas que devem ser escritas, foram utilizadas as seguintes consultas para análise de objetos JSON no banco de dados MongoDB:

- Consulta para contagem de objetos *answers*:

```
[
    {
    "$match": {"answers": {"$exists": True, "$not": {"$size": 0}}}
    },
    {
    "$group": {
            "_id": None,
            "totalAnswers": {"$sum": {"$size": "$answers"}}
    }
    }
]
```

- Consulta para contagem de objetos *owner*:

```
[
    {
        "$unwind": "$answers"
    },
    {
        "$group": {
            "_id": None,
            "count": {"$sum": 1}
        }
    }
]
```

Esta consulta obteve como resultado 217966 objetos totais, sendo 100000 objetos *questions*, 58983 objetos *answers* e o mesmo número de objetos *owner*. Dessa forma, é esperado que sejam criados no Neo4j 100000 nós, sendo que a eles deve ser atribuído a etiqueta de *questions*, além de mais 58983 nós correspondentes às respostas, com etiqueta de *answers*, e também 58983 nós correspondentes aos usuários donos das respostas, com a etiqueta *owner*. Para validar a quantidade de dados JSON convertidos e migrados, foram efetuadas as seguintes consultas no *Neo4j*:

- Quantidade de nós *questions*:

```
match (n: questions) return count(n);
```

- Quantidade de nós *answers*

```
match (n: answers) return count(n);
```

- Quantidade de nós *owner*

```
match (n: owner) return count(n);
```

- Quantidade de relações

```
MATCH ()-[relationship]->()
RETURN TYPE(relationship) AS type, COUNT(relationship) AS amount
ORDER BY amount DESC;
```

Para validar a migração, além das consultas executadas no MongoDB para contagem dos objetos, foi efetuada uma migração utilizando uma ferramenta, disponibilizada pelo próprio Neo4j para a importação de documentos JSON, por meio de um *plugin* de conexão com o banco de dados. Os resultados dessa migração foram analisados por meio das consultas anteriores para garantir que os números de nós e arestas esperados poderiam ser alcançados. Essa consulta foi adaptada segundo o site do Neo4j e executou no tempo de um pouco mais de **1 hora**. A consulta utilizada para a migração dos dados foi a seguinte:

```
CALL apoc.mongo.find(
 "mongodb://192.168.0.34:27017/stackoverflow.questions?replicaSet=rs0"
) YIELD value as q
MERGE (question:questions {
   _id: q.question_id,
   title: coalesce(q.title, "no_title"),
   share_link: q.share_link,
   favorite_count: coalesce(q.favorite_count, 0),
   tags: q.tags
})

FOREACH (a IN q.answers |
   MERGE (question)<-[:ANSWERS]-(answer:answers {_id:a.answer_id})
   MERGE (answerer:owner {
    _id:coalesce(a.owner.user_id, "DELETED_" + a.owner.display_name),
    display_name: a.owner.display_name
   })
   MERGE (answer)<-[:PROVIDED]-(answerer)
)
```

5.2.4 Resultados

A Tabela 2 mostra as configurações de réplicas por componente, sendo criadas três configurações que executaram a migração de todos os dados e posteriormente foram analisadas por meio das consultas para contagem de nós e arestas. A *Configuração 1* foi executada localmente e com base no consumo das filas foi criada a *Configuração 2* e *Configuração 3*, que executaram no cluster local *Ray*. O número de réplicas por configuração foram variadas segundo as métricas das filas, assim as filas com maior acumulação de dados para serem consumidos indicaram quais componentes deveriam ter o seu número de réplicas aumentado, assim podendo consumir mais dados da fila. As réplicas de **Consumer** e **Executor** gerenciam o processamento de eventos e execução dos *handlers*, respectivamente.

Tabela 2 – Configurações de réplicas por componente

Réplicas	Configuração 1	Configuração 2	Configuração 3
MongoDBProducer	1	1	1
JsonToNode	1	3	4
NodeToNeo4j	2	3	5
Neo4jCreateRelationship	3	5	6
Consumer	1	2	3
Executor	3	4	6

A Tabela 3 condensa o que era esperado da migração e os resultados obtidos ao final da execução, analisados por meio das consultas de contagem de nós e arestas no Neo4j. É possível notar que todos os dados esperados foram escritos e as arestas estabelecidas.

Tabela 3 – Comparação dos dados esperados e escritos por configuração

Resultado	Esperado	Configuração 1	Configuração 2	Configuração 2
Nós totais	217966	217966	217966	217966
Nós *questions*	100000	100000	100000	100000
Nós *answers*	58983	58983	58983	58983
Nós *owners*	58983	58983	58983	58983
Arestas *questions* -> *answers*	58983	58983	58983	58983
Arestas *answers* -> *owners*	58983	58983	58983	58983

A Tabela 4 mostra o desempenho por configuração, sendo ele composto pelo tempo médio para criação de nós e arestas e o tempo total da migração. As métricas de memória e CPU foram coletadas por meio das métricas do *Ray*, sendo os gráficos presentes no *Prometheus*. As configurações que rodam em cluster possuem duas métricas, sendo a primeira referente ao primeiro notebook, de 16GB, e a segunda para o notebook de 32GB.

A *Configuração 1* executou no tempo total de 39 minutos e 7 segundos, sendo que, conforme mostra a Figura 73, o uso de CPU teve alguns picos, mas se manteve próximo da

Tabela 4 – Desempenho

Desempenho	Configuração 1	Configuração 2	Configuração 3
Tempo de execução total	39m:7s	30m:21s	27m:57s
Tempo médio de escrita de nós	92,87/segundo	119,70/segundo	129,97/segundo
Uso de memória médio	6,011GB	4,498GB + 2,492GB	5,087GB + 2,560GB
Uso de CPU médio	19,55%	22,11% + 11%	24,64% + 11,86%

média de 19%, enquanto que o uso de memória, conforme mostra a Figura 74 aumentou até o pico de um pouco mais de 6GB.

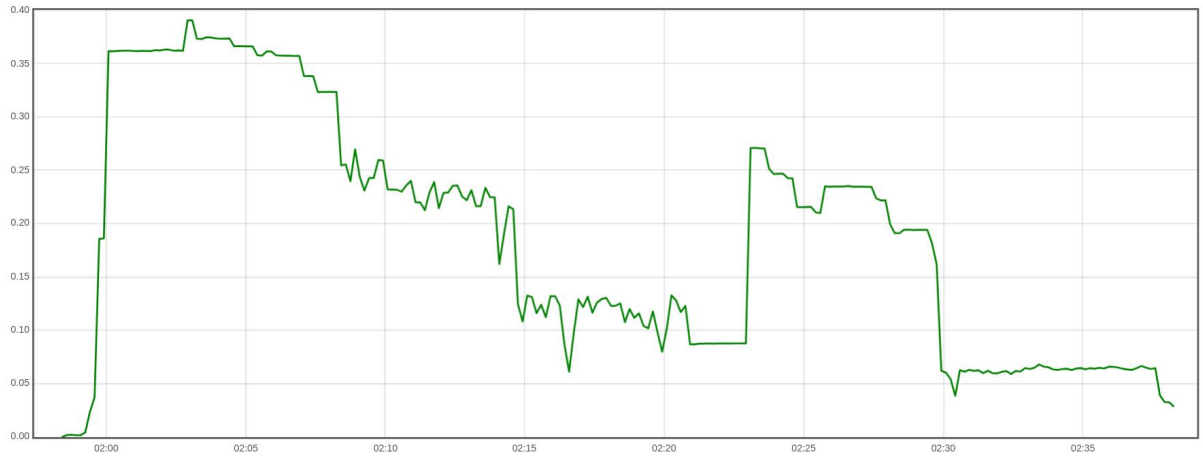


Figura 73 – Uso de CPU durante a migração para configuração 1

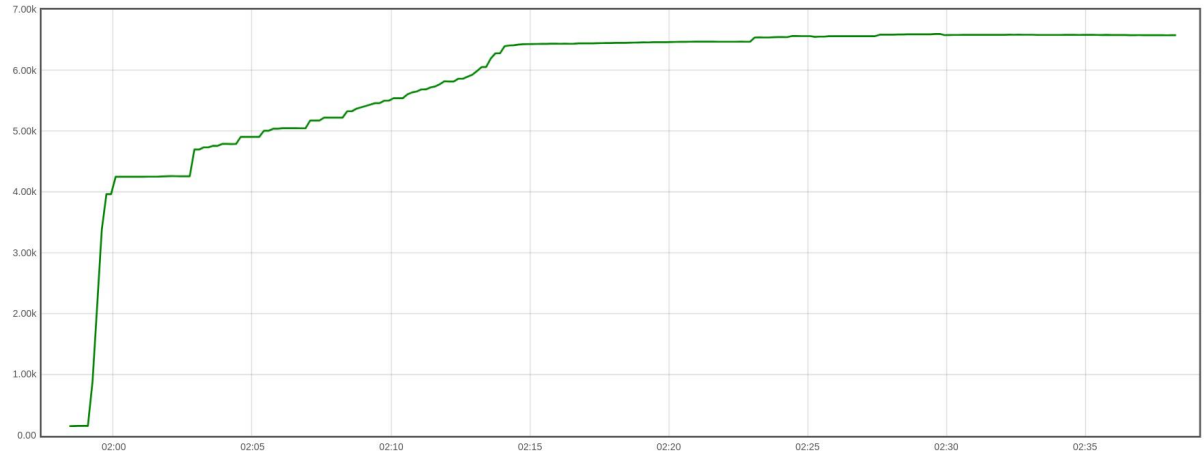


Figura 74 – Uso de memória durante a migração (em Megabytes) para configuração 1

Já a *Configuração 2* conseguiu rodar em 30 minutos e 21 segundos, sendo que sua execução se deu no *cluster* criado na rede local contendo os dois notebooks. O uso de memória e CPU se dividiu entre as duas máquinas, conforme mostram as Figuras 75 e 76, com um pouco mais de uso na média.

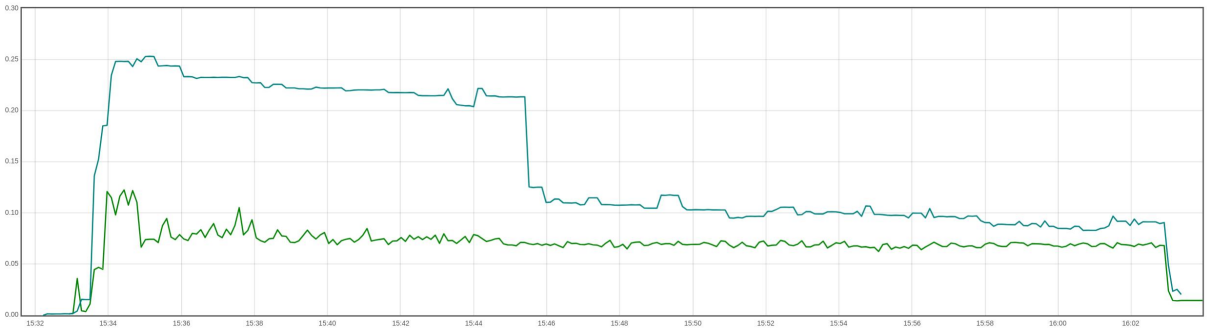


Figura 75 – Uso de CPU durante a migração para configuração 2

Fonte: Elaboração própria

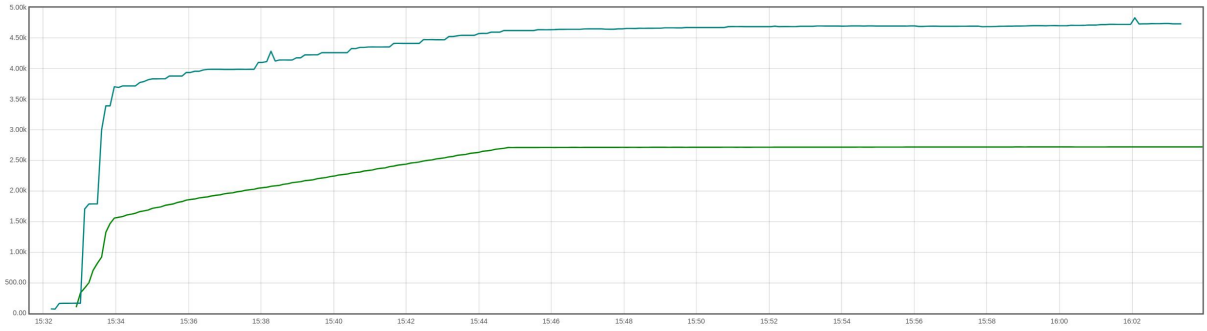


Figura 76 – Uso de memória durante a migração (em Megabytes) para configuração 2

Fonte: Elaboração própria

Por fim, a *Configuração 3* obteve o tempo de 27 minutos e 57 segundos, sendo ela também executada no *cluster*. O uso de memória e CPU ficaram próximos da *configuração 2*, conforme mostram as Figuras 77 e 78, mas houve um aumento da média.

Figura 77 – Uso de CPU durante a migração para configuração 3

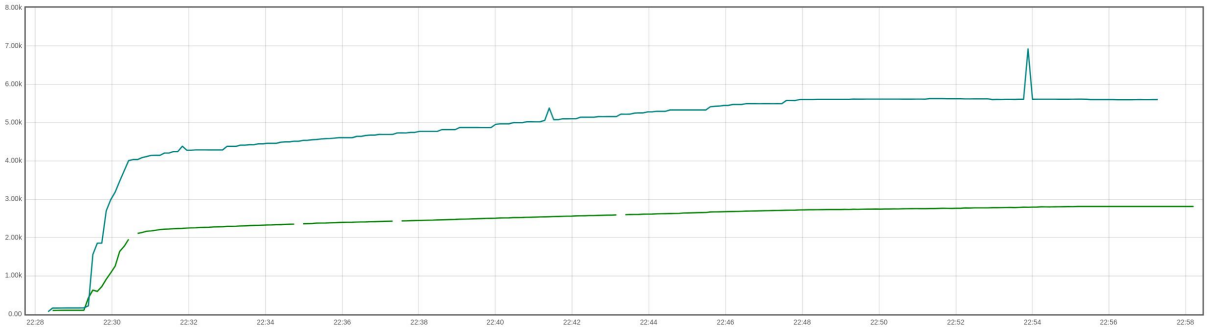


Figura 78 – Uso de memória durante a migração (em Megabytes) para configuração 3

A Figura 79 condensa o tempo resultante das migrações para as três configurações, comparando os resultados destas configurações com o tempo gasto pelo *plugin* fornecido pelo *Neo4j*. É possível verificar o melhor desempenho da nossa solução em relação à ferramenta do *Neo4j*.

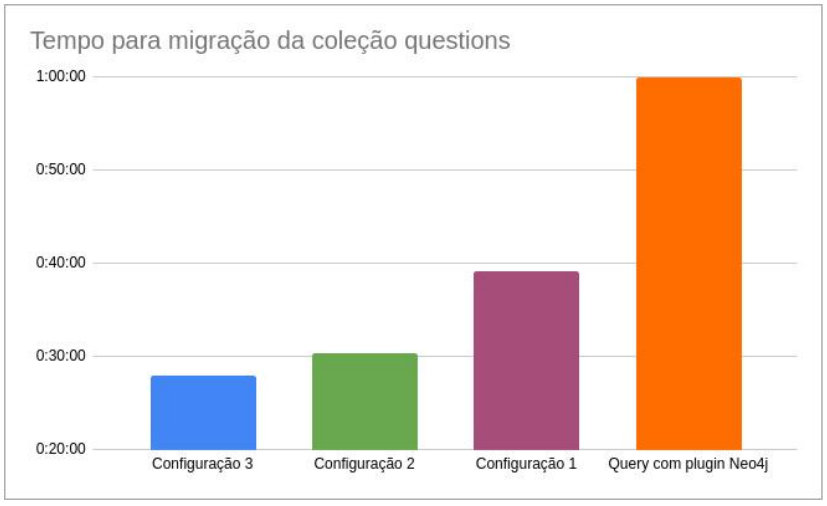


Figura 79 – Gráfico de resultados para tempo de migração

Vale observar ainda que a ferramenta proposta neste trabalho não necessita de nenhum dado adicional, diferente dos métodos de acesso providos pelo *plugin* do *Neo4j*, que necessitam definir manualmente as arestas e as propriedades dos objetos.

6 CONCLUSÃO

Este trabalho de conclusão de curso teve como objetivo o desenvolvimento de uma ferramenta que permite a migração de dados semiestruturados, sendo o foco nos tipos *JSON*, armazenados em um banco de dados, sendo o banco de dados escolhido *MongoDB*, para um banco de dados grafos, no caso o banco de dados *Neo4j*.

Esta ferramenta se diferencia dos trabalhos relacionados por possibilitar a migração de dados sem a necessidade de especificação de esquema, já que a migração ocorre de um banco de dados NoSQL para outro, e sua execução acontece de forma concorrente, distribuída e em *streaming*, com uma estratégia assíncrona de recomposição de informações.

A motivação para o desenvolvimento desse trabalho foi facilitar a migração de dados entre bancos de dados com diferentes modelos de dados de forma fácil e distribuída, sem a necessidade de esforço para modelar dados NoSQL, que podem ter esquema diversificado, mantendo as informações aninhadas, que se traduzem como relacionamentos no banco de dados destino.

As principais contribuições desse trabalho são:

- Desenvolvimento de uma *pipeline* de processamento de dados *JSON* aninhados;
- Regras de mapeamento entre o modelo de dados JSON e o modelo de dados de grafos de propriedades;
- Arquitetura distribuída em *streaming* utilizando a linguagem de programação *Python*;
- Execução de código assíncrono com base em sentenças lógicas provenientes da estrutura de *eventos*;
- Interface *web* que possui painéis e ferramentas para criação e gerenciamento de migrações;
- Ferramenta *Json2Node*, que realiza a migração de dados em *streaming* com execução concorrente e distribuída sem a necessidade de nenhuma informação sobre os dados que serão migrados.

Conforme demonstrado no Capítulo 5, foi possível realizar a migração de dados *JSON* aninhados presentes no banco de dados *MongoDB* para o banco de de dados *Neo4j*, mantendo todos os campos originais e estabelecendo todas as relações com os objetos internos. Dessa forma, o objetivo geral do trabalho pode ser considerado atingido.

Como atividades futuras relacionadas a esse trabalho, podemos considerar:

- Implementação de leitura concorrente no *MongoDB* pelo módulo *MongoDBProducer*;

- Disponibilização de parâmetros para configurar a estratégia de migração do módulo *JsonToNode*, possibilitando o processamento de dados completos caso o tamanho do objeto não seja grande o suficiente para o *overhead* de envio para a fila de retroalimentação;
- Correção dos problemas de desempenho do módulo de processamento de eventos;
- Criação de mais painéis voltados a uma análise mais aprofundada das métricas;
- Mais métricas de acompanhamento;
- Análise de desempenho de cada módulo buscando aperfeiçoar sua execução;
- Melhorias no algoritmo de detecção de tabelas e esquemas, buscando se tornar resiliente aos dados *JSON* com campos nulos escondidos;
- Possibilidade de criar relações com nomes diferentes dos padrões, podendo o usuário definir ou então a escolha por meio de um modelo de linguagem.

Por fim, o código da aplicação está disponível em um repositório público do site GitHub por meio do link https://github.com/gorgonun/json-to-graph.

REFERÊNCIAS

ABDELHEDI, Fatma *et al.* UMLtoNoSQL: Automatic Transformation of Conceptual Schema to NoSQL Databases. *In:* 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA). [*S.l.*: *s.n.*], 2017. P. 272–279. DOI: <10.1109/AICCSA.2017.76>.

ALOTAIBI, Obaid; PARDEDE, Eric. Transformation of Schema from Relational Database (RDB) to NoSQL Databases. **Data**, v. 4, n. 4, 2019. ISSN 2306-5729. DOI: <10.3390/data4040148>. Disponível em: <https://www.mdpi.com/2306-5729/4/4/148>.

CODD, E. F. A Relational Model of Data for Large Shared Data Banks. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 13, n. 6, p. 377–387, jun. 1970. ISSN 0001-0782. DOI: <10.1145/362384.362685>. Disponível em: <https://doi.org/10.1145/362384.362685>.

DE VIRGILIO, Roberto; MACCIONI, Antonio; TORLONE, Riccardo. Converting Relational to Graph Databases. *In:* FIRST International Workshop on Graph Data Management Experiences and Systems. New York, New York: Association for Computing Machinery, 2013. (GRADES '13). DOI: <10.1145/2484425.2484426>. Disponível em: <https://doi.org/10.1145/2484425.2484426>.

HEWITT, Carl. Actor Model of Computation: Scalable Robust Information Systems. arXiv, ago. 2010.

HOLUBOVA, Irena; CONTOS, Pavel; SVOBODA, Martin. Multi-Model Data Modeling and Representation: State of the Art and Research Challenges. *In:* PROCEEDINGS of the 25th International Database Engineering & Applications Symposium. Montreal, QC, Canada: Association for Computing Machinery, 2021. (IDEAS '21), p. 242–251. DOI: <10.1145/3472163.3472267>. Disponível em: <https://doi.org/10.1145/3472163.3472267>.

MEIER, Andreas; KAUFMANN, Michael. **SQL & NoSQL databases**. [*S.l.*]: Springer, 2019. ISBN 978-3-658-24549-8.

MICHAEL NASH, Wade Waldron. **Applied Akka Patterns**. [*S.l.*: *s.n.*]. Disponível em: <https://www.oreilly.com/library/view/applied-akka-patterns/9781491934876/ch01.html>.

MORITZ, Philipp *et al.* Ray: a distributed framework for emerging AI applications. *In:* PROCEEDINGS of the 13th USENIX Conference on Operating Systems Design and Implementation. Carlsbad, CA, USA: USENIX Association, 2018. (OSDI'18), p. 561–577.

MUS, MAM. Comparison between SQL and NoSQL databases and their relationship with big data analytics, 2019.

NEO4J. **Cypher Query Language**. [*S.l.*: *s.n.*]. Disponível em: <https://neo4j.com/developer/cypher/>.

POKORNY, Jaroslav. NoSQL Databases: A Step to Database Scalability in Web Environment. *In:* PROCEEDINGS of the 13th International Conference on Information Integration and Web-Based Applications and Services. Ho Chi Minh City, Vietnam: Association for Computing Machinery, 2011. (iiWAS '11), p. 278–283. DOI: <10.1145/2095536.2095583>. Disponível em: <https://doi.org/10.1145/2095536.2095583>.

QUEIROZ, Jordan S. *et al.* AMANDA: A Middleware for Automatic Migration between Different Database Paradigms. **Applied Sciences**, v. 12, n. 12, 2022. ISSN 2076-3417. DOI: <10.3390/app12126106>. Disponível em: <https://www.mdpi.com/2076-3417/12/12/6106>.

SAHATQIJA, Kosovare *et al.* Comparison between relational and NOSQL databases. *In:* IEEE. 2018 41st international convention on information and communication technology, electronics and microelectronics (MIPRO). [*S.l.*: *s.n.*], 2018. P. 0216–0221.

SAMPAIO, RF; MANCINI, MC. Estudos de revisão sistemática: um guia para síntese criteriosa da evidência científica. **Brazilian Journal of Physical Therapy**, Associação Brasileira de Pesquisa e Pós-Graduação em Fisioterapia, v. 11, n. 1, p. 83–89, jan. 2007. ISSN 1413-3555. DOI: <10.1590/S1413-35552007000100013>. Disponível em: <https://doi.org/10.1590/S1413-35552007000100013>.

SILBERSCHATZ, Avi; KORTH, Henry F.; SUDARSHAN, S. **Database System Concepts, Seventh Edition**. [*S.l.*]: McGraw-Hill Book Company, 2020. ISBN 9780078022159. Disponível em: <https://www.db-book.com/>.

ÜNAL, Yelda; OĞUZTÜZÜN, Halit. Migration of data from relational database to graph database. *In:* p. 1–5. DOI: <10.1145/3200842.3200852>.

Apêndices

APÊNDICE A – EXEMPLO DE ESQUEMA JSON UTILIZADO PARA VALIDAÇÃO DOS DADOS


```
{
    "$defs": {
        "RandomModel_2f2cde6cc9014f7a98fa218a5ae304d6": {
            "properties": {
                "data_uuid": {
                    "format": "uuid",
                    "title": "Data Uuid",
                    "type": "string"
                },
                "t4a0": {
                    "default": [],
                    "items": {
                        "type": "string"
                    },
                    "title": "T4A0",
                    "type": "array"
                },
                "t4a1": {
                    "title": "T4A1",
                    "type": "integer"
                },
                "t4a2": {
                    "title": "T4A2",
                    "type": "number"
                },
                "t4a3": {
                    "title": "T4A3",
                    "type": "string"
                },
                "t4a4": {
                    "title": "T4A4",
                    "type": "boolean"
                },
                "t4a5": {
                    "title": "T4A5",
                    "type": "integer"
```

```
            },
            "t4a6": {
                "title": "T4A6",
                "type": "string"
            },
            "t4a7": {
                "title": "T4A7",
                "type": "string"
            }
        },
        "required": [
            "data_uuid",
            "t4a1",
            "t4a2",
            "t4a3",
            "t4a4",
            "t4a5",
            "t4a6",
            "t4a7"
        ],
        "title": "RandomModel_2f2cde6cc9014f7a98fa218a5ae304d6",
        "type": "object"
    },
    "RandomModel_4c481c3a9d2447dca5b74cebb3d21100": {
        "properties": {
            "data_uuid": {
                "format": "uuid",
                "title": "Data Uuid",
                "type": "string"
            },
            "t3a0": {
                "title": "T3A0",
                "type": "boolean"
            },
            "t3a1": {
                "default": [],
                "items": {
                    "type": "boolean"
                },
```

```
        "title": "T3A1",
        "type": "array"
    },
    "t3a2": {
        "title": "T3A2",
        "type": "number"
    },
    "t3a3": {
        "title": "T3A3",
        "type": "boolean"
    },
    "t3a4": {
        "anyOf": [
            {
                "type": "number"
            },
            {
                "type": "null"
            }
        ],
        "default": None,
        "title": "T3A4"
    },
    "t3a5": {
        "title": "T3A5",
        "type": "string"
    },
    "t3a6": {
        "anyOf": [
            {
                "type": "integer"
            },
            {
                "type": "null"
            }
        ],
        "default": None,
        "title": "T3A6"
    },
```

```
            "t3a7": {
                "title": "T3A7",
                "type": "boolean"
            }
        },
        "required": [
            "data_uuid",
            "t3a0",
            "t3a2",
            "t3a3",
            "t3a5",
            "t3a7"
        ],
        "title": "RandomModel_4c481c3a9d2447dca5b74cebb3d21100",
        "type": "object"
    },
    "RandomModel_6d9a8f1a6b484d3c97a83d65a51694d2": {
        "properties": {
            "data_uuid": {
                "format": "uuid",
                "title": "Data Uuid",
                "type": "string"
            },
            "t6a0": {
                "title": "T6A0",
                "type": "integer"
            },
            "t6a1": {
                "anyOf": [
                    {
                        "type": "boolean"
                    },
                    {
                        "type": "null"
                    }
                ],
                "default": None,
                "title": "T6A1"
            },
```

```
"t6a2": {
    "anyOf": [
        {
            "items": {
                "type": "integer"
            },
            "type": "array"
        },
        {
            "type": "null"
        }
    ],
    "default": None,
    "title": "T6A2"
},
"t6a3": {
    "default": [],
    "items": {
        "type": "number"
    },
    "title": "T6A3",
    "type": "array"
},
"t6a4": {
    "title": "T6A4",
    "type": "boolean"
},
"t6a5": {
    "default": [],
    "items": {
        "type": "boolean"
    },
    "title": "T6A5",
    "type": "array"
},
"t6a6": {
    "title": "T6A6",
    "type": "number"
},
```

```
            "t6a7": {
                "default": [],
                "items": {
                    "type": "integer"
                },
                "title": "T6A7",
                "type": "array"
            },
            "t6a8": {
                "title": "T6A8",
                "type": "integer"
            },
            "t6a9": {
                "title": "T6A9",
                "type": "integer"
            }
        },
        "required": [
            "data_uuid",
            "t6a0",
            "t6a4",
            "t6a6",
            "t6a8",
            "t6a9"
        ],
        "title": "RandomModel_6d9a8f1a6b484d3c97a83d65a51694d2",
        "type": "object"
    },
    "RandomModel_8af5a6c4f36e4d3792052e5ade486b69": {
        "properties": {
            "data_uuid": {
                "format": "uuid",
                "title": "Data Uuid",
                "type": "string"
            },
            "t5a0": {
                "anyOf": [
                    {
                        "type": "string"
```

```
                },
                {
                    "type": "null"
                }
            ],
            "default": None,
            "title": "T5A0"
        },
        "t5a1": {
            "title": "T5A1",
            "type": "integer"
        },
        "t5a2": {
            "title": "T5A2",
            "type": "integer"
        },
        "t5a3": {
            "title": "T5A3",
            "type": "boolean"
        },
        "t5a4": {
            "title": "T5A4",
            "type": "number"
        },
        "t5a5": {
            "title": "T5A5",
            "type": "integer"
        },
        "t5a6": {
            "title": "T5A6",
            "type": "string"
        }
    },
    "required": [
        "data_uuid",
        "t5a1",
        "t5a2",
        "t5a3",
        "t5a4",
```

```
            "t5a5",
            "t5a6"
        ],
        "title": "RandomModel_8af5a6c4f36e4d3792052e5ade486b69",
        "type": "object"
    },
    "RandomModel_929eab692ccf4be2b9a074c174c49876": {
        "properties": {
            "data_uuid": {
                "format": "uuid",
                "title": "Data Uuid",
                "type": "string"
            },
            "t7a0": {
                "title": "T7A0",
                "type": "number"
            },
            "t7a1": {
                "anyOf": [
                    {
                        "type": "boolean"
                    },
                    {
                        "type": "null"
                    }
                ],
                "default": None,
                "title": "T7A1"
            }
        },
        "required": [
            "data_uuid",
            "t7a0"
        ],
        "title": "RandomModel_929eab692ccf4be2b9a074c174c49876",
        "type": "object"
    },
    "t2": {
        "properties": {
```

```
"data_uuid": {
    "format": "uuid",
    "title": "Data Uuid",
    "type": "string"
},
"t2a0": {
    "title": "T2A0",
    "type": "string"
},
"t2a1": {
    "title": "T2A1",
    "type": "integer"
},
"t2a2": {
    "title": "T2A2",
    "type": "boolean"
},
"t2a3": {
    "title": "T2A3",
    "type": "integer"
},
"t2a4": {
    "title": "T2A4",
    "type": "integer"
},
"t2a5": {
    "title": "T2A5",
    "type": "integer"
},
"random_model_4c481c3a9d2447dca5b74cebb3d21100": {
    "$ref": "#/$defs/RandomModel_
    4c481c3a9d2447dca5b74cebb3d21100"
},
"random_model_2f2cde6cc9014f7a98fa218a5ae304d6": {
    "$ref": "#/$defs/RandomModel_
    2f2cde6cc9014f7a98fa218a5ae304d6"
},
"random_model_8af5a6c4f36e4d3792052e5ade486b69": {
    "$ref": "#/$defs/RandomModel_
```

```
                    8af5a6c4f36e4d3792052e5ade486b69"
                },
                "random_model_6d9a8f1a6b484d3c97a83d65a51694d2": {
                    "items": {
                        "$ref": "#/$defs/RandomModel_
                        6d9a8f1a6b484d3c97a83d65a51694d2"
                    },
                    "title": "Random Model 6D9A8F1A6B484D3C97A83D65A51694D2",
                    "type": "array"
                }
            },
            "required": [
                "data_uuid",
                "t2a0",
                "t2a1",
                "t2a2",
                "t2a3",
                "t2a4",
                "t2a5",
                "random_model_4c481c3a9d2447dca5b74cebb3d21100",
                "random_model_2f2cde6cc9014f7a98fa218a5ae304d6",
                "random_model_8af5a6c4f36e4d3792052e5ade486b69",
                "random_model_6d9a8f1a6b484d3c97a83d65a51694d2"
            ],
            "title": "t2",
            "type": "object"
        }
    },
    "properties": {
        "data_uuid": {
            "format": "uuid",
            "title": "Data Uuid",
            "type": "string"
        },
        "t1a0": {
            "title": "T1A0",
            "type": "integer"
        },
        "t1a1": {
```

```
            "title": "T1A1",
            "type": "boolean"
        },
        "t1a2": {
            "title": "T1A2",
            "type": "boolean"
        },
        "t1a3": {
            "title": "T1A3",
            "type": "integer"
        },
        "t1a4": {
            "title": "T1A4",
            "type": "boolean"
        },
        "t1a5": {
            "anyOf": [
                {
                    "items": {
                        "type": "boolean"
                    },
                    "type": "array"
                },
                {
                    "type": "null"
                }
            ],
            "default": None,
            "title": "T1A5"
        },
        "t1a6": {
            "anyOf": [
                {
                    "type": "integer"
                },
                {
                    "type": "null"
                }
            ],
```

```
            "default": None,
            "title": "T1A6"
        },
        "t2": {
            "$ref": "#/$defs/t2"
        },
        "random_model_929eab692ccf4be2b9a074c174c49876": {
            "$ref": "#/$defs/RandomModel_929eab692ccf4be2b9a074c174c49876"
        }
    },
    "required": [
        "data_uuid",
        "t1a0",
        "t1a1",
        "t1a2",
        "t1a3",
        "t1a4",
        "t2",
        "random_model_929eab692ccf4be2b9a074c174c49876"
    ],
    "title": "t1",
    "type": "object"
}
```

APÊNDICE B – EXEMPLO DE DADO ALEATÓRIO GERADO ATRAVÉS DO JSON ESQUEMA (TRUNCADO DEVIDO AO TAMANHO DE 7543 LINHAS)


```
{
    "data_uuid": "a4972156-7ac1-494f-a0be-f5a2d08f3438",
    "t1a0": -52380638,
    "t1a1": true,
    "t1a2": false,
    "t1a3": 28481929,
    "t1a4": false,
    "t1a5": [
        true
    ],
    "t1a6": 90879003,
    "t2": {
        "data_uuid": "fcc0f664-232b-41b5-b89e-759051a6aebc",
        "t2a0": "COP3dQLe",
        "t2a1": 83411830,
        "t2a2": false,
        "t2a3": 24157531,
        "t2a4": -30400224,
        "t2a5": 32125767,
        "random_model_4c481c3a9d2447dca5b74cebb3d21100": {
            "data_uuid": "1da3e3c8-0b9b-40ca-85d5-5a085e276121",
            "t3a0": false,
            "t3a1": [
                false,
                false,
                true,
                true,
                false,
                true,
                false,
                false,
                false,
                false,
                true,
                true,
                false,
```

```
                false,
                false,
                false,
                false,
                true,
                false,
                false,
                true,
                false,
                true,
                true,
                true,
                true,
                true,
                false,
                true,
                false,
                true,
                true,
                false,
                false,
                false,
                false,
                false,
                true,
                false,
                true,
                false,
                false,
                true,
                false,
                true,
                false,
                false,
                true,
                false,
                true,
                true,
                false,
```

```
                false,
                true,
                true,
                false,
                false,
                false,
                true,
                true,
                false,
                true,
                true,
                true,
                true,
                false,
                false,
                true,
                true,
                true,
                false,
                false,
                true,
                false,
                true,
                false,
                true,
                false,
                false,
                false,
                false,
                true,
                false,
                false,
                true,
                true,
                false,
                false,
                true,
                false,
                true,
```

```
                false,
                false,
                false,
                true,
                true,
                false,
                true,
                true,
                true
            ],
            "t3a2": 86040820.0,
            "t3a3": true,
            "t3a4": null,
            "t3a5": "GXmH6PSxgBd",
            "t3a6": 21071659,
            "t3a7": true,
            "validation_model": true
        },
        "random_model_2f2cde6cc9014f7a98fa218a5ae304d6": {
            "data_uuid": "eaaad889-b8e2-448d-aee4-55ad5d082f3a",
            "t4a0": [
                "2mq6A7GbasY6wkncPLqc8r0ekvPjsIGhTCyyLi8su",
                "oqg5LNbMoCUjBPB5LJjmSNGCL7pFvOoW",
                "V8edlSHmX6pwWcBMDSlQXhnmQOdOUF
                b2lHZgUmD1HswTH0gRyOm41CJ16HnA7bTxQm8",
                "KwZIJgAB1Bwa",
                "fn6",
                "Dxh",
                "wi8QTdHel8w5yWekZaMFr82B5PmjcNM6hGBzIUtrEW2mEkQnmg
                EW1mJN30to97sSvr4RKbQugPvAYgWVK3gtLyC6",
                "zBpMYbwTC6QyDV7w6YkFkQao4Dfck9Cg8eTN5XVBwHNBURU0MI2sJ2vt",
                "Puvr9xmarYh5Ld0FF",
                "3Dw3rYLrPQMYmTcmrQ6MKDzYp1DANGTUrbT",
                "h67fwghrA9q0Gp9rc1rpBS8rDEYxvqp2PwLnMX1LrmLMWkATJ",
                "4CWHXUB9",
                "s",
                "1SWT8hf5BZSIrJZ4LsSgvJ5pnKLef9Hu0AqK7",
                "lCqlpnPt7ei1XfFabQMqDKBidfEOJaJAUwza",
                "Lvp07IlPf0ZZK257wxRhlSYsEJ7E4"
```

```
            ],
            "t4a1": 30197612,
            "t4a2": 59900225.0,
            "t4a3": "1egNJK4Uq05vkeEVAo7GAPStsiDPlVBEun0cYyj1LV4JTep1nzFyE
            qIjk7UOiUfoPs5LsAkyBgNjJKAsPoIZAFKXVkWa808lwmrT",
            "t4a4": false,
            "t4a5": 48657761,
            "t4a6": "uWKiLse4kbbFv580ELwmSrLNTrPAAc7e
            RSU0bUyZ9cAY9bkeBbiBOdC4PffidCvPJ",
            "t4a7": "7SI5Wh",
            "validation_model": true
        },
        "random_model_8af5a6c4f36e4d3792052e5ade486b69": {
            "data_uuid": "b9d5c609-0e71-4ffb-93fe-40e7c364a5b5",
            "t5a0": "enKF1gz2IvXw40PyJ1JBQQHy3JJe07TlbX0naKP9J
            GtkHsGGh5Afdobn34dsd7lsLx9eubBaK",
            "t5a1": 23121770,
            "t5a2": -64692867,
            "t5a3": false,
            "t5a4": -67702000.0,
            "t5a5": -95912701,
            "t5a6": "oDOQ8",
            "validation_model": true
        },
        [...]
        "validation_model": true
    },
    "random_model_929eab692ccf4be2b9a074c174c49876": {
        "data_uuid": "51eee250-9444-4815-b399-4433ce681fbc",
        "t7a0": 95210841.0,
        "t7a1": null,
        "validation_model": true
    },
    "validation_model": true
}
```

APÊNDICE C – ARTIGO NO FORMATO SBC

Uma Proposta para Mapeamento de Documentos JSON para Bancos de Dados de Grafos de Propriedades

Patrick V. Leguizamon[1]

[1]Departamento de Informática e Estatıstica (INE)
Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

patrick.leguizamon@grad.ufsc.br

Abstract. *NoSQL databases have existed for a long time and their features enable efficient storage of various data formats, such as JSON or CSV documents, and graphic. This work presents a solution for the automatic mapping of JSON document data to the graph model, considering an application that aims to migrate JSON data, widely used by document-oriented NoSQL databases, to graph-oriented NoSQL databases. The focus is on persisting these data in the Neo4j database management system, which is the leading graph-oriented NoSQL database in the industry. Preliminary evaluation demonstrates that the solution is scalable.*

Resumo. *Bancos de dados NoSQL existem há muito tempo e suas características permitem que diferentes formatos de dados possam ser armazenados de forma eficiente para consumo, como documentos JSON ou CSV, e grafos. Este trabalho apresenta uma solução para o mapeamento automático de dados no formato de documento JSON para o modelo de grafo, considerando uma aplicação que deseja migrar dados JSON, amplamente utilizados por bancos de dados NoSQL orientados a documentos, para bancos de dados NoSQL orientados a grafos. O foco deste trabalho é a persistência destes dados no sistema de gerência de bancos de dados Neo4j, que é o principal representante de bancos de dados NoSQL orientados a grafos na indústria. Uma avaliação preliminar demonstra que a solução é escalável.*

1. Introdução

Em junho de 1970, Edgar F. Codd publicou um artigo chamado ”A Relational Model of Data for Large Shared Banks”, que introduziu o modelo relacional de dados, que segundo ele deveria proteger os usuários dos detalhes de armazenamento interno, assegurando a continuidade das aplicações mesmo com mudanças no armazenamento [Codd 1970]. Esse modelo possibilitou avanços como transações ACID e a linguagem SQL, sendo amplamente utilizado até hoje.

Os bancos de dados relacionais inicialmente possuíam escalabilidade vertical, projetados para um único servidor que podia ser expandido adicionando recursos [Pokorny 2011]. Com o aumento da internet e a geração de grandes volumes de dados, especialmente em empresas focadas na análise de dados, surgiram novos desafios para esses bancos, como problemas de performance e dificuldade em lidar com diferentes tipos de dados. Os bancos de dados relacionais não podiam ter seu esquema modificado

com o tempo e também não lidavam bem com diferentes tipos de dados. Por conta disso, novas tecnologias surgiram para solucionar esses problemas, sendo uma delas os bancos de dados NoSQL [Sahatqija et al. 2018].

NoSQL é uma família de bancos de dados caracterizados pela flexibilidade no armazenamento de dados e pela escalabilidade, sendo amplamente utilizados em contextos de Big Data [Abdelhedi et al. 2017]. Eles armazenam dados de forma diferente dos bancos relacionais, permitindo persistência e recuperação independentes da estrutura e do conteúdo [MUS 2019].

Os bancos de dados NoSQL podem armazenar dados em pares chave-valor, conjuntos de colunas, documentos ou grafos, priorizando alta disponibilidade e escalabilidade, muitas vezes sacrificando a consistência dos dados [Meier and Kaufmann 2019]. Entre os principais tipos de bancos de dados NoSQL estão os bancos de documentos, que armazenam dados complexos em formatos como JSON, e os bancos de grafos, que representam dados como nós e relações entre eles [Meier and Kaufmann 2019].

Os bancos de dados de documentos, apesar de não terem esquemas rígidos, utilizam convenções que facilitam a análise de dados. Alguns dados apresentam estruturas que seriam mais naturalmente representadas em formato de grafos [De Virgilio et al. 2013], justificando a adoção de bancos de dados orientados a grafos.

No entanto, migrar dados de bancos de dados de documentos para bancos de dados de grafos apresenta desafios, como a identificação manual de relacionamentos entre dados e a necessidade de aprender novos paradigmas e linguagens. Este trabalho propõe um processo automatizado de mapeamento e migração de dados de bancos de dados de documentos para bancos de dados de grafos, focando nos sistemas MongoDB e Neo4j, que são representativos em suas respectivas categorias e de código aberto. O MongoDB armazena dados em documentos JSON, enquanto Neo4j utiliza uma estrutura de grafos.

1.1. Trabalhos relacionados

A Tabela 1 a seguir apresenta um comparativo das principais características observadas nos trabalhos relacionados ao mapeamento de modelos de bancos de dados para o modelo de grafo de propriedades. As características consideradas são as seguintes: *(i) origem; (ii) estrutura; (iii) linguagem de programação; (iv) destino; (v) modo de definição de esquema; (vi) modo de execução;* e *(vii) forma de execução*.

Tabela 1. Comparação entre trabalhos correlatos

Trabalho	(ÜNAL; OĞUZTÜZÜN, 2018)	(QUEIROZ *et al.*, 2022)	(ALOTAIBI; PARDEDE, 2019)	**Este trabalho**
Origem	MySQL	Postgres	Oracle Live	**MongoDB**
Estrutura	Estruturado	Estruturado	Estruturado	**Semiestruturado**
Linguagem de programação	Java	Python	SQL (execução manual de query)	**Python**
Destino	Neo4J	Dgraph	Neo4J	**Neo4J**
Modo de definição de esquema	Automatizado (Crawler)	Manual (Arquivo de configuração)	Manual	**Automático (evolução de esquema em *streaming*)**
Modo de execução	Sem informação	Local	Manual	**Local ou Distribuído**
Forma de execução	Em lote	Em lote	Manual	***Streaming***

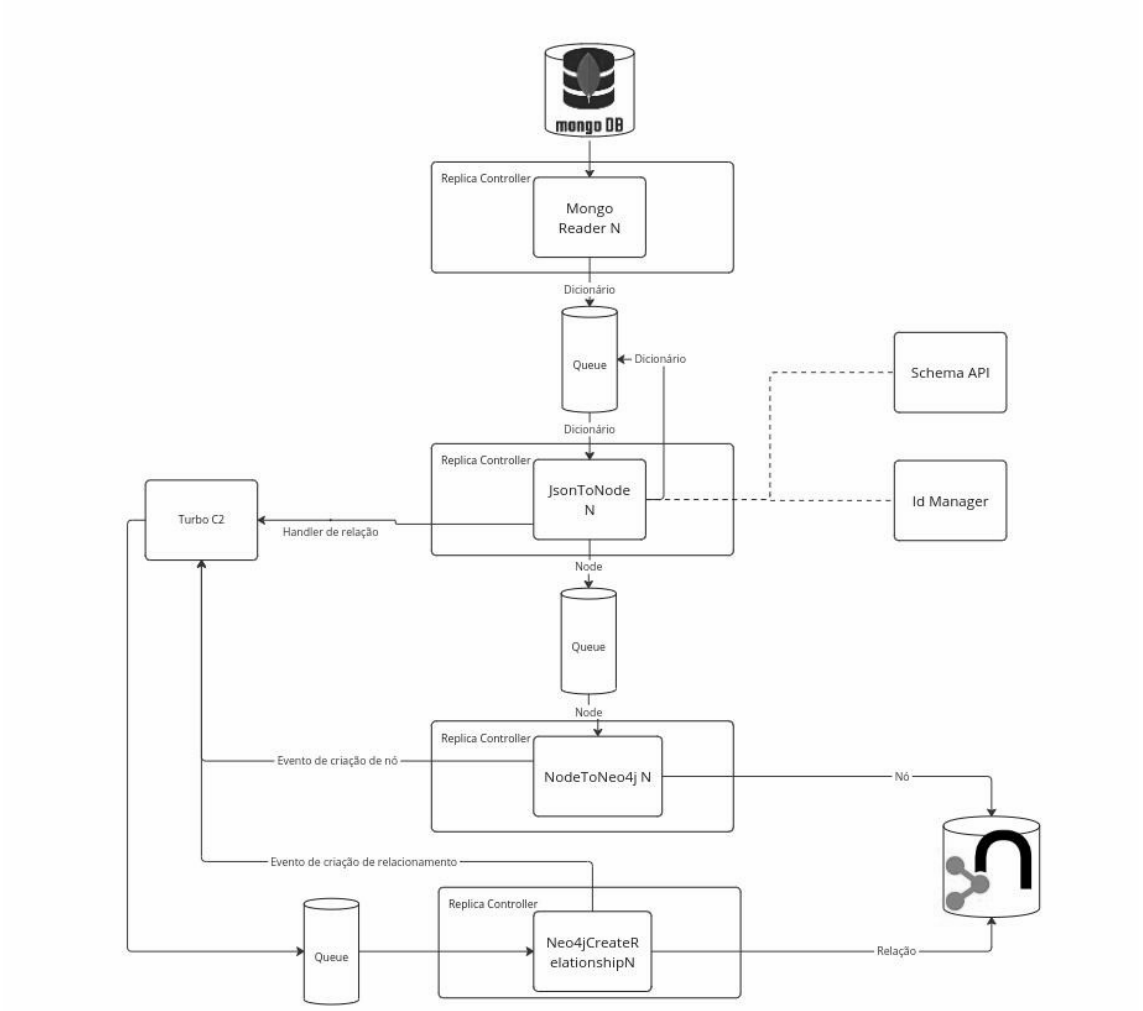
Como é possível notar na Tabela 1, este trabalho apresenta características bem

diferentes em relação aos estudos encontrados que haviam algum tipo de mapeamento para o banco de dados de grafo, sendo seu principal diferencial lidar com dados de entrada semiestruturados no formato JSON, ser completamente automático, construir o esquema dos dados durante a migração (evolução de esquema em v), a possibilidade de rodar localmente ou de forma distribuída, sendo a distribuição gerenciada pelo *framework Ray*[1], e a possibilidade de rodar como *streaming* ao invés de apenas em lote.

2. Ferramenta

A ferramenta proposta é composta por um *script* em *Python* de migração de documentos JSON que estão presentes no banco de dados MongoDB para o banco de dados de grafos Neo4j de forma concorrente, distribuída e em streaming sem arquivo de configurações ou instruções pré definidas. A arquitetura desse *script* é orientada a eventos, utilizando estratégias para execução paralela dos componentes por meio de outros processos gerenciados pelo *framework Ray*, e utilizando filas para o balanceamento do trabalho, conforme mostra a Figura 1.

Figura 1. Arquitetura distribuída e escalável baseada em eventos



[1]https://www.ray.io/

Para viabilizar essa arquitetura foi desenvolvido, por intermédio do *framework Ray*, um *framework* de orquestração distribuído que executa códigos em Python de forma assíncrona, paralela e concorrente, com fácil configuração de escalabilidade. Este *framework*, chamado de *Turbo C2*[2], é utilizado pela solução de mapeamento proposta neste trabalho, sendo disponibilizado como código aberto.

2.1. Turbo C2

O *framework Turbo C2* utiliza o modelo de atores para distribuição de trabalho e assim auxiliar a criação de algoritmos escaláveis. Para isso, ele disponibiliza uma *API* baseada em decoradores da linguagem *Python* para sinalizar a existência de três elementos que fazem o auxilio para a criação de algoritmos escaláveis, sendo eles o *job*, *fila* e *handler*.

Qualquer função pode ser um *job*, sendo ele um ator que pode realizar computações e gerenciar estado interno, executando constantemente de forma paralela em outro processo *Python*. Todos os componentes da ferramenta são *jobs*, sendo sua representação presente na Figura 1. A Figura 2 mostra uma função *Python* que se transformou em um *job*.

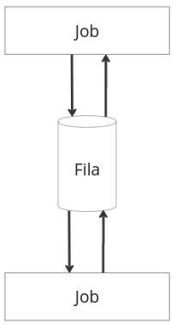
Figura 2. Job que realiza o monitoramento do prazo

```python
# ------------------------------------- Job que monitora data -------------------------------------
@job(
    wait_time=24 * 60 * 60,
    replicas=1,
    meta={"dispatches": [PrazoParaPostagemDeRelatorioDeTCC1Chegando]},
    prazo_final=datetime.datetime.fromisoformat("2024-05-01T23:59:59"),
    data_para_enviar_email=datetime.datetime.fromisoformat("2024-04-24T00:00:00"),
)
def monitorar_prazo_para_postagem_de_relatorio_de_tcc_1(
    prazo_final: datetime.datetime,
    data_para_enviar_email: datetime.datetime,
):
    agora = datetime.datetime.now()

    if agora >= data_para_enviar_email and agora < prazo_final:
        return PrazoParaPostagemDeRelatorioDeTCC1Chegando(prazo_final)

```

Uma fila, por sua vez, pode ser criada de forma natural, quando referenciada em uma instancia ou no encadeamento de *jobs*, ou por meio do decorador @*queue*. Cada fila é um *ator*, que pode executar em um processo separado ou até mesmo em outra máquina. Ela é responsável por armazenar dados e disponibilizar para os atores, conforme mostra a Figura 3.

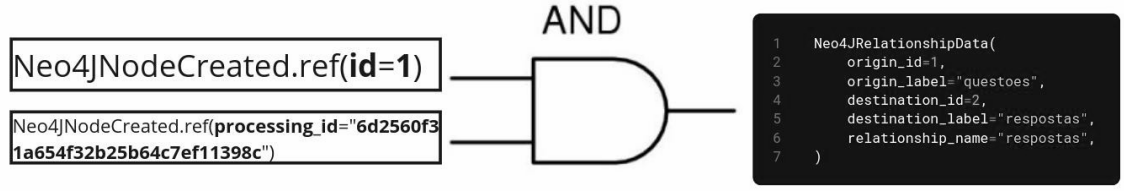
[2]https://github.com/gorgonun/turbo-c2

Figura 3. Relação entre filas e jobs



Já um *handler* pode ser criado por meio de um decorador chamado *@when*. Ele é composto por uma sentença lógica, sendo ela composta por operadores de eventos ou um predicado (caso seja atômica). A Figura 4 mostra o *handler* utilizado pela ferramenta para a escrita assíncrona de arestas. Ela usa o identificador único do objeto pai e o id de processamento do objeto filho e quando ambos são criados o *handler* executa, enviando os dados de relacionamento para o componente *Neo4jCreateRelationship*.

Figura 4. Handler criado que espera pelos eventos de questoes e respostas

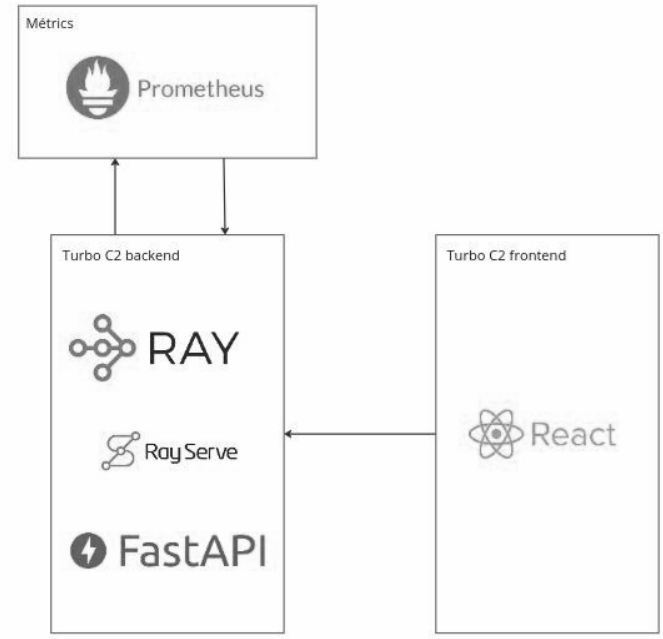


Ele pode estar associado a comportamentos quando a sentença é verdadeira ou falsa. Para que a sentença seja avaliada, é necessário que um evento que possua dados iguais aos definidos de forma explícita na referencia aconteça. Caso todos os eventos aconteçam, a sentença é avaliada como verdadeira e as instruções definidas são executadas.

O *Turbo C2* também disponibiliza uma *API* para declaração de *endpoints HTTP* com o auxilio das bibliotecas *FastApi* e *Pydantic*, além da biblioteca *Ray Serve* para a distribuição e controle de réplicas, conforme mostra a Figura 5. Para a coleta de métricas de recursos utilizados, como CPU e memória, e para alimentar os *dashboards* com a contagem de nós e arestas, foi utilizada a ferramenta *Prometheus*[3] devido a sua integração nativa com o *framework Ray* e a sua facilidade de uso.

[3]https://prometheus.io/

Figura 5. Arquitetura de alto nível do Turbo C2



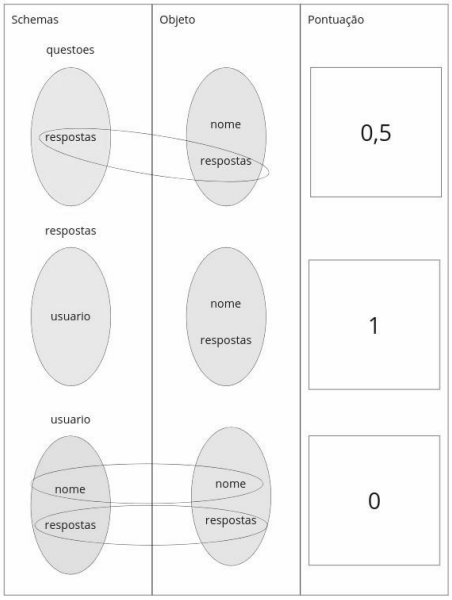
2.2. Componentes

Cada parte principal da migração foi separada em um componente, sendo todos eles *jobs* com filas entre si. O primeiro componente é o *MongoDB Reader*, responsável por ler dados BSON provenientes do MongoDB e os transformar em um formato que a aplicação consegue processar (dicionário Python).

Seguido a este componente está o *JsonToNode*, que recebe dados no formato de dicionário Python e processa suas informações, transformando-as em uma estrutura interna (Node) por onde é possível aplicar as regras de mapeamento. Para que ele funcione é necessario dois componentes externos, sendo eles a *SchemaAPI*, que identifica objetos JSON por meio da pontuação de diferença de seus atributos, e o *Id Manager*, que gerencia identificadores únicos sequências criados pela aplicação.

Para a *SchemaAPI* identificar objetos JSON ela calcula uma pontuação de diferença entre os *schemas* já identificados e os atributos do objeto. Essa pontuação é obtida por meio da relação entre a união da diferença dos atributos entre esquema e objeto sobre a união de todos os atributos únicos, sendo que o esquema com menor pontuação é selecionado, desde que a diferença seja menor que o limite estabelecido. Na Figura 6 está a pontuação resultante de um exemplo. Como o esquema **usuario** possui a pontuação de 0, enquanto que **questoes** tem a pontuação de 0,5, o *Schema* será **usuario**.

Figura 6. Pontuação para identificar um objeto usuario



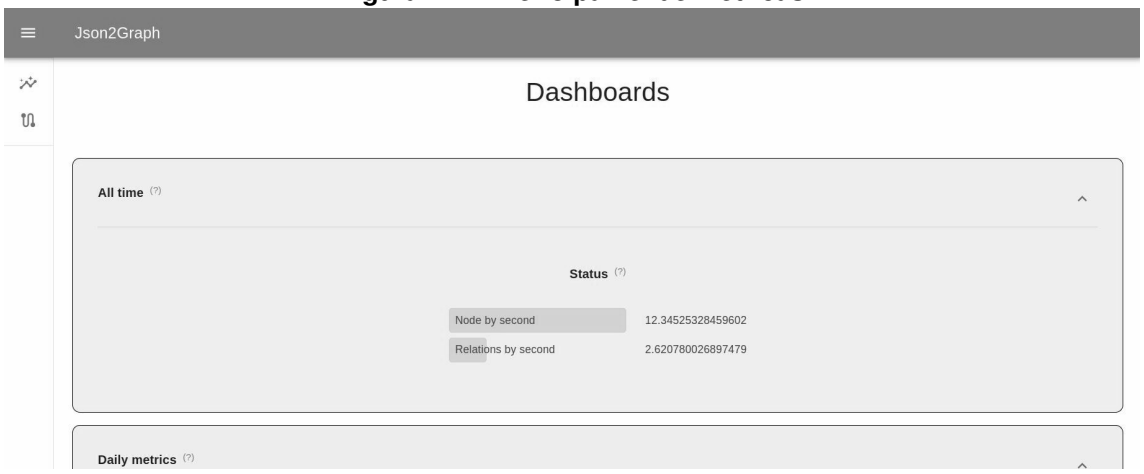
Após esse componente está o *NodeToNeo4j*, que recebe a estrutura interna Node e cria as consultas necessárias para a escrita do nó no Neo4j.

Para a escrita de arestas está o componente *Neo4jCreateRelationship*, que por meio do *Turbo C2* recebe informações de arestas e faz a consulta para sua criação de forma assíncrona.

2.3. Aplicação Web para visualização e gerenciamento de migrações

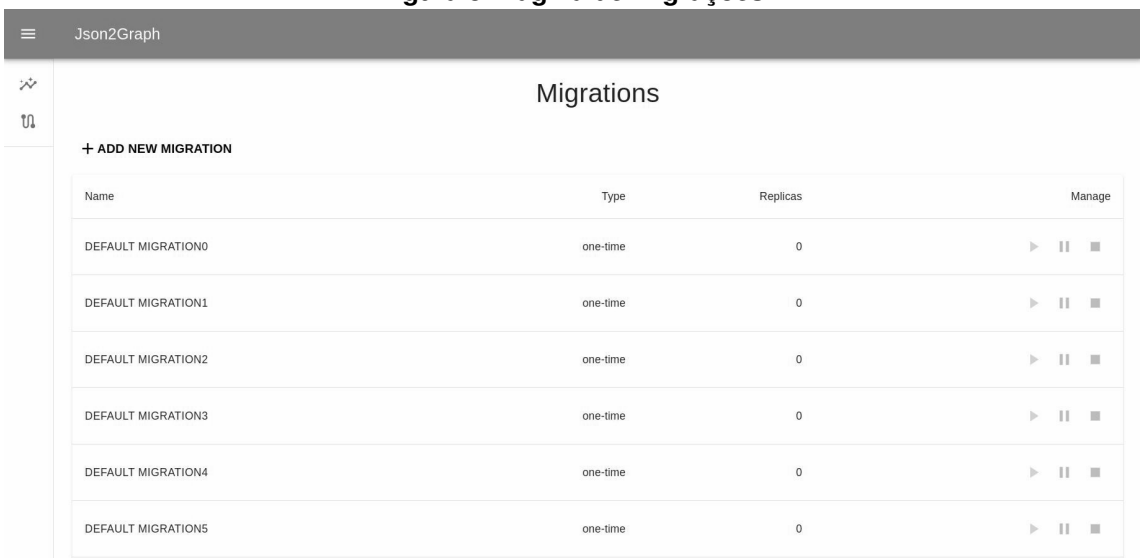
Para o gerenciamento de mapeamentos, migrações e análise das métricas, foi disponibilizada uma aplicação Web que se comunica com uma *API* presente na aplicação responsável por fazer as migrações. Essa aplicação é composta por duas páginas, uma contendo diversos painéis referente às métricas coletadas durante as migrações e a outra contendo ferramentas para visualização e gerenciamento de migrações. A Figura 7 mostra a primeira página da aplicação, que contém algumas métricas por migração.

Figura 7. Primeiro painel de métricas



Ja a pagina de migrações contém uma listagem de todas as migrações criadas, que podem ser oriundas da aplicação Web ou do código. Além disso, é possível obter informações dos nomes das migrações, o seu tipo, o número total de réplicas em todas as instâncias dos *jobs*, fazer o gerenciamento por meios dos botões de *play*, *pause* e *stop* e, por fim, criar novas migrações. Ela é demonstrada na Figura 8.

Figura 8. Página de migrações



3. Regras de mapeamento

O processo de mapeamento de objetos JSON para o esquema de bancos de dados de grafo aplica o seguinte conjunto de regras de mapeamento, que é uma adaptação das regras apresentadas em [Alotaibi and Pardede 2019]:

- Um objeto JSON mantido em uma coleção de nome C_x se torna um nó cujo rótulo é C_x;
- Um atributo simples de um objeto JSON O_i se torna uma propriedade do nó correspondente à O_i;
- Um atributo a_n de um objeto JSON O_i que mantém um objeto aninhado gera:
  - Um nó O_j cujo rótulo é o nome de a_n;
  - Uma aresta direcionada de O_i e O_j.

Para fim de exemplificação e simulação do algoritmo, a Figura 9 mostra um dado que representa uma **questão** fictícia guardada em uma coleção chamada de **questões**, sendo que essas questões possuem dados aninhados, sendo eles **respostas** e dados aninhados em **respostas** (**usuário**). Esse dado (**questão**) é um elemento da coleção que está sendo iterada, e ele é transformado em uma entrada no dicionário e irá alimentar o próximo componente.

Coleção questões

```
{
    "respostas": [
        {
            "usuario": {
                "nome": "__snoopyDog",
                "respostas": 10
            }
        },
        {
            "usuario": {
                "nome": "elchatgpt",
                "respostas": 7
            }
        }
    ]
}
```

Figura 9. Representação da coleção questões em JSON

Esta solução realiza a escrita de nós por meio da árvore criada a partir dos objetos JSON aninhados. Para os dados de *questoes* (Figura 9), o nó patriarca é o primeiro a ser escrito no *Neo4j*. Na Figura 10 estão todos os *Nodes* resultantes do processamento do objeto JSON no componente *JsonToNode*, sendo o patriarca (**questoes**) detentor do id 1, as **respostas** id 2 e o **usuario** id 3. Essa árvore de *Node* é utilizada como ponto de partida para a escrita dos nós e arestas.

```
Node(
    id=1,
    table=Table(
        id=1,
        name="questões",
        schema=Schema(
            id=1,
            column_information={
                "respostas": Column(
                    name="respostas",
                    is_optional=True,
                    is_relationship=True
                )
            }
        )
    ),
    relationship_references={"respostas": RelationshipReference(table_name="respostas", node_id=2)},
    relations=Node(
        id=2,
        table=Table(
            id=2,
            name="respostas",
            schema=Schema(
                id=2,
                column_information={
                    "usuario": Column(
                        name="usuario",
                        is_optional=True,
                        is_relationship=True
                    )
                }
            )
        ),
        common_values={},
        relationship_references={"usuario": RelationshipReference(table_name="usuario", node_id=3)},
        relations=[
            Node(
                id=3,
                table=Table(…
                common_values={
                    "nome": "__snoopyDog",
                    "respostas": 10
                },
                relations=[],
                relationship_references={}
            )
        ]
    )
)
```

Figura 10. Resultado do componente JsonToNode para o objeto raiz

Para a Figura 9, o mapeamento completo do documento *JSON* de *questoes* é mostrado na Figura 11.

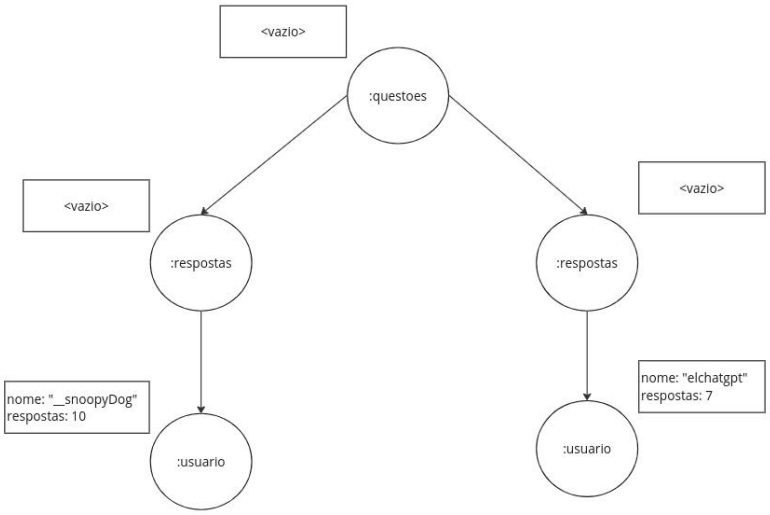


Figura 11. Mapeamento do documento JSON para nós do Neo4J

4. Conclusão

Este trabalho teve como objetivo o desenvolvimento de uma ferramenta que permite a migração de dados semiestruturados, sendo o foco nos tipos *JSON*, armazenados em um banco de dados, sendo o banco de dados escolhido *MongoDB*, para um banco de dados grafos, no caso o banco de dados *Neo4j*. O código da aplicação está disponível em um repositório público do GitHub[4].

As principais contribuições desse trabalho são: desenvolvimento de uma *pipeline* de processamento de dados *JSON* aninhados, as regras de mapeamento entre o modelo de dados JSON e o modelo de dados de grafos de propriedades, a arquitetura distribuída em *streaming* utilizando a linguagem de programação, a execução de código assíncrono com base em sentenças lógicas provenientes da estrutura de *eventos*, a interface *web* que possui painéis e ferramentas para criação e gerenciamento de migrações e a ferramenta *Json2Node*, que realiza a migração de dados em *streaming* com execução concorrente e distribuída sem a necessidade de nenhuma informação sobre os dados que serão migrados.

Como atividades futuras relacionadas a esse trabalho, podemos considerar: a implementação de leitura concorrente no *MongoDB* pelo módulo *MongoDBProducer*, a criação de mais painéis voltados a uma análise mais aprofundada das métricas, a adição de métricas de acompanhamento, a análise de desempenho de cada módulo buscando aperfeiçoar sua execução, melhorias no algoritmo de detecção de tabelas e esquemas, buscando se tornar resiliente aos dados *JSON* com campos nulos escondidos e a possibilidade de criar relações com nomes diferentes dos padrões, podendo o usuário definir ou então a escolha por meio de um modelo de linguagem.

Referências

Abdelhedi, F., Brahim, A. A., Atigui, F., and Zurfluh, G. (2017). Umltonosql: Automatic transformation of conceptual schema to nosql databases. In *2017 IEEE/ACS 14th In-*

[4]https://github.com/gorgonun/json-to-graph

ternational Conference on Computer Systems and Applications (AICCSA), pages 272–279.

Alotaibi, O. and Pardede, E. (2019). Transformation of schema from relational database (rdb) to nosql databases. *Data*, 4(4).

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

De Virgilio, R., Maccioni, A., and Torlone, R. (2013). Converting relational to graph databases. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, New York, NY, USA. Association for Computing Machinery.

Meier, A. and Kaufmann, M. (2019). *SQL & NoSQL databases*. Springer.

MUS, M. (2019). Comparison between sql and nosql databases and their relationship with big data analytics.

Pokorny, J. (2011). Nosql databases: A step to database scalability in web environment. In *Proceedings of the 13th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS '11, page 278–283, New York, NY, USA. Association for Computing Machinery.

Sahatqija, K., Ajdari, J., Zenuni, X., Raufi, B., and Ismaili, F. (2018). Comparison between relational and nosql databases. In *2018 41st international convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 0216–0221. IEEE.