Comparativo de desenvolvimento e desempenho entre MongoDB e PostgreSQL em aplicações com mapeamento de objetos

Pedro L. L. Matias[1]

[1] Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)

pedrollmatias@gmail.com

Abstract. *Non-relational databases have been frequent choices in the composition of software, becoming an alternative to consolidated relational databases. With the aim of increasing productivity and narrowing the differences in the implementation level between the application layer and the data layer, the use of object mappers, called ORMs and ODMs, has become common. The proposed work analyze a Node.js web application which uses the PostgreSQL relational database with ORM Sequelize.js and the non-relational MongoDB with ODM Mongoose, evaluating differences in complexity, consistency, maintainability, performance and usability between technologies and demonstrating how the results obtained can contribute to systems architecture decisions.*

Resumo. *Bancos de dados não-relacionais têm sido escolhas frequentes na composição de softwares, tornando-se uma alternativa aos consolidados bancos de dados relacionais. Com o objetivo de aumentar a produtividade e estreitar as diferenças em nível de implementação entre a camada de aplicação e a camada de dados, o uso de mapeadores de objetos, chamados ORMs e ODMs, vem sendo comum. O trabalho proposto analisa uma aplicação web Node.js que utiliza o banco de dados relacional PostgreSQL com o ORM Sequelize.js e o não-relacional MongoDB com o ODM Mongoose, avaliando as diferenças de complexidade, consistência, manutenibilidade, performance e usabilidade entre as tecnologias e demonstrando como os resultados obtidos podem contribuir em decisões de arquitetura de sistemas.*

1. Introdução

Os bancos de dados ocupam uma posição de destaque em relação aos recursos utilizados no ambiente da tecnologia da informação para a construção de software. A escolha de quais bancos de dados irão compor a camada de persistência de uma aplicação é fundamental e deve ser consciente e criteriosa para atender aos requisitos do sistema de forma escalável, performática, produtiva e consistente.

De acrodo com [Date 2003], um sistema gerenciador de banco de dados (SGBD) é um software genérico para manipular banco de dados. Existem diversos SGBDS, tais como MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, SQLite, MongoDB, Cassandra, Redis, CouchDB, dentre outros. Devido a esta gama de opções, existe uma discussão envolvendo as diferenças e semelhanças no que diz respeito ao armazenamento de informações em bancos de dados que utilizam Structured Query Language (SQL),

quando comparados com abordagens mais recentes que não utilizam SQL nem o modelo relacional de dados para realização de consultas, como é o caso dos bancos de dados não-relacionais, chamados também de Not-only SQL, ou simplesmente NoSQL.

Além de diferenças de modelagem de dados, que resultam em maneiras distintas de realizar consultas a dados, há ainda ferramentas que auxiliam os desenvolvedores de software a tornar o processo de implementação e escrita de código mais eficiente. Existem bibliotecas e *frameworks* que atuam como uma camada intermediária entre a linguagem de programação e o banco de dados. Essa camada é conhecida como mapeadora de objeto e podem ser *Object-Relational Mapping* (ORM), para os bancos relacionais, e *Object-Document Mapping* (ODM), para bancos não-relacionais orientados a documentos. A Figura 1 ilustra a arquitetura de uma aplicação *web* em três camadas, utilizada com objetivo de promover a separação das funcionalidades em camadas de lógica de apresentação, lógica de negócio (aplicação) e lógica de acesso a dados [Lemos et al. 2013].

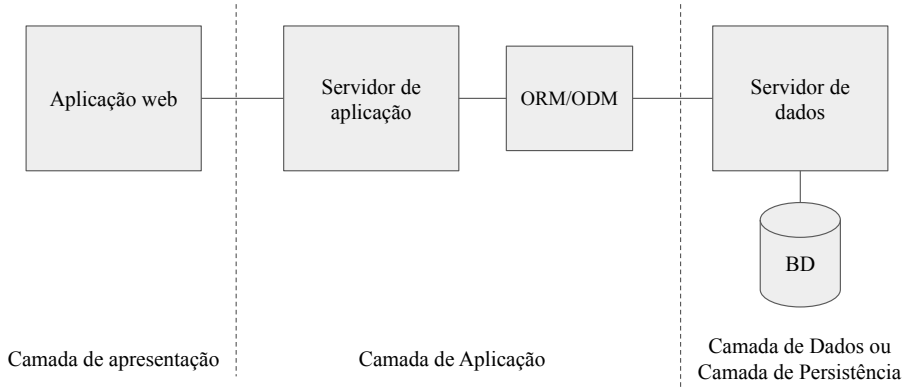

Figura 1. Arquitetura de uma aplicação *web* em três camadas

Devido a bancos de dados serem parte central dos sistemas de informação, é conveniente que se faça uma análise minuciosa a respeito dos impactos gerados em uma aplicação pela escolha de um banco em detrimento de outro, observando as diferenças conceituais, de produtividade, performance e ainda se podem haver diferenças que serão perceptíveis para o usuário final (como diferenças de desempenho nítidas em consultas, por exemplo).

Desse modo, a proposta deste trabalho é realizar uma comparação no uso de bancos de dados, a partir da construção de uma aplicação *web* que consuma os mesmos dados de fontes diferentes, sendo a primeira fonte um banco de dados SQL *PostgreSQL* e a outra fonte, de mesmos dados, um banco não-relacional MongoDB, ambas utilizando mapeadores de objetos na camada de aplicação. A partir do desenvolvimento dessa aplicação, será possível realizar uma comparação considerando fatores como modelagem, complexidade de implementação, consistência, performance e usabilidade entre os bancos e suas ferramentas.

1.1. Motivação

É importante elencar alguns fatores que respaldam e justificam a decisão de desenvolvimento deste trabalho. Cada banco de dados é melhor explorado dependendo das circunstâncias em que estão submetidos. Fatores como propósitos da aplicação, volume

de dados e necessidade de consistência podem tornar no uso de um SGBD específico mais adequado que outro, por exemplo. [Anderson and Nicholson 2020] descreveram, de forma geral, alguns prós e contras gerais dos modelos SQL e NoSQL os quais são descritos na Tabela 1 e 2.

Tabela 1. Prós e contras do SQL

Prós	**Contras**
Espaço reduzido de armazenamento de dados devido à normalização e outras oportunidades de otimização	Modelos de dados rígidos que requerem um design inicial cuidadoso para garantir um desempenho adequado e resistir à evolução
Semântica de integridade de dados forte e bem compreendida por meio das propriedades ACID (*Atomicity, Consistency, Isolation, Durability*) [Date 2003]	O dimensionamento horizontal é um desafio, pouco explorado e ainda imaturo - as vezes não há suporte ou ainda apenas suporte com utilização de tecnologias emergentes
Acesso padrão aos dados via SQL	As *engines* não distribuídas são geralmente um ”ponto único de falha”que deve ser mitigado por técnicas de replicação e *fail over*
Geralmente, suporte de consulta mais flexível, capaz de lidar com uma gama mais ampla de cargas de trabalho. O SQL abstrai a implementação subjacente e permite que o mecanismo otimize as consultas para caber em sua representação no disco	

Tabela 2. Prós e contras do NoSQL

Prós	**Contras**
Altamente escaláveis - muitos bancos de dados NoSQL geralmente são projetados para oferecer suporte a escalabilidade horizontal contínua, sem pontos únicos de falha significativos	Interpretações vagas das restrições do ACID - apesar das alegações generalizadas de suporte do ACID para sistemas NoSQL, a interpretação do ACID costuma ser tão ampla que não se pode colher muito sobre a semântica do banco de dados em questão
Modelos de dados flexíveis - a maioria dos sistemas não relacionais não exige que os desenvolvedores façam compromissos iniciais com os modelos de dados; os esquemas que existem podem ser alterados rapidamente	Exige entendimento por parte dos desenvolvedores de como o sistema distribuído NoSQL funciona. Em determinados casos é necessário compreender profundamente, por exemplo, como serão garantidas a consistência, disponibilidade e tolerância de partição pelo banco de dados
Alto desempenho - limitando o intervalo do que o banco de dados pode fazer (por exemplo, relaxando as garantias de durabilidade), muitos sistemas NoSQL são capazes de atingir níveis extremamente altos de desempenho	Falta de flexibilidade nos padrões de acesso - a abstração relacional / SQL dá ao mecanismo de banco de dados amplos poderes para otimizar consultas para os dados subjacentes
Abstrações de dados de alto nível - indo além do modelo de dados ”valor em uma célula”, os sistemas NoSQL podem fornecer APIs de alto nível para estruturas de dados poderosas	

Mediante esse cenário, torna-se necessário responder o seguinte questionamento: se cada tipo de banco de dados tem casos de uso específicos, por que há necessidade de desenvolver um estudo comparativo quando as aplicabilidades são distintas? A resposta para esta pergunta passa pelas motivações deste estudo.

Com o avanço da tecnologia, os bancos NoSQL tornaram-se cada vez mais maduros e incorporam novas funcionalidades. Além disso, novas soluções surgem e a gama de tipos de bancos de dados e SGBDS aumentam. Esses fatores fazem com que a escolha por determinada solução em detrimento de outra possa não ser simples.

Dizer que uma tecnologia é mais performática que outra significa também que menos poder de processamento é necessário para obter os mesmos resultados da tecnologia de menor performance.

Outro ponto importante está relacionado ao mercado de trabalho e como as tecnologias são empregadas nas empresas. A diferença de performance entre tecnologias comparáveis está relacionada indiretamente com os custos de infraestrutura. Dizer que uma tecnologia é mais performática que outra significa também que menos poder de processamento é necessário para obter os mesmos resultados da tecnologia de menor performance. Um poder de processamento mais baixo implica em uma infraestrutura mais simples, o que consequentemente reduz custos com infraestrutura. Portanto, não é incomum as empresas optarem por soluções menos custosas, mesmo que possam não ser as mais adequadas. Apesar de haver muitas variáveis para cálculo de custos, gastos com hardware não deixam de ser relevantes.

Ainda, quando a decisão pela escolha de um banco de dados é tomada, os impactos gerados por tal escolha são percebidos a fundo, afinal os impactos passam do âmbito teórico para o âmbito prático e empírico. Porém, como mensurar os impactos gerados por uma possível escolha alternativa? Como essas alternativas poderiam ser comparados na prática? Portanto, uma comparação nesse nível é possível apenas se um mesmo sistema for construído tomando os dois caminhos simultaneamente.

Por fim, alguns bancos de dados NoSQL possuem a vantagem de ter uma sintaxe para escrita de consultas mais simples quando comparados à sintaxe SQL. Porém, com o advento dos ORMs, é possível reduzir a dificuldade da escrita de consultas. Ora, se as diferenças de sintaxe foram estreitadas, e portanto o esforço para a construção de consultas é semelhante em ambos os casos, quais serão as outras diferenças existentes?

Todos esses fatores e questionamentos reforçam a importância do estudo e motivam o trabalho a apresentar contribuições importantes para a área.

2. Referencial teórico

E.F Codd demonstrou os fundamentos da teoria dos bancos de dados relacionais em 1970 [Kumar and Azad 2017]. Um banco de dados relacional é uma coleção de dados organizados e dispostos através de tabelas, cujos registros podem ser acessados e consultados de diferentes formas [Jatana et al. 2012]. Uma base de dados relacional é composta por um conjunto de tabelas referidas como relações, em que os dados são categorizados e divididos em colunas [Jatana et al. 2012]. Cada linha da tabela é constituída do agrupamento dos valores de colunas que possuem significado lógico entre si.

Por outro lado, NoSQL surgiu com o requisito de ser escalável. Algumas estratégias exigem bancos de dados com maior disponibilidade em detrimento de alta consistência, abordagem conhecida como BASE (*Basically Available, Soft-state and Eventually consistent*). Os bancos NoSQL cumprem este papel, sendo bastante distintos de SGBDS relacionais, e podem ser organizados em quatro categorias mais co-

muns: documentos, colunas, chave-valor e grafos. Existem também categorias híbridas que combinam vários modelos de dados, conhecidas como *multi-model databases* [Diogo et al. 2019].

Um banco de dados relacional costuma conter muitas relações, com tuplas nas relações que estão relacionadas de várias maneiras. Também possuem maior consistência no acesso aos dados, uma vez que normalmente não há duplicidade de dados e inconsistências [Elmasri and Navathe 2011], além de maior padronização no uso de dados e menos erros de manipulação nas rotinas de negócio.

Bancos não-relacionais possuem maior flexibilidade para manipulação de dados, permitindo maior facilidade na edição e adição de novas propriedades às entidades, apresentam melhor performance com grandes volumes de dados (*big data*) e comumente não é necessário dedicação de pessoal para administração do banco de dados, conhecidos como *Database Administrators* (DBAs) [Diogo et al. 2019].

Bancos NoSQL tem melhora capacidade em lidar com uma infraestrutura de hardware escalável horizontalmente de forma distribuída com hardware de baixo custo [Gupta et al. 2017]. Por outro lado, SGBDS relacionais escalam verticalmente [Ali et al. 2019] e demandam custos de infraestrutura mais elevado.

Para ambos os tipos de modelos de dados utilizados pelos bancos, existem tecnologias para abstração de consultas. De maneira geral, os ORMs são ferramentas que utilizam uma técnica de mapeamento do paradigma de orientação a objeto para o paradigma relacional, permitindo manipular os dados do banco como objetos, enquanto ODMs são análogos para bancos orientados a documentos. Alguns exemplos dessas ferramentas são o Hibernate, Sequelize, Knex, Django ORM, Mandango, Mongoose, dentre outras [Fonseca 2019].

Em SQL, existem quatro instruções básicas responsáveis por manipular as informações armazenadas no banco de dados: *SELECT, INSERT, UPDATE, DELETE* [Elmasri and Navathe 2011]. Estas instruções, de forma geral, também são chamadas de consultas (ou *queries*) e estes termos também podem ser utilizados no âmbito NoSQL para se referir à consultas análogas.

A forma como desenvolvedor com foco em aplicações *server-side* realiza consultas pode sofrer grandes alterações, já que consultas escritas de maneira nativa são diferentes quando escritas com utilização de um ORM/ODM. Para exemplificar e tornar mais nítida uma atuação prática de um ORM, um código em SQL nativo para atualizar um registro é descrito na instrução 1.

Instrução 1. Exemplo de instrução de atualização utilizando SQL

```
1 UPDATE Users
2 SET firstName = 'Joao', lastName = 'Silva'
3 WHERE userId = 1
```

Essa mesma consulta pode ser realizada com o uso de um ODM de maneira diferente, mas análoga. O Sequelize, um *promise-based* ORM para Node.js [Wu 2019], realiza as mesmas operações da consulta acima conforme descrito na instrução 2.

Instrução 2. Exemplo de instrução de atualização utilizando Sequelize.js

```
1 Users.update(
2     { firstName: 'Joao', lastName: 'Silva' },
3     { where: { userId: 1 } }
4 );
```

Observa-se que a consulta nesse último é realizada a partir da chamada de uma função em que parâmetros definem as especificações da atualização, diferentemente da consulta nativa que é construída com utilização de *keywords* e literais como parâmetros.

Além de diferenças entre a forma de consulta aos dados, existem outras diferenças entre os modelos relacional e não-relacional. Produtividade para escrita de código, performance, desempenho e experiência de usuário são alguns dos pontos importantes que são adjacentes as diferenças entre os modelos. Esses pontos extrapolam a área de banco de dados e estão relacionadas à Engenharia de software, como por exemplo, métricas de desenvolvimento de software.

Métricas de software são utilizadas com o objetivo de controlar e identificar eficientemente os parâmetros essenciais que afetam o desenvolvimento do programa, avaliando partes importantes do processo, como produtivas e gerenciais [Meirelles 2013].

As métricas de software, do ponto de vista de medição, podem ser divididas em duas categorias: medidas diretas e indiretas. Pode-se considerar como medidas diretas do processo de engenharia de software o custo e o esforço aplicados ao desenvolvimento e manutenção do software e do produto, a quantidade de linhas de código produzidas e o total de defeitos registrados durante um determinado período de tempo. Porém, a qualidade e a funcionalidade do software, ou a sua capacidade de manutenção, são mais difíceis de serem avaliadas e só podem ser medidas de forma indireta [Meirelles 2013].

3. Trabalhos relacionados

3.1. *A performance comparison of document-oriented NoSQL databases*

Os bancos NoSQL são reivindicados para terem um desempenho melhor do que os bancos de dados SQL tratando-se de grande volume de dados [Yishan and Manoharan 2017]. Processar uma grande quantidade de dados requer velocidade, esquemas flexíveis e bancos de dados distribuídos. Os bancos não-relacionais se tornaram mais propensos a operar grandes volumes de dados, já que satisfazem esses requisitos.

Para confirmar esse comportamento, a metodologia deste trabalho se baseia na realização de quatro operações básicas em diferentes tipos de bancos de dados: *Create, Retrieve, Update* e *Delete*, comumente conhecidas como CRUD. Essas operações foram realizadas em sete bancos NoSQL distintos e no *Microsoft SQL Server*.

Após a execução das consultas e medição dos tempos necessários para realizar cada uma das operações com cargas variáveis de dados, notou-se que, para volumes de dados pequenos, os tempos foram bastante similares. Entretanto, as diferenças se tornaram mais nítidas à medida que a ordem de grandeza dos dados aumentava nas operações, como, por exemplo, inserir 100.000 instâncias em vez de somente 10 instâncias.

Percebeu-se que os bancos de dados NoSQL geralmente são otimizados para armazenamentos de chave-valor, enquanto os bancos de dados SQL não são. Todavia, observou-se que nem todos os bancos de dados NoSQL têm melhor desempenho do que o banco de dados SQL testado. Mesmo entre bancos de dados NoSQL, há uma grande variação no desempenho com base no tipo de operação, como leitura e gravação, por exemplo.

O trabalho apresentado analisa as consultas dos bancos sem que esses façam parte de uma estrutura maior que relacione as entidades dos bancos em um nível mais alto, como, por exemplo, serem pertencentes a um sistema de informação, um software ou alguma aplicação que exija uma relação mais estreita entre as entidades. Além disso, não há o uso de camadas de abstração para realização das consultas. O foco da pesquisa citada se concentra principalmente na dinâmica de desempenho, que é apenas um dos pontos a serem abordados no estudo aqui desenvolvido.

3.2. *Testing Spatial Data Deliverance in SQL and NoSQL Database Using NodeJS Fullstack Web App*

[Laksono 2018] realizou um estudo quantitativo de comparação de desempenho entre bancos de dados relacionais e não-relacionais empregados em uma aplicação *web fullstack Node.js*, que utiliza grande volume de dados geoespaciais. Bancos de dados relacionais SQL eram conhecidos no passado por lidar muito bem com dados geoespaciais, porém a abundância de *Big Data* geoespacial impulsionou a necessidade de banco de dados NoSQL, em que se espera ter um melhor desempenho em termos de manuseio e armazenamento de *big data* geoespacial.

Para realizar a análise, foi desenvolvido um aplicativo *web* utilizando o *framework* JavaScript Angular para a camada de apresentação e Node.js na camada de aplicação. Foram gerados diferentes números de pontos para testar a capacidade de armazenamento e carregamento de dados geoespaciais nos bancos de dados *PostGIS* e o MongoDB. O teste foi conduzido comparando o resultado de XHR (XML HTTP *Request*) de ambas as bases de dados em cada caso.

A Figura 2 ilustra a arquitetura da aplicação desenvolvida, que utiliza o *framework* Javascript Angular 2+ no *front-end* e Node.js no *back-end* com utilização dos mapeadores de objetos Mongoose e Sequelize.js. Para a camada de persistência, é utilizado o MongoDB e o *PostGIS*, uma variante do PostgreSQL para tratamento de dados geoespaciais. Os dados são inseridos nos bancos via linha de comando.

Os resultados mostraram que o MongoDB tem melhor desempenho no carregamento de grandes dados em comparação com o *PostGIS*, uma vez que conforme o volume de dados aumenta, as diferenças de tempo de execução tornam-se mais discrepantes. O aplicativo mostrou que o banco de dados NoSQL supera o SQL tradicional para a entrega de um grande volume de dados.

Apesar do foco da análise do trabalho ser apenas na camada de banco de dados, desconsiderando o uso das camadas de abstração e mapeadores, o autor utiliza os mesmos ORM (Sequelize) e ODM (Mongoose) que são objeto de estudo desta pesquisa, servindo como base teórica para o desenvolvimento da estrutura e arquitetura do software desenvolvido.

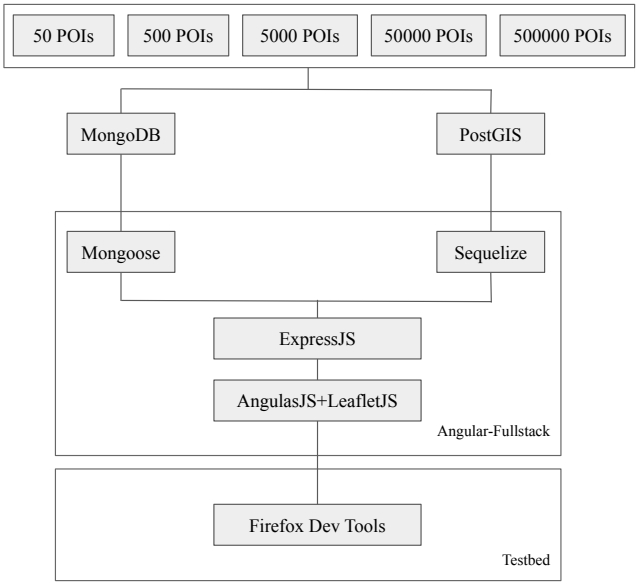

Figura 2. Arquitetura do *web app* para manipulação de dados geoespaciais

3.3. Monitoramento de métricas de código-fonte em projetos de software livre

[Meirelles 2013] apresenta uma abordagem para a observação das métricas de código-fonte, estudando-as através de formas associadas e distribuídas, aplicadas em *softwares* livres. Segundo o autor, embora existam diversas empresas e governos interessados em utilizar componentes de software livre em suas infraestruturas de tecnologia de informação, muitas vezes eles se perdem diante da quantidade de opções disponíveis e a falta de critérios para avaliar as alternativas existentes.

A pesquisa teve como objetivo investigar e avaliar a qualidade de projetos de software livre, tendo como base o estudo de métricas que permitem analisar automaticamente o código-fonte, a partir da criação de uma ferramenta que provem essas informações.

Segundo [Meirelles 2013], o padrão internacional *Software Engineering - Product Quality* define um conjunto de atributos para a avaliação da qualidade de software, para diminuir a subjetividade do conceito de qualidade. Estes atributos são:

- Funcionalidade: execução das funções requeridas.
- Confiança: maturidade, tolerância a falhas e capacidade de recuperação.
- Usabilidade: esforço necessário para operar o software.
- Eficiência: desempenho e utilização dos recursos.
- Manutenibilidade: esforço necessário para modificar o software.
- Portabilidade: esforço necessário para transferir o software de ambiente.

Esses parâmetros são utilizados pelo autor para medir a qualidade dos *softwares* analisados. Foi avaliado a distribuição e a correlação dos valores das métricas de 38 projetos de software livre que tinham mais contribuidores ativos em seus repositórios. Além disso, para mostrar a utilidade do monitoramento de métricas foi descrita uma extensão e adaptação do modelo de causalidade do conceito de atratividade de projetos de software livre, relacionando os valores das métricas de código-fonte e a quantidade de *downloads*, contribuidores e atualizações (*commits*) nos repositórios dos projetos.

O projeto desenvolvido pelo autor permitiu a avaliação da "saúde" do código-fonte de projetos de software livre por órgãos e entidades interessadas no produto, garantindo as funcionalidades previstas em contrato de maneira segura e confiável. Sendo assim, uma funcionalidade incorporada na versão de desenvolvimento somente será integrada ao projeto oficial caso atenda os requisitos estabelecidos pela ferramenta desenvolvida.

O estudo citado fornece uma base para compreender métricas de engenharia de software, como são realizadas as medições e aplicadas em projetos. O atributo eficiência foi avaliado de forma quantitativa neste estudo.

Alguns atributos para medir a qualidade do software utilizadas pelo autor foram empregados neste trabalho, que permitiram avaliação do código gerado de maneira não-arbitrária e com base em um referencial. O atributo eficiência foi avaliado de forma quantitativa, enquanto a confiança e manutenibilidade foram analisados qualitativamente.

4. A Aplicação

Para realizar uma análise aprofundada de todo o ecossistema que envolve ambos os bancos de dados, é importante que a camada de persistência possua entidades que permitam diversidade e complexidade de consultas e exigem uma modelagem de dados concisa. Portanto, uma boa aplicação a ser utilizada como base para a comparação deve conter elementos como relações entre entidades bem definidas e aninhadas, regras de negócio presentes na camada de dados, restrições que exijam validações de persistência e outros fatores que estimulam a comparação em aspectos variados.

Sistemas de *Enterprise Resource Planning* (ERP) podem conter todos estes elementos. Como o próprio nome diz, esses sistemas normalmente são utilizados por empresas que buscam controle de recursos, auxílio em processos internos, automatização de operações manuais e garantia da guarda de informações. Por se tratar de um tipo de sistema relativamente comum, há bastante conteúdo a seu respeito.

Devido a esse fatores, foi realizado o projeto de um ERP simplificado próximo do mundo real que atua como elemento central para desenvolvimento deste estudo. Sistemas ERP já existentes e maduros no mercado possuem um nível de complexidade alto. Mas diferentemente - neste trabalho - o sistema de informação projetado contém apenas as funcionalidades-núcleo de *softwares* ERP para gestão de recursos. A entidade produtos é a entidade central. As regras de negócio definidas no projeto do sistema são:

- Um produto deve pertencer a somente uma categoria e deve conter uma unidade de medida, sendo ambas (categoria e unidade) parametrizáveis.
- A partir de um produto cadastrado, é possível realizar vendas e compras.
- Vendas e compras de um produto implicam em saída e entrada de estoque, respectivamente.
- Uma venda ou compra deve, obrigatoriamente, conter uma forma de pagamento vinculada e pode ter um vendedor/comprador (é possível realizar vendas sem vendedor).
- Não é possível realizar vendas de produtos sem estoque ou compras de produtos acima do limite máximo definido.
- Uma venda ou compra deve, obrigatoriamente, conter uma forma de pagamento vinculada e pode ter um vendedor/comprador (é possível realizar vendas/compras sem vendedor/comprador).

- Deve ser possível emitir relatórios de vendas.

As funcionalidade do software foram listadas em uma possível ordem de recorrência de utilização:

1. Ponto de Vendas (Frente de caixa);
2. Compras e vendas por busca de produtos;
3. Emissão de relatórios;
4. Recuperação e atualização de estoque;
5. Recuperação de dados em geral;
6. Demais edições e remoções de registros;

Ou seja, presume-se que a funcionalidade de ponto de vendas, para realizar venda de produto somente com cliques, será a mais utilizada, enquanto edições e remoções de registros terão menor frequência de utilização.

5. Decisões de engenharia e desenvolvimento

O desenvolvimento do projeto foi feito utilizando tecnologias de aplicações *web* com arquitetura de três camadas e as soluções utilizadas em cada camada foram definidas criteriosamente. A modelagem dos dados foi feita com o intuito de atender as regras de negócio especificadas e considerando listagem das funcionalidades que serão utilizadas com maior frequência.

5.1. Arquitetura da aplicação

A arquitetura de três camadas foi empregada fazendo o uso das tecnologias conforme o diagrama da Figura 3. O uso de cada uma dessas tecnologias será justificado a seguir.

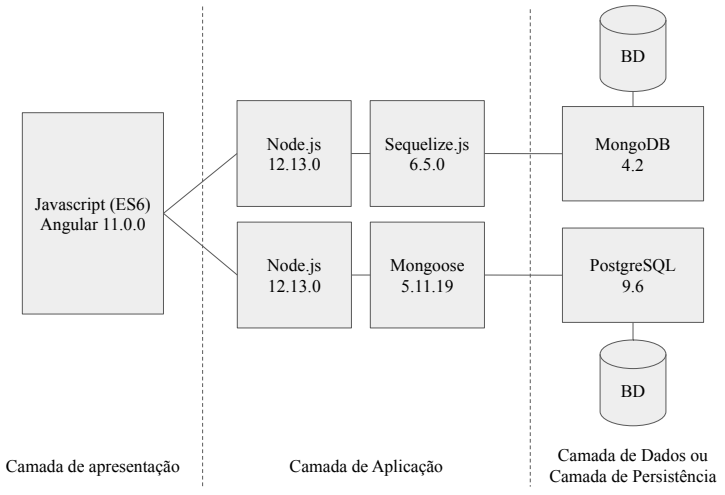

Figura 3. Arquitetura e tecnologias utilizadas no desenvolvimento do sistema

5.1.1. Camada de apresentação

Neste estudo, a camada de apresentação tem um papel coadjuvante, mas não menos importante. Apesar do foco da análise ser direcionado às camadas de persistência, a camada

de apresentação atua como um materializador de resultados. Com a construção de uma interface para visualização dos dados, compreender os acontecimentos se torna uma tarefa mais simples, além de colocar a camada de persistência à prova, de maneira gráfica. Para construção das telas e interfaces, foi utilizado o *framework JavaScript Angular* 11.0.0.

5.1.2. Camada de dados

Na camada de dados, os candidatos escolhidos para representar os modelos SQL e NoSQL foram os SGBDS *PostgreSQL* e o MongoDB, respectivamente.

Durante a escolha do banco de dados relacional, foram levados em consideração fatores como popularidade, custos, comunidade atuante, boa documentação e ser uma solução já consolidada. Muitos bancos relacionais atendem esses fatores, incluindo o PostgreSQL, porém um fator extra de afinidade pessoal com a tecnologia foi um ponto de desempate. A versão utilizada foi a 9.6.

No caso da escolha do banco de dados não-relacional, o cenário muda de figura. Como bancos de dados não-relacionais foram propostos para resolver problemas específicos, nem todas as opções disponíveis são ideais para desenvolvimento software especificamente.

Ser um banco de dados de propósito geral e ter boa popularidade foram alguns dos fatores que tiveram forte influência na escolha do MongoDB, candidato representante do modelo não-relacional.

Além disso, um outro fator teve peso nessa decisão de engenharia. Uma grande peculiaridade dos bancos não-relacionais é que muitos não fazem uso do conceito ACID, o que inclusive impossibilita seus usos em alguns casos exclusivamente por esta característica. Porém, a partir da versão 4, lançada em agosto de 2018, o MongoDB incluiu o suporte a *transactions*, permitindo controle de condições de concorrência e oferecendo uma solução não relacional com grande consistência de dados. Devido a esta funcionalidade, bastante útil para a proposta do projeto, o MongoDB 4.2.0 se tornou a opção mais interessante.

5.1.3. Camada de aplicação

Desde que a mesma linguagem de programação seja usada em ambos os servidores - um conectado ao PostgreSQL e outro ao MongoDB - não há restrições em relação a qual linguagem utilizar, afinal o impacto da linguagem será o mesmo para ambos os lados. Apesar da linguagem de programação não ter impactos na análise final, a escolha do *JavaScript* como linguagem *server-side* é muito pertinente.

Primeiramente, o MongoDB é um banco de dados orientado a documentos. A estrutura de dados utilizada para armazenamento dos dados tem grande similaridade com o *JavaScript Object Notation* (JSON), um formato leve de troca de dados, originário na linguagem *JavaScript*. Além disso, toda parte de interface da aplicação foi desenvolvida utilizando o próprio *JavaScript*, com utilização do *framework Angular*, como já mencionado.

Com o *JavaScript*, portanto, é possível trabalhar com a mesma linguagem em praticamente todas as camadas da aplicação. Para se utilizar o *JavaScript server-side*, entra em cena o Node.js, um ambiente que permite execução do JavaScript no lado de servidores, tecnologia também utilizada no desenvolvimento do projeto.

Com a definição do JavaScript e Node.js como tecnologias atuantes na camada de aplicação, o Sequelize.js - ORM que mapeia consultas SQL para Node.js - e o Mongoose, ODM de MongoDB para Node.js foram os mapeadores escolhidos. Ambas as bibliotecas possuem boa popularidade e esse também foi um fator relevante.

Tanto o Sequelize quanto o Mongoose auxiliam na produtividade de escrita de código. O Sequelize converte a escrita de consultas SQL para algo muito próximo da linguagem JavaScript. No caso do Mongoose, sua principal vantagem é atuar como uma solução baseada em esquemas para modelar os dados da sua aplicação. Possui sistema de conversão de tipos, validação, criação de consultas e *hooks* para lógica de negócios. Todas essas tecnologias desempenham papel fundamental na condução deste trabalho.

5.2. Modelagem

A modelagem dos dados em ambos os bancos teve como base as entidades, regras de negócio e requisitos da aplicação prototipada previamente. Nas modelagens SQL, em geral, principalmente de sistemas com relações entre as entidades bem definidas, não há muito espaço para variações e diferenças entre modelos. Em contrapartida, a modelagem dos dados NoSQL é mais flexível e está relacionada à forma como os dados serão consumidos. A não-estruturação dos dados permite maior diversidade de esquema de dados e há abertura a diferentes possibilidades.

No caso do sistema em questão, os diagramas das modelagens adotadas estão representados nas Figuras 4 e 5. Para o caso do MongoDB, é importante ressaltar que, para cada documento criado, por padrão, é gerado o atributo "_id", que atua como uma chave única em todo o banco de dados para identificação dos documentos. Para fins de legibilidade, este atributo não foi representado.

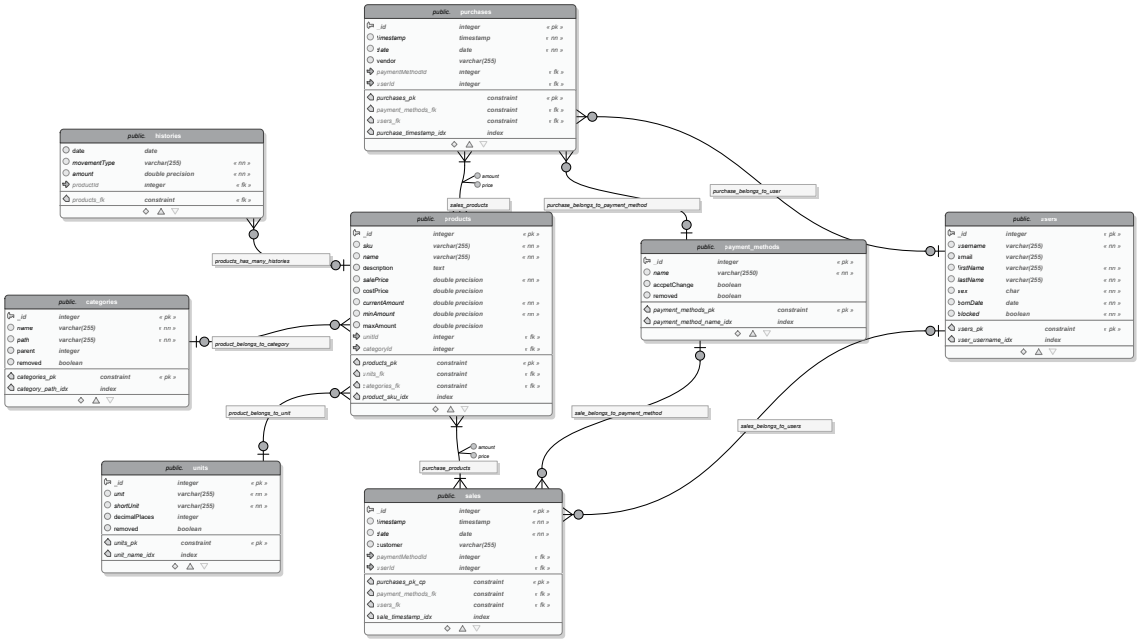

Figura 4. Modelagem relacional dos dados

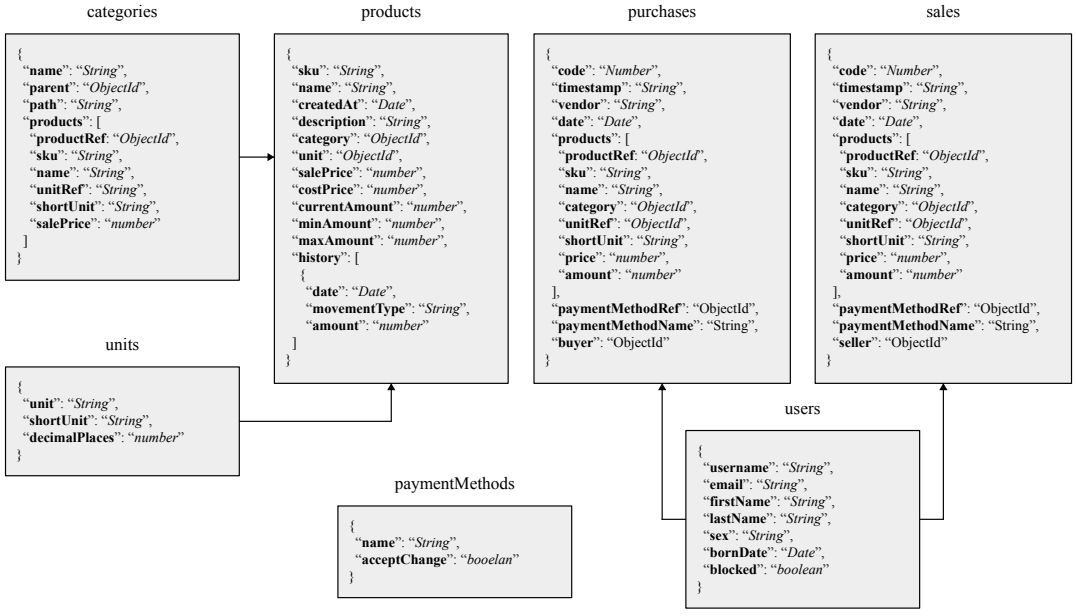

Figura 5. Modelagem não-relacional dos dados

Na modelagem de dados do *PostgreSQL*, os dados foram normalizados e as chaves primárias foram armazenadas nas colunas de nome *"_id"* das respectivas entidades. Os dados foram organizados de forma a não haver duplicidade e inconsistência. Por outro lado, na modelagem do MongoDB, a abordagem utilizada foi bastante distinta do modelo relacional.

No diagrama do MongoDB, campos com tipo de dados *"ObjectId"* referenciam outros documentos. As setas representadas apontam para outras entidades que referenciam a entidade em questão, o que significa que haverá consultas que envolvem mais de

uma *collection*. Para os demais campos do tipo *"ObjectId"* em que as entidades origem não possuem setas apontando para a entidade destino, significa que esse campo é utilizado apenas para referência, não sendo necessárias consultas com aninhamento de documentos.

Os dados da entidade *product* foram replicados nas entidades *sale, purchase* e *category*. Essa decisão é embasada pela proposta do sistema. Como a tela de frente de caixa possivelmente será a mais utilizada, é interessante que haja uma consulta rápida por produtos já agrupados por categorias. Assim, com a modelagem proposta, basta recuperar uma categoria específica e produtos pertencentes a ela já serão retornados na estrutura *products*.

A replicação dos produtos nas vendas e compras segue a mesma lógica. Como vendas e compras têm acesso frequente, quanto menos entidades forem envolvidas na recuperação de uma venda ou compra, é esperado que mais rápida seja a consulta.

Essas decisões foram tomadas para melhorar a performance de consultas comuns. Porem, é sabido que atualizações e remoções de instâncias com dados replicados, como edição de produtos, por exemplo, serão trabalhosas e custosas, já que as inconsistências terão de ser tratadas no nível de aplicação. Todavia, esse tipo de consulta não é frequente quando comparada com as citadas, justificando as decisões de modelagem.

5.3. O software: Monquelize

A aplicação recebeu o nome de Monquelize. As Figuras 6, 7, 8 e 9 apresentam algumas telas do resultado do final desenvolvimento.

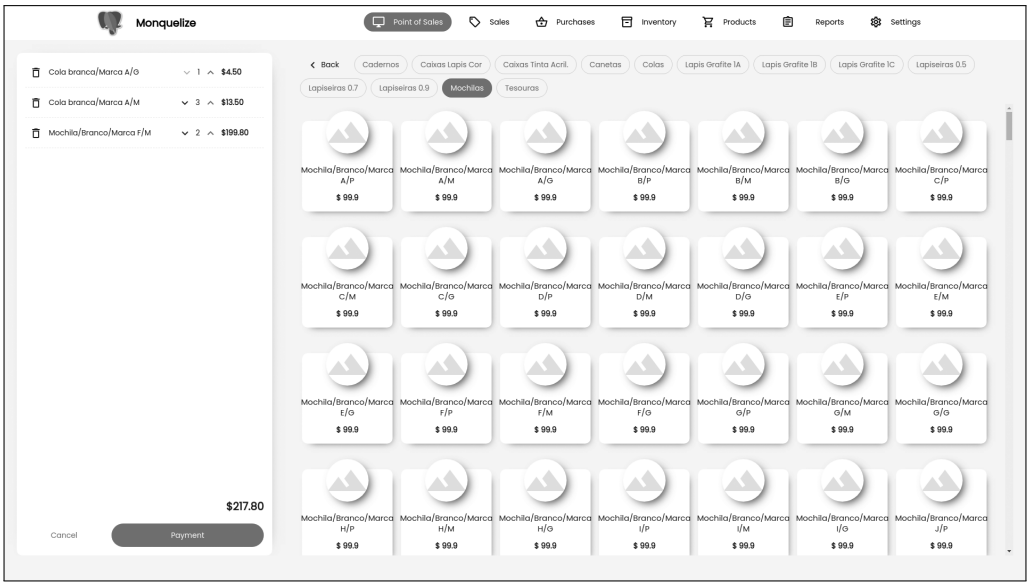

Figura 6. Tela de frente de caixa para venda de produtos

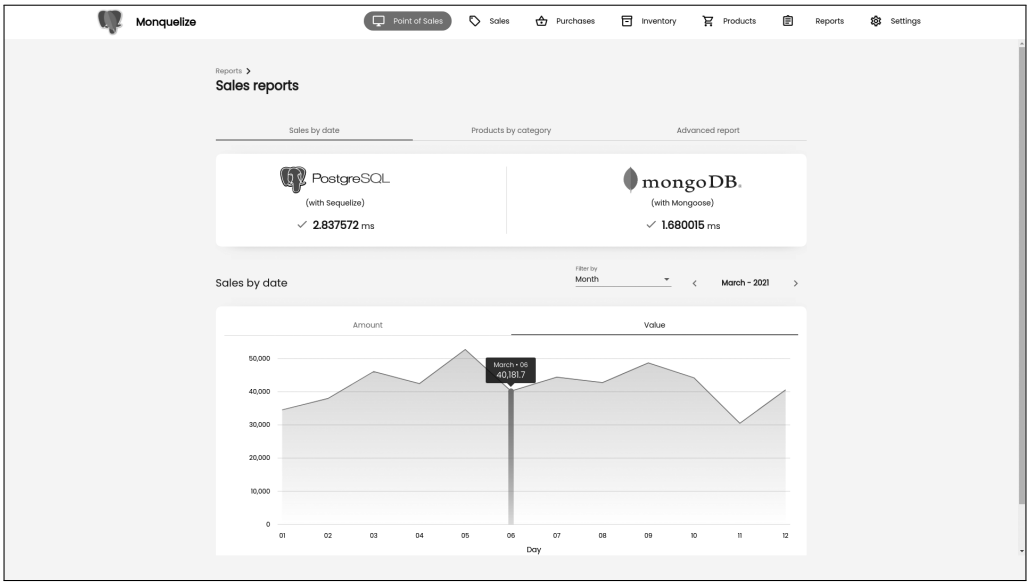

Figura 7. Tela de relatório de vendas

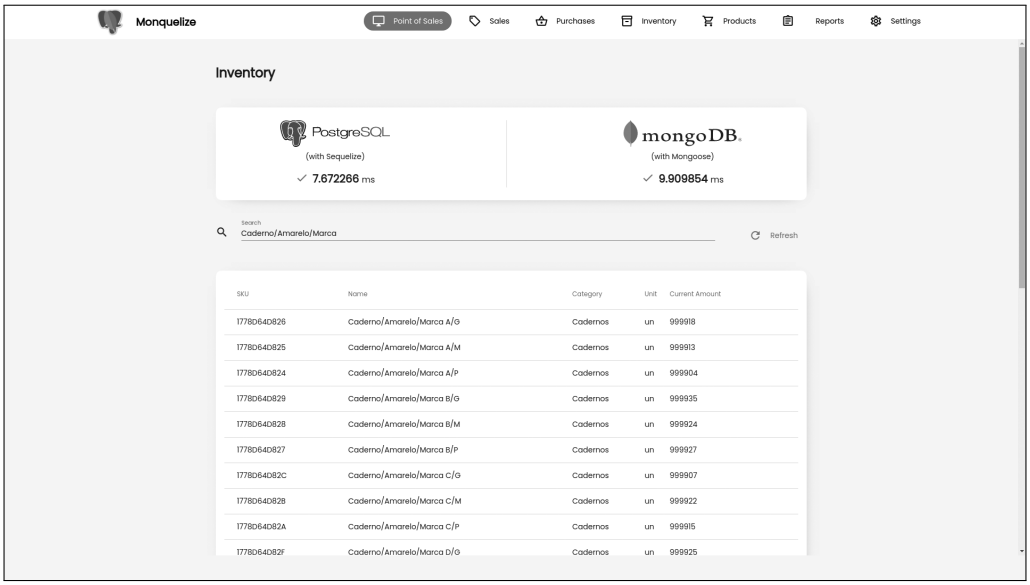

Figura 8. Tela de busca por produtos para consulta de estoque

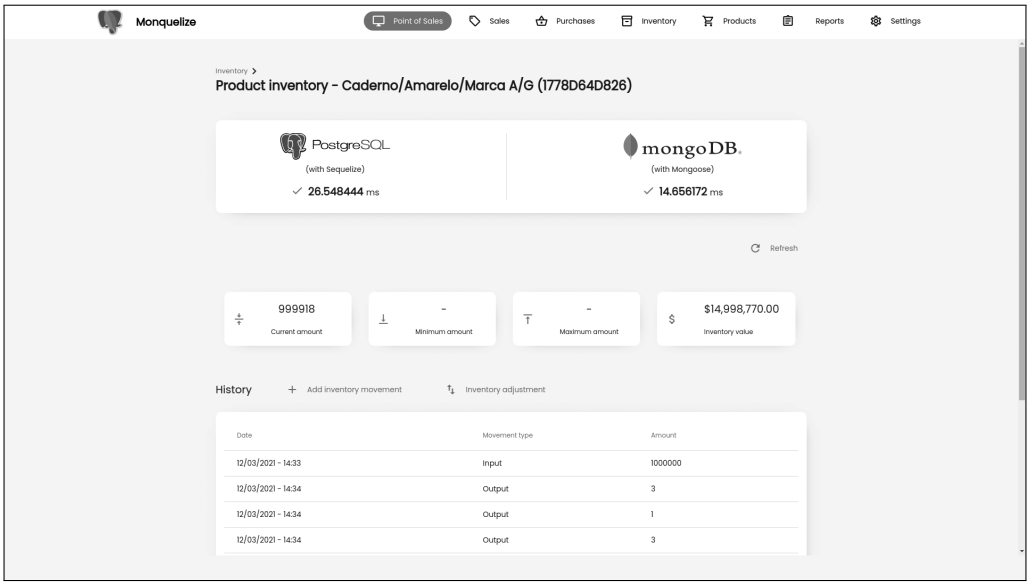

Figura 9. Tela de informações sobre estoque de um produto

A medição dos tempos foi feita considerando apenas o tempo para consulta nos bancos de dados. Não foram levados em consideração fatores como latência ou tempo de renderização. Os tempos de cada servidor são exibidos em tempo real nas telas que enviaram as requisições.

6. Avaliação das diferenças

Após finalização da implementação, foi possível realizar uma série de avaliações e comparações acerca de diversos aspectos.

6.1. Modelagem dos dados

Apesar de ambos os bancos de dados terem a função de entregar os mesmos resultados, quando submetidos às mesmas requisições, a modelagem dos dados foi significativamente distinta.

A modelagem em SQL teve um resultado objetivo e direto. O modelo relacional, quando normalizado corretamente, não permite certa diversidade de modelagens quando comparado com o não-relacional, sendo mais rígido e pouco mutável. Quando o núcleo e principais entidades do sistema foram modelados em relacional, houve poucas alterações subsequentes e o resultado final não exigiu muito esforço.

Porém, essas afirmações não se aplicam a modelagem no MongoDB . A disposição dos dados no MongoDB está fortemente relacionada a como esses dados serão utilizados e a flexibilidade e não-estruturação de dados permite uma gama maior de variedade de modelagens com eficiências equivalentes. Se por um lado, a flexibilidade traz uma maior liberdade, por outro esta liberdade faz com que a modelagem seja menos objetiva, mais abstrata, complexa e propensa a alterações.

Durante a experiência de desenvolvimento do software, foram frequente os casos de mudanças de campos e propriedades das entidades, o que não ocorreu no lado relacional. Essas mudanças em entidades ocasionaram grande mudanças em diversas partes

do sistema, afinal mudanças como esta impactam como a aplicação manipula os dados. Por se tratar de tecnologias mais recentes, modelagens de bancos orientados a documentos, de forma geral, ainda estão em fase de maturação e existem variações e divergências, exigindo experiência para encontrar um modelo ideal.

Em contrapartida, o MongoDB possui menos entidades, já que é possível utilizar a estratégia de documentos aninhados. No caso da entidade produto, o estoque e todo o histórico de movimentação é controlado unicamente pela entidade *product*, enquanto no relacional são necessárias três entidades para realizar este mesmo controle.

6.2. Implementação, complexidade de código e manutenibilidade

Como no lado do MongoDB existem situações de replicação de dados, era esperado que as operações executadas tivessem maior dificuldade de implementação em comparação com o PostgreSQL. Apesar de esse ser um resultado esperado, não havia como mensurar o quão mais complexo seria, antes do desenvolvimento.

Após o desenvolvimento do sistema, foi possível mensurar visualmente as diferenças práticas de implementação. As instruções 3 e 4 são comparativos das mesmas atividades sendo executadas em ambos os casos. Os dois se referem a funcionalidade de edição de um produto, como alteração de nome, descrição, categoria, preço de venda, dentre outros parâmetros.

Instrução 3. Edição de produto utilizando Sequelize/PostgreSQL

```
1  const { Product } = require('../../models');
2
3  module.exports = async function editProduct(productId, product) {
4    let [, updatedProduct] = await Product.update(product, {
5      where: { _id: productId },
6      returning: true,
7      plain: true,
8    });
9
10   updatedProduct = updatedProduct.dataValues;
11
12   return updatedProduct;
13 };
```

Instrução 4. Edição de produto utilizando Mongoose/MongoDB

```
1  const { productModel } = require('../../models');
2  const { editProduct: editProductInSales } = require('../sale');
3  const { editProduct: editProductInPurchases } = require('../purchase');
4  const {
5    editProduct: editProductInCategory,
6    addProduct: addProductInCategory,
7    removeProduct: removeProductInCategory,
8  } = require('../category');
9
10 module.exports = async function editProduct(productId, productData, session) {
11   const productDoc = await productModel.retrieve(productId, session);
12
13   const productObj = productDoc.toObject();
14
15   const updatedProductDoc = await productDoc.edit(productData);
16
17   await updatedProductDoc
```

```
18      .populate([
19        { path: 'category', select: 'name' },
20        { path: 'unit', select: ['unit', 'shortUnit'] },
21      ])
22      .execPopulate();
23
24    if (hasToUpdateInSalesOrPurchases(productObj, updatedProductDoc)) {
25      await editProductInPurchases(updatedProductDoc._id, updatedProductDoc, session);
26      await editProductInSales(updatedProductDoc._id, updatedProductDoc, session);
27    }
28
29    await updateProductInCategory(productObj, updatedProductDoc, session);
30
31    return updatedProductDoc;
32  };
33
34  function hasToUpdateInSalesOrPurchases(oldProduct, newProduct) {
35    const relevantFields = ['sku', 'name', 'category', 'unit'];
36
37    return relevantFields.some((field) => oldProduct[field] !== newProduct[field]);
38  }
39
40  async function updateProductInCategory(oldProduct, newProduct, session) {
41    if (oldProduct.category.equals(newProduct.category._id)) {
42      await editProductInCategory(oldProduct.category, newProduct, session);
43    } else {
44      await addProductInCategory(newProduct.category._id, newProduct, session);
45      await removeProductInCategory(oldProduct.category, oldProduct._id, session);
46    }
47  }
```

É importante ressaltar que a mesma formatação de código foi utilizada para ambos os casos, fazendo o uso das bibliotecas ESLint e Prettier, que auxiliam tanto na escrita de códigos utilizando boas práticas de programação quanto na formatação padronizada e automatizada. Ou seja, não há interferência da formatação nas comparações.

O exemplo de atualização de produtos foi utilizado para demonstrar a diferença entre as duas camadas de persistência e apresentar uma situação recorrente em outras partes do código.

No caso do Sequelize/*PostgreSQL*, como as entidades estão interligadas por relações através de chaves, apenas a atualização na instância de produto é suficiente para que a mudança seja refletida em outras partes do sistema. Já no caso do *Mongoose/MongoDB*, características do produto estão replicadas nas entidades de venda, compra e categorias, como demonstrado anteriormente na modelagem. Portanto, além de ser necessário atualizar os valores no documento da própria entidade produto, essa mesma alteração tem de ser repetida para as demais entidades. É importante ressaltar que nas regras de negócio da aplicação foi definido que alterações em produtos impactam em compras e vendas já realizadas. Considerando todos esses fatores, atualização de produto é uma tarefa custosa tanto em nível de processamento quanto de produtividade, no caso do NoSQL.

Deve-se deixar claro que no código da instrução 4 há várias chamadas de funções para realizar alterações nas demais entidades. Se ao invés de funções, as atualizações ocorressem em um único método, a quantidade de código na instrução seria aumentada significativamente.

Como já mencionado, esse tipo de situação foi recorrente durante o desenvolvimento do projeto. As inconsistências provenientes da replicação de dados com o obje-

tivo de melhorar o desempenho precisaram ser tratadas manualmente no ambiente não-relacional, fazendo com que o código possuísse maior complexidade e maior suscetibilidade a erros.

Além de ter processos equivalentes ao *Sequelize/PostgreSQL* consideravelmente mais complexos de serem implementados, o *Mongoose/MongoDB* necessitou de mais serviços/métodos exportados para realizar as operações. Cada serviço contém as regras de negócio necessárias para realizar as alterações na camada de persistência. A Figura 10 representa a estrutura de arquivos utilizada para exportar serviços da entidade venda, como consulta de vendas, inserção, remoção, etc.

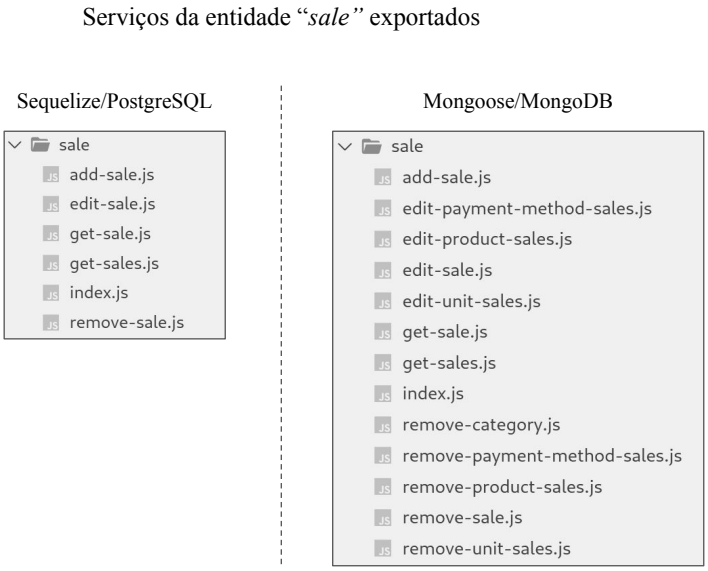

Figura 10. Serviços da entidade *sale* exportados

É nítido que a camada de persistência com o Mongoose/MongoDB possui mais arquivos. Isso, novamente, reflete em mais código escrito, maior probabilidade de ocorrência de novos erros e maior dificuldade de manutenção.

Optou-se por utilizar métricas qualitativas de engenharia de software para realizar as comparações, pois elas, por si só, já evidenciam as diferenças pretendidas neste estudo.

6.3. Consistência

Por se tratar de uma aplicação com grande necessidade de consistência de dados, é necessário que haja garantia desse princípio. Com isso, as inconsistências de dados no modelo não-relacional precisaram ser tratadas a nível de aplicação, uma vez que dados replicados não possuíam relação direta entre si.

Como mencionado anteriormente, a versão do MongoDB utilizada possui o suporte a *multi-document transactions* para situações que requerem atomicidade de leituras e escritas a múltiplos documentos. Essa funcionalidade do SGBD, já consolidada no ambiente SQL, foi amplamente utilizada no desenvolvimento do projeto. Por outro lado, as *transactions* foram utilizadas em raros casos no *PostgreSQL*.

O que pode ser analisado, a partir desta observação, é que, por mais que haja suporte a operações atômicas em ambos os casos, fazer com que consistências dependam de intervenção manual para serem garantidas é algo que pode comprometer a confiabilidade do sistema. A desnormalização de dados implica em necessidade de escrita de código, com maiores chances a introdução de *bugs*, para que a consistência seja mantida, o que já ocorre naturalmente na abordagem relacional devido a própria rigidez do modelo.

O MongoDB faz amplo uso desse tipo de intervenção manual, o que indica ser um sistema mais suscetível a se encontrar falhas. Se não houverem ferramentas extras que auxiliem no controle dessas inconsistências a nível de desenvolvimento, como técnicas de teste de software (custosas, por sinal), manter os dados concisos após séries de manutenções no código pode se tornar uma tarefa complexa.

6.4. Performance

A modelagem do MongoDB foi projetada para se obter alto desempenho em operações específicas da aplicação. Em outras palavras, para telas e operações que têm maior frequência de uso, evitou-se o uso de referências com o intuito de entregar a resposta desejada com consulta a somente uma entidade. No caso do *PostgreSQL*, os dados foram normalizados, portanto, mais entidades foram consultadas.

Para realizar as medições de desempenho, ambos os bancos de dados foram populados com a mesma carga de dados. O volume de dados foi definido com base na usabilidade e proposta do sistema. Não faz sentido inserir uma carga muito alta de dados, incondizente com os fins do sistema, nem uma carga pequena demais tal que não seja possível visualizar diferenças. De acordo com estes critérios, os dados foram persistidos de acordo com a seguinte distribuição:

- 12 usuários, que atuam no sistema como vendedores e compradores;
- 14 categorias de produtos;
- 2 unidades de medida;
- 6 formas de pagamento;
- 3.189 produtos, pertencentes as categorias cadastradas;
- 345.983 vendas, distribuídas ao longo de 12 meses (média de 100 vendas diárias);

Para realizar a análise e diferenciação de performance, foram elencadas as principais consultas realizadas no sistema, além de consultas com expectativa por grande diferença de tempo. São elas:

1. Consulta de produtos por categoria, para realização de vendas na frente de caixa
2. Consulta de vendas por mês
3. Consulta de produtos, utilizando paginação *server-side*
4. Inserção de novas vendas
5. Edição de produtos
6. Remoção de produto
7. Relatório de vendas por mês, retornando somatório do total vendido e quantidade de vendas
8. Relatório de vendas por mês distribuídas por categorias de produtos, retornando total vendido e quantidade de vendas por categoria

As configurações do hardware utilizado para realização dos testes estão descritos na Tabela 3.

Tabela 3. Hardware utilizado na realização dos testes de desempenho

Elemento	Especificação
Sistema operacional	Linux - Ubuntu 20.04 (kernel 5.4)
Processador	Intel Core i5-8265U (8ª geração) 1.6 GHz até 3.9 GHz, 6MB Cache
Memória	Dell 8GB DDR4 2666MHz
Armazenamento	SSD Crucial 500GB SATA III 6Gb/s - Leituras 560MB/s e Gravações 510MB/s

Para medição dos tempos, para cada uma das oito consultas listadas, foram feitas dez medições de tempo e calculada a média. Os resultados obtidos estão representados na Tabela 4 e Figura 11.

Tabela 4. Tempos de execução para realização de consultas

Consulta	Sequelize & PostgreSQL Tempo de consulta (ms)	Mongoose & MongoDB - Tempo de consulta (ms)	Mais performático	Percentual relativo ao maior tempo
#1	119,14	17,00	MongoDB	14,26 %
#2	54,69	852,34	PostgreSQL	6,42 %
#3	59,52	18,37	MongoDB	30,86 %
#4	77,60	66,01	MongoDB	85,06 %
#5	33,85	181,04	PostgreSQL	18,70 %
#6	1,51	719,79	PostgreSQL	0,21 %
#7	64,70	54,69	MongoDB	84,53 %
#8	62,84	57,32	MongoDB	91,22 %

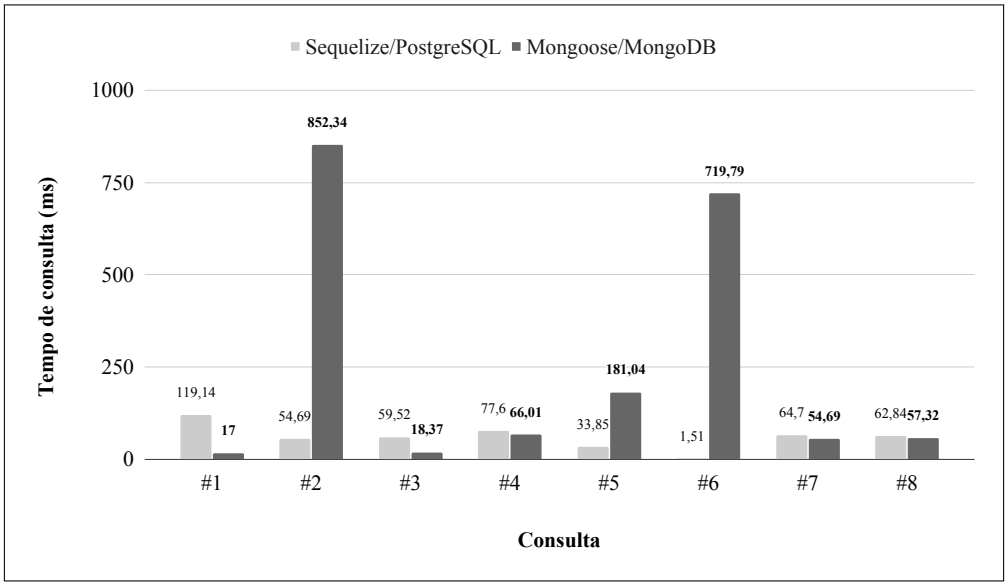

Figura 11. Arquitetura de uma aplicação *web* em 3 camadas

Observou-se que, para execução da consulta #1 do Monquelize, o ecossistema MongoDB foi obteve um desempenho no tempo de resposta de cerca de 7 vezes maior quando comparado com o *PostgreSQL*. O mesmo ocorreu a consulta de produtos com paginação (#3), em que o MongoDB realizou a consulta em aproximadamente 31% do tempo do PostgreSQL. Essas duas consultas tiveram diferenças de tempo significativas em relação ao ecossistema *PostgreSQL*. As demais consultas em que o MongoDB obteve vantagem tiveram diferenças de tempo pouco notáveis para o volume de dados utilizados.

Para as outras consultas nas quais tiveram vantagem no ecossistema *PostgreSQL* (#2, #5 e #6), a diferença de tempos foi consideravelmente alta, chegando a até 0,21% do tempo relativo ao MongoDB. Para as consultas #5 e #6, de edição e remoção, respectivamente, era esperado que o MongoDB apresentasse maior tempo de consultas, pois são necessárias modificações em várias instâncias.

A consulta #2, de recuperação das vendas mensais, teve desempenho muito abaixo do esperado no MongoDB. Deve ser destacado o fato dessa consulta ter uma espécie de *JOIN* do SQL na *collection* de usuários, já que são referenciados nas vendas. A explicação para o desempenho ruim está relacionada ao Mongoose, que foi utilizado para manipular a segunda consulta responsável por inserir as características do vendedor (usuário) em cada venda retornada, através do método *populate*. Ou seja, apesar do Mongoose tornar a consulta mais simples, houve grande perda de performance.

A queda de performance do Mongoose pôde ser verificada de outra maneira. Após a realização de uma consulta não listada, de recuperação das vendas sem necessidade de saber seus vendedores, o tempo do *Mongoose/MongoDB* foi bastante similar ao do *Sequelize/PostgreSQL*, provando que o método *populate* foi o grande responsável pela queda de performance.

Em resumo, no quesito desempenho, o MongoDB foi pouco melhor quando era esperado que realmente o fosse e muito pior quando também era previsto tal desempenho. Em outras palavras, houve pouco ganho nas vantagens e altas perdas nas desvantagens para o *Mongoose/MongoDB*.

6.5. Usabilidade

Por mais que os tempos relativos às performance de consultas sejam bastante distintos, alguns sendo até 7 vezes mais rápidos, o usuário final só irá notar essa diferença quando as consultas exigirem um volume de dados retornado maior.

Vale destacar que a consulta que mais demandou tempo nos testes realizados obteve tempo total de 1.210,9283 *ms*, para recuperar vendas mensais com vendedores no MongoDB. A diferença para o *PostgreSQL* foi na casa dos milissegundos.

Se as diferenças no nível de performance são notáveis, as relativas à experiência do usuário final não seguem a mesma linha. A experiência de uso do usuário final da aplicação praticamente não sofreu alterações em qualquer uma das opções.

7. Conclusão

A partir da finalização do projeto, foi possível perceber e mensurar as diferenças entre as camadas de persistência com abordagens relacional e não relacional. Em relação a desenvolvimento, complexidade de código, produtividade e manutenibilidade, o ambiente

não relacional do MongoDB foi nitidamente mais complexo e moroso. Se for comparado o esforço e custo de implementação *versus* o ganho de performance do MongoDB, os resultados não foram satisfatórios. O trabalho e esforço para fazer um mesmo sistema ERP simplificado foi substancialmente maior em NoSQL e os ganhos de performance não justificam a escolha desta tecnologia para composição do sistema proposto.

A contribuição do presente estudo está ligada, principalmente, à área de Engenharia de software, auxiliando e servindo como base de consulta para tomadas de decisão em projetos de arquiteturas de software. Não há problema em uma determinada solução ser mais complexa ou menos performática que outra, por exemplo, desde que os responsáveis por tomar a decisão estejam cientes das consequências e se estão alinhadas com o objetivo do projeto. As soluções podem ser utilizadas em conjunto para extrair o melhor dos dois mundos.

O software projetado para realização das análises não possui alta complexidade de relação entre as entidades. Logo, um incremento de entidades e mais regras de negócio no sistema proposto, que exijam consultas mais elaboradas em ambos os lados (SQL e NoSQL) e permitam maior diversidade de comparações, é algo interessante de ser executado em trabalhos posteriores para extrair análises mais robustas. Além disso, comparações em sistemas com grandes volumes de dados (*big data*) permitirão evidenciar ainda mais as diferenças de desempenho.

Referências

Ali, W., Shafique, M. U., Majeed, M. A., and Raza, A. (2019). Comparison between sql and nosql databases and their relationship with big data analytics. *Future Internet*, 4(2):1–10.

Anderson, B. and Nicholson, B. (2020). Sql vs. nosql databases: What's the difference?

Date, C. J. (2003). *An Introduction to Database Systems*. Pearson, 8 edition.

Diogo, M., Cabral, B., and Bernardino, J. (2019). Consistency models of nosql databases. *Future Internet*, 11(2).

Elmasri, R. and Navathe, S. B. (2011). *Sistemas de Bancos de Dados*. Addison-Wesley, 7 edition.

Fonseca, E. (2019). O que é orm?

Gupta, A., Tyagi, S., Panwar, N., Sachdeva, S., and Saxena, U. (2017). Nosql databases: Critical analysis and comparison. In *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*, pages 293–299.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., and Gosain, D. (2012). A survey and comparison of relational and non-relational database. *International Journal of Engineering Research Technology (IJERT)*, 6:6.

Kumar, K. and Azad, S. K. (2017). Database normalization design pattern. In *2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON)*, pages 318–322.

Laksono, D. (2018). Testing spatial data deliverance in sql and nosql database using nodejs fullstack web app.

Lemos, M. F., de Oliveira Leandro César Ruela, P. C., da Silva Santos, M., and Silveira, T. C. (2013). Aplicabilidade da arquitetura mvc em uma aplicação web (webapps). *Revista Eletrônica Científica de Ciência da Computação*, 8(1).

Meirelles, P. R. M. (2013). Monitoramento de métricas de código-fonte em projetos de software livre.

Wu, X. (2019). Metadata-based im a-based image collec ge collecting and d ting and databasing for abasing for sharing and analysis.

Yishan, L. and Manoharan, S. (2017). A performance comparison of sql and nosql databases.

MINISTÉRIO DA EDUCAÇÃO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO - NG

ATA Nº 23 / 2021 - DECOM (11.56.03)

Nº do Protocolo: 23062.013062/2021-95

Belo Horizonte-MG, 25 de março de 2021.

Às **15** horas do dia **25** do mês de **Março** de **2021**, realizou-se a sessão pública de defesa do Trabalho de Conclusão de Curso do(a) aluno **Pedro Luiz Ligeiro Matias**, intitulado **Comparativo de Desenvolvimento e Desempenho entre MongoDB e PostgreSQL em Aplicações com Mapeamento de Objetos**, por videoconferência, na plataforma **Google Meet (meet.google.com/sbo-muyr-zog)**.

Abrindo a sessão, o Presidente da Banca Examinadora de Trabalho de Conclusão de Curso, professor **Evandrino Gomes Barros** (orientador), constituída também pelos professores **Cristiano Amaral Maffort** (coorientador), **Luciana Maria de Assis Campos (DECOM)** e **Edson Marchetti da Silva (DECOM)**, informou aos presentes as regras de condução das atividades da defesa do TCC e passou a palavra ao candidato para apresentação do trabalho. Seguiu-se a arguição pelos examinadores, com a respectiva defesa do candidato. Logo após, a banca se reuniu, sem a presença do candidato e do público, para julgamento e expedição do resultado da avaliação, que corresponde a parte da Nota Final da disciplina de TCC, transcrita abaixo:

Item de Avaliação	Nota Orientador(a)/Coorientador(a)	Nota 1 Banca	Nota 2 Banca	Média das Notas
Desenvolvimento do trabalho durante a disciplina: **30 pontos**.	30 pontos	-------	-------	-------
Monografia: formatação, redação e conteúdo do trabalho: **40 pontos**.	32 pontos	32 pontos	35 pontos	33 pontos
Apresentação oral: domínio do tema, clareza, tempo: **30 pontos.**	30 pontos	30 pontos	30 pontos	30 pontos

Observações e orientações finais da banca: a banca delibou sobre as alterações a serem feitas no texto final, sendo que cada membro enviou suas correções diretamente ao aluno. O aluno fará as alterações no prazo

estipulado pela Orientadora da Disciplina de TCC da Eng. Comp., Profa. Natalia Batista, com aprovação do professor orientador.

O resultado da avaliação foi comunicado publicamente ao aluno pela banca, que recebeu as orientações finais. Nada mais havendo a tratar, o Presidente encerrou a sessão e lavrou a presente ATA que será assinada pelos membros participantes da Banca Examinadora.

(Assinado digitalmente em 25/03/2021 21:38)
CRISTIANO AMARAL MAFFORT
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOM (11.56.03)
Matrícula: 1576838

(Assinado digitalmente em 26/03/2021 08:38)
EDSON MARCHETTI DA SILVA
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOM (11.56.03)
Matrícula: 1509224

(Assinado digitalmente em 25/03/2021 17:14)
EVANDRINO GOMES BARROS
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOM (11.56.03)
Matrícula: 2606264

(Assinado digitalmente em 01/04/2021 21:28)
LUCIANA MARIA DE ASSIS CAMPOS
PROFESSOR ENS BASICO TECN TECNOLOGICO
DECOM (11.56.03)
Matrícula: 2557171

Para verificar a autenticidade deste documento entre em https://sig.cefetmg.br/public/documentos/index.jsp informando seu número: **23**, ano: **2021**, tipo: **ATA**, data de emissão: **25/03/2021** e o código de verificação: **d90943b3c9**