

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus de Freitas Andrade

**Avaliação do desempenho de sistemas armazém
de dados baseados em NoSQL.**

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus de Freitas Andrade

**Avaliação do desempenho de sistemas armazém de dados
baseados em NoSQL.**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, como parte dos requi-
sitos exigidos para a obtenção título de Ba-
charel em Sistemas de Informação.

Orientador: Maria Camila Nardini Barioni

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2024

Agradecimentos

Gostaria de expressar meus sinceros agradecimentos aos meus pais, ao meu irmão, aos meus tios e aos meus avós, que nunca mediram esforços para me apoiar em todos os momentos da minha vida. O incentivo de vocês foi fundamental para a minha trajetória. É graças a vocês que pude realizar meus sonhos.

Agradeço também à minha orientadora, Maria Camila Nardini Barioni, pelo apoio, paciência e orientação durante todo o desenvolvimento deste trabalho.

Lista de ilustrações

Figura 1 – Ranking de todos os SGBD's Fonte: (DB-Engines, 2024).	10
Figura 2 – Ranking de todos os SGBD's baseados em documentos Fonte: (DB-Engines, 2024).	11
Figura 3 – Exemplo de dados estruturados. Fonte: (BARROSO, 2023).	14
Figura 4 – Exemplo de dados semiestruturados (MELLO et al., 2000)	15
Figura 5 – Exemplo dados não estruturados (BERRUTTI, 2019).	16
Figura 6 – Modelo descrito por Barry Devlin e Paul Murphy, imagem traduzida de (DEVLIN; MURPHY, 1988).	17
Figura 7 – Exemplo de modelagem dimensional, imagem traduzida de (KIMBALL; ROSS, 2013).	18
Figura 8 – Modelo Plano, imagem adaptada de (CHEVALIER et al., 2016).	21
Figura 9 – Modelo Deco, imagem adaptada de (CHEVALIER et al., 2016).	21
Figura 10 – Modelo Fragmentado, imagem adaptada de (CHEVALIER et al., 2016).	22
Figura 11 – Modelo Híbrido, imagem adaptada de (CHEVALIER et al., 2016).	23
Figura 12 – Exemplo de coleções, imagem traduzida de (MONGODB, s.d.).	24
Figura 13 – Organização da estrutura do MongoDB, imagem traduzida de (studyo-pedia, s.d.).	24
Figura 14 – Exemplo de SQL.	25
Figura 15 – Exemplo de MQL.	25
Figura 16 – Exemplo de schema no modelo baseado em documentos	26
Figura 17 – schema dinâmico no modelo baseado em documentos	27
Figura 18 – Exemplo de criação da tabela usuário	27
Figura 19 – Exemplo de criação da tabela telefones	28
Figura 20 – Exemplo de criação da tabela endereços	28
Figura 21 – Diagrama do método de trabalho.	31
Figura 22 – Tela inicial MongoDBCompass.	33
Figura 23 – Modelo dos dados definidos para o projeto.	34
Figura 24 – Código para ingestão dos dados.	36
Figura 25 – Classe para interação com o MongoDB.	37
Figura 26 – Informação geral sobre a carga dos dados.	38
Figura 27 – Coleções criadas após a carga dos dados.	39
Figura 28 – Exemplo dos dados nas coleções.	39
Figura 29 – Consulta para identificar os produtos com maiores valores de venda de um vendedor.	40
Figura 30 – Consulta para identificar o cliente com menor valor em compras.	41

Figura 31 – Consulta para analisar a quantidade de cada produto vendido e o preço total.	41
Figura 32 – Consulta para contar o número de clientes atendidos por cada vendedor.	42
Figura 33 – Diferente maneira de se consultar quantos clientes cada vendedor possui.	42
Figura 34 – Consulta para calcular o gasto médio de todos os clientes.	43
Figura 35 – Código utilizado para estabelecer a conexão com o MongoDB.	43
Figura 36 – Código utilizado para listar as coleções no banco de dados.	44
Figura 37 – Código utilizado para armazenar todas as consultas de interesse.	44
Figura 38 – Método utilizado para executar as consultas e registrar o tempo de execução.	44
Figura 39 – Trecho de código para executar todas as consultas	45

Lista de tabelas

Tabela 1 – Produtos com os maiores valores de venda de um vendedor. Consulta definida em 29.	46
Tabela 2 – Consulta para identificar o cliente com menor valor em compras. Consulta definida em 30	47
Tabela 3 – Consulta para analisar a quantidade de cada produto vendido e o preço total. Consulta definida em 31	47
Tabela 4 – Consulta para contar o número de clientes atendidos por cada vendedor. . Consulta definida em 32	47
Tabela 5 – Segunda forma de se consultar quantos clientes cada vendedor possui. Consulta definida em 33	48
Tabela 6 – Consulta para calcular o gasto médio de todos os clientes. Consulta definida em 34	48

Lista de abreviaturas e siglas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
IOT	Internet das coisas (Internet of Things)
JSON	JavaScript Object Notation
NoSQL	Not Only SQL
SGBD	Sistema Gerenciador de Banco de Dados
SQL	Linguagem de consulta estruturada (Structured Query Language)
XML	Linguagem de Marcação Extensível(eXtensible Markup Language)

Sumário

1	INTRODUÇÃO	9
1.1	Objetivos	11
1.2	Justificativa	11
1.3	Organização do texto	12
2	REFERENCIAL TEÓRICO	13
2.1	Dados	13
2.1.1	Estruturados	13
2.1.2	Semiestruturados	14
2.1.3	Não estruturados	15
2.2	Armazém de dados	16
2.2.1	Armazém de Dados Baseado em SQL	18
2.2.1.1	Dimensional	18
2.2.2	Armazém de Dados Baseado em NoSQL	19
2.2.2.1	Documentos	19
2.3	MongoDB	23
2.4	Necessidade de uma Abordagem NoSQL	25
2.4.1	Situações em que NoSQL é mais apropriado	25
2.4.2	Comparação com o modelo SQL	26
3	TRABALHOS CORRELATOS	29
4	MÉTODO DE TRABALHO	31
4.1	Definição do SGBD	31
4.2	Configuração do Ambiente de Teste	32
4.3	Definição dos Dados	33
4.4	Desenvolvimento do Processo de Carga dos Dados	35
4.5	Carga dos Dados	37
4.6	Definição das Consultas	40
4.7	Execução das Consultas	43
5	ANÁLISE E DISCUSSÃO DOS RESULTADOS OBTIDOS	46
6	CONCLUSÕES E RECOMENDAÇÕES PARA TRABALHOS FUTUROS	49
	REFERÊNCIAS	50

A	APÊNDICES	53
----------	----------------------------	-----------

1 Introdução

Conforme os sistemas empresariais foram se desenvolvendo e as empresas se tornando cada vez mais tecnológicas, foi inevitável a crescente dos dados em todos os setores. Neste contexto de desenvolvimento e aumento na geração de dados, tornou-se cada vez mais evidente a necessidade de realizar o armazenamento e análise destas informações de maneira eficiente, buscando redução de custos, menor redundância e maior disponibilidade.

Nos primórdios dos bancos de dados os grandes problemas enfrentados pelas companhias eram bastante relacionados com o fato de que os conjuntos de dados eram armazenadas de forma bruta, sem transformações, sem capacidade de relacionamento e com dificuldade para analisar dados históricos. A partir destes problemas a arquitetura de armazém de dados foi desenvolvida na década de 80 para auxiliar nas tomadas de decisões dos negócios ([FOOTE, 2023](#)).

Como descrito por ([KIMBALL; ROSS, 2013](#)), o armazém de dados tem como objetivo não apenas selecionar as melhores tecnologias e práticas para o desenvolvimento, mas também atender as necessidades do negócio e criar uma solução que consiga de forma efetiva entregar aos usuários os resultados esperados com eficiência.

Com o avanço tecnológico cada vez mais acelerado e a proliferação de diversas novas fontes de dados, o cenário evoluiu para além dos dados estruturados que tradicionalmente dominavam os ambientes de armazém de dados. Hoje, há uma vasta quantidade de dados semiestruturados e dados não estruturados provenientes de jogos, redes sociais, veículos, dispositivos IOT, entre outras fontes. Assim como na década de 80, as empresas rapidamente perceberam a necessidade de obter o poder de analisar e entender todas estas novas informações. Isto impulsionou o desenvolvimento de tecnologias e arquiteturas capazes de lidar com estes dados de maneira eficiente ([YANGUI; NABLI; GARGOURI, 2017](#)).

Neste sentido, o modelo NoSQL surgiu como uma alternativa aos modelos tradicionais, oferecendo vantagens e desvantagens para o gerenciamento de grandes volumes de dados, sejam eles estruturados, semiestruturados ou não estruturados. Bancos de dados NoSQL são conhecidos por sua alta velocidade de leitura e gravação, além de sua flexibilidade na implementação. Características estas que são bastante procuradas por empresas e desenvolvedores que querem construir soluções fáceis de serem implementadas e com alta performance. Os modelos de bancos de dados NoSQL podem ser divididas em cinco principais: bancos de dados orientados a documentos, orientados a colunas, chave-valor, orientados a grafos e híbridos ([FOWLER, 2015](#)).

As arquiteturas clássicas de armazéns de dados têm como principal objetivo facilitar o acesso aos dados, tornando-os fáceis de entender não apenas para os desenvolvedores,

mas também para o negócio. Além disto, devem ser capazes de oferecer boa performance para suportar os diferentes tipos de agregações que o negócio pode desejar realizar sobre os dados (KIMBALL; ROSS, 2013). Sendo assim, é do interesse das empresas que uma nova proposta de solução, que ofereça flexibilidade e a capacidade de processar grandes volumes e dados complexos, atenda aos requisitos do negócio. De forma escalável, esta solução deve ser capaz de entregar informações valiosas que impulsionem o crescimento e atendam às demandas modernas dos clientes (NALLA; REDDY, 2024).

Com a existência de diversas formas de organizar um armazém de dados, seja baseado em SQL, utilizando diferentes formas de modelagem, como a relacional ou a dimensional, ou baseado em NoSQL, como o modelo orientado a documentos, é essencial entender o básico dessas opções para avaliar a necessidade de adotar novas abordagens. Este trabalho apresenta, de forma resumida, o modelo dimensional baseado em SQL e aprofunda-se no modelo orientado a documentos, que foi escolhido para a realização dos testes e das conclusões deste projeto. Esse modelo foi selecionado devido à sua popularidade e à ampla disponibilidade de material de apoio, como demonstrado na Figura 1. Nessa classificação, o MongoDB, um sistema orientado a documentos, ocupa a quinta posição até outubro de 2024, posicionando-se atrás apenas dos SGBDs baseados em SQL e liderando entre os SGBDs orientados a documentos, conforme demonstrado na Figura 2.

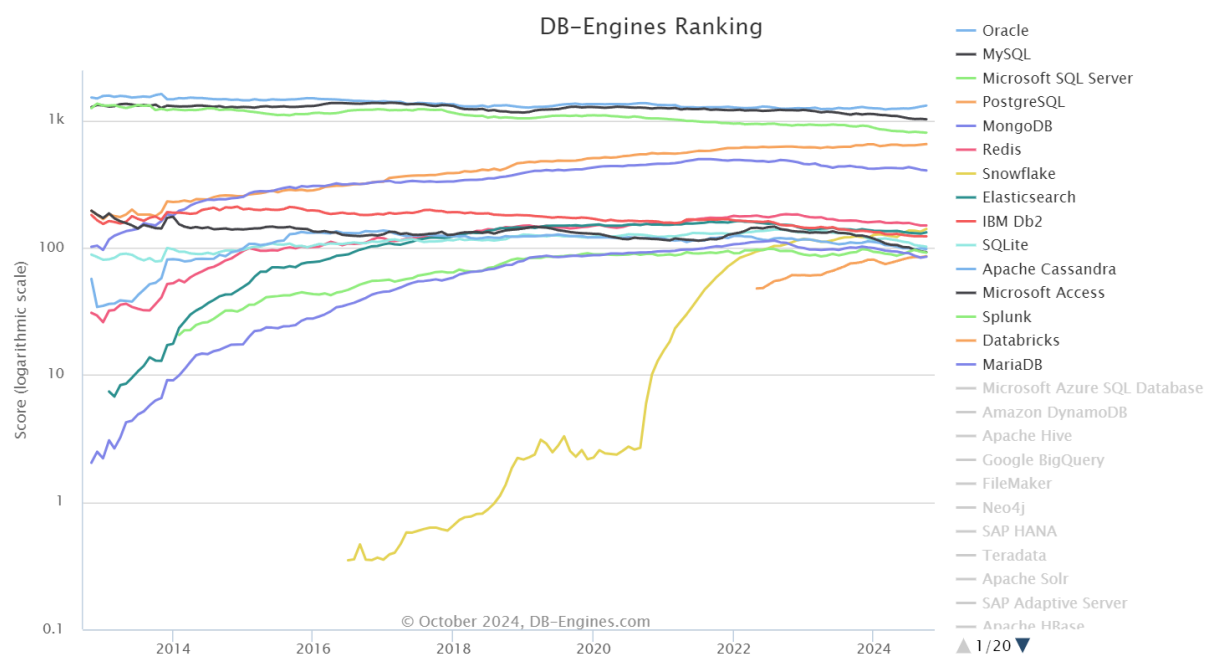


Figura 1 – Ranking de todos os SGBD's Fonte: (DB-Engines, 2024).

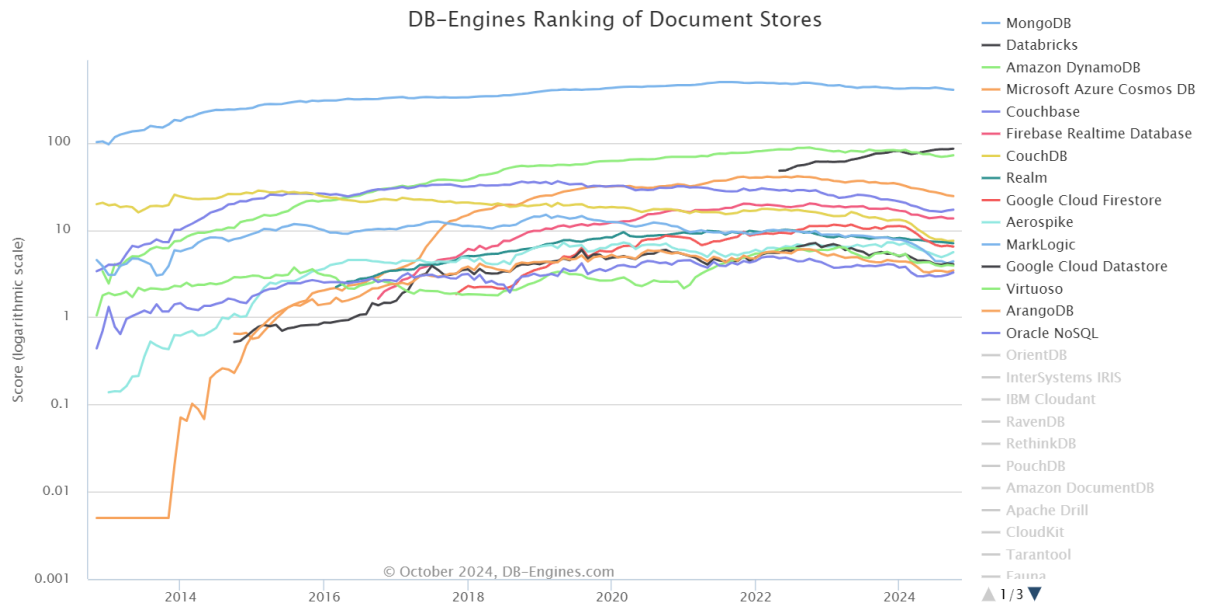


Figura 2 – Ranking de todos os SGBD's baseados em documentos Fonte: (DB-Engines, 2024).

1.1 Objetivos

O objetivo deste trabalho foi testar o desempenho de um SGBD que suporta a criação de um armazém de dados NoSQL, por meio de consultas agregadas em uma extensa base de dados que simulou, o máximo possível, um ambiente real. Também foram demonstradas as vantagens percebidas ao utilizar tal solução, assim como todos os problemas encontrados durante o desenvolvimento. Os objetivos específicos deste trabalho foram:

- Desenvolver/replicar um ambiente onde é possível testar o SGBD.
- Estudar e analisar a performance do SGBD através de queries que utilizaram de operações de agregação para elaborar um relatório sobre os aspectos positivos e negativos observados, incluindo tempo de resposta e quantidade de dados processados.
- Demonstrar quais as vantagens e desvantagens percebidas ao longo do desenvolvimento do trabalho em se utilizar um armazém de dados baseado em NoSQL.

1.2 Justificativa

Este trabalho busca auxiliar desenvolvedores e demais partes interessadas do negócio a compreender melhor o modelo orientado a documentos, avaliando sua aplicabilidade em ambientes reais, além de suas vantagens e desvantagens. A pesquisa também pode fornecer informações úteis para pessoas que desejam aprofundar seus conhecimentos ou

desenvolver novas investigações sobre o tema, aproveitando alguns dos conceitos descritos e testados aqui. Por estes motivos, todas as ferramentas e códigos utilizados serão disponibilizados e documentados, visando oferecer um recurso de apoio a futuros interessados.

1.3 Organização do texto

Este trabalho está organizado da seguinte forma: primeiramente, no Capítulo 2, Referencial Teórico, são abordados os conceitos essenciais para a compreensão do estudo, tais como dados e seus tipos (estruturados, semiestruturados e não estruturados), o conceito e as propostas de armazém de dados baseados em SQL e NoSQL, a real necessidade de uma abordagem NoSQL, os contextos nos quais o NoSQL é apropriado, e uma comparação lógica entre o modelo orientado a documentos e o modelo relacional. Em seguida, no Capítulo 3, Trabalhos Correlatos, é apresentada uma revisão da literatura e pesquisas relacionadas aos temas discutidos. Nesta seção, foram selecionadas pesquisas com maior sinergia com o que será apresentado, buscando entender, através de outros pesquisadores, suas conclusões e visões sobre o tema. No Capítulo 4, Método de Trabalho, são descritos os procedimentos adotados no desenvolvimento da pesquisa, detalhando todas as informações técnicas referentes ao desenvolvimento do projeto. Já o Capítulo 5, Análise e Discussão dos Resultados Obtidos, apresenta os resultados do trabalho, além dos desafios e limitações encontrados durante o estudo. O Capítulo 6, Conclusões e Recomendações Para Trabalhos Futuros, traz algumas recomendações para caso futuramente alguém tenha o interesse em se basear neste projeto para realizar novos estudos. Para finalizar, Referências, traz a lista completa das fontes consultadas ao longo deste projeto e no apêndice podem ser consultados os códigos desenvolvidos para o projeto.

2 Referencial Teórico

Notícias como a publicada em [The Economist \(2017\)](#) destacam que os dados se tornaram um recurso com valor superior ao do petróleo, evidenciando a crescente importância dos dados para as empresas. Muitas já reconhecem que o valor gerado por seus dados pode superar o de outros ativos. Neste contexto, se faz necessário compreender como as organizações conseguem extrair informações de conjuntos de dados que, à primeira vista, podem parecer desconexos, utilizando-os para estudar e melhorar seus negócios. Além disso, é relevante examinar a evolução das arquiteturas de armazém de dados ao longo do tempo para identificar como e onde os modelos NoSQL se encaixam.

2.1 Dados

Para compreender armazém de dados, modelagem de dados e arquiteturas, é fundamental começar com uma definição básica do que são dados. Segundo [Statistics \(2024\)](#), dados são tudo aquilo que pode ser coletado como fonte de informação. Estes dados, quando gerados e armazenados digitalmente, podem ser manipulados por um computador e, quando analisados, fornecem informações relevantes para os interessados. No geral, é possível classificar os dados em dois diferentes grupos:

- Quantitativos: são todos os dados que podem ser contados, quantificados e mensurados ([DEHALWAR; SHARMA, 2024](#)).
- Qualitativos: dados descritivos que representam características, qualidades, sendo utilizados para descrever atributos ou propriedades que não podem ser expressos numericamente ([DEHALWAR; SHARMA, 2024](#)).

Com o crescimento acelerado na geração de dados por diversas fontes, surgiu também a necessidade de novas formas de armazenamento dessas informações ([PRAVEEN; CHANDRA, 2020](#)). Neste contexto, avaliar o desempenho dos sistemas de armazém de dados requer uma compreensão da natureza e diversidade dos dados que eles processam. Esses dados podem apresentar diferentes formas e estruturas, sendo classificados como estruturados, semiestruturados ou não estruturados.

2.1.1 Estruturados

Dados estruturados são aqueles organizados em um formato predefinido, geralmente quantitativo. Exemplos incluem arquivos CSV, planilhas Excel ou bancos de dados

relacionais em SQL ([AKSHAY M. MANU et al., 2024](#)). Na Figura 3, é apresentado um exemplo de dados estruturados em um SGBD relacional.

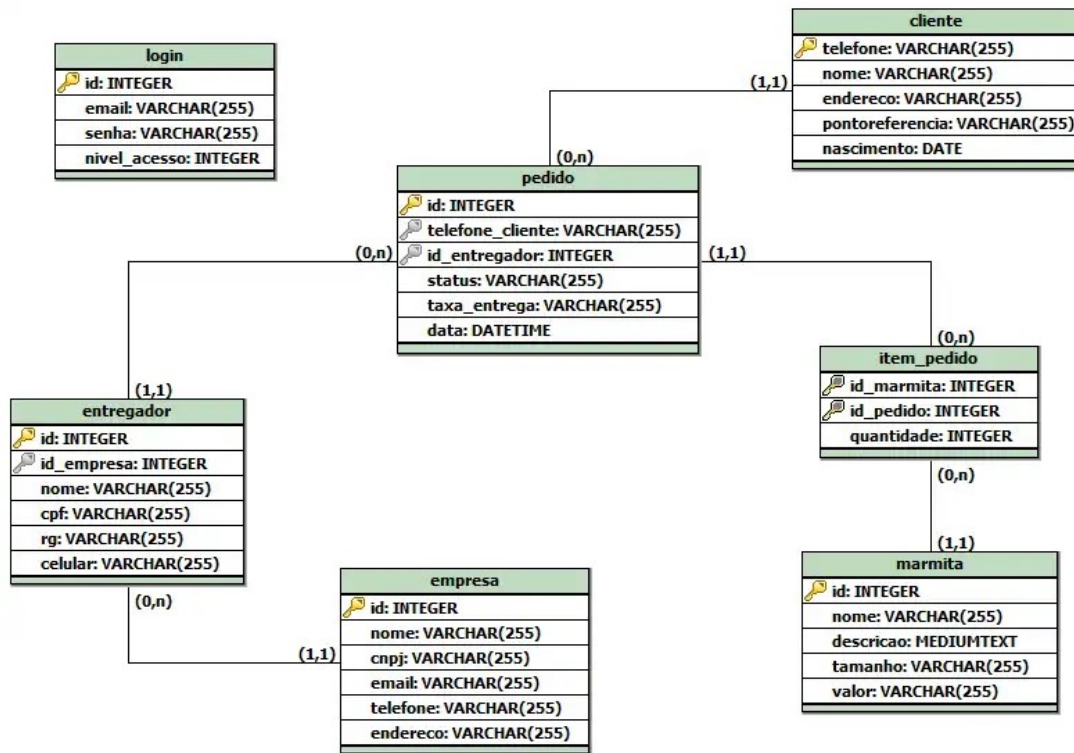


Figura 3 – Exemplo de dados estruturados. Fonte: ([BARROSO, 2023](#)).

2.1.2 Semiestruturados

Os dados semiestruturados têm ganhado relevância com a crescente demanda por sistemas de software web. Embora possuam uma organização semântica, estes dados não seguem uma estrutura rígida e podem variar em sua organização. Um exemplo comum de dados semiestruturados são os arquivos em XML ([SARKAR, 2012](#)), como demonstrado na Figura 4.

```
<?xml version="1.0" encoding="UTF-8"
standalone="no" ?>
<!DOCTYPE SBBD SYSTEM SBBD.dtd>
<SBBD ano="2000">
</artigo>
  <título> Dados Semi-Estruturados </título>
  <autor>
    <nomea> Ronaldo Mello </nomea>
    <instituição>
      <nomei> UFRGS </nomei>
      <endereço> Bento Gonçalves, 9500
      </endereço>
    </instituição>
    <instituição>
      <nomei> UFSC </nomei>
    </instituição>
    <email> ronaldo@inf.ufrgs.br </email>
    <email> ronaldo@inf.ufsc.br </email>
  </autor>
  <autor>
    <nomea> Carlos Heuser </nomea>
    <instituição>
      <nomei> UFRGS </nomei>
      <endereço> Bento Gonçalves, 9500
      </endereço>
    </instituição>
    <email> heuser@inf.ufrgs.br </email>
  </autor>
  <resumo> ... </resumo>
  <seção>
    <nomes> Introdução </nomes>
    <texto> ... </texto>
  </seção>
  ...
</artigo>
...
</SBBD>
```

Figura 4 – Exemplo de dados semiestruturados (MELLO et al., 2000)

2.1.3 Não estruturados

Os dados não estruturados são geralmente associados a informações qualitativas e podem ser originados de uma ampla variedade de fontes, como postagens em redes sociais, dispositivos IoT, vídeos e mensagens de voz. Ao contrário dos dados estruturados, os dados não estruturados não seguem um formato ou estrutura predefinidos, o que requer métodos específicos para seu armazenamento e análise. Devido à sua falta de estrutura, estes dados são flexíveis e comumente associados a bancos de dados NoSQL (AKSHAY M. MANU et al., 2024). Seguem exemplos de coleções em um SGBD NoSQL na Figura 5.

```

Coleção: deputados
{
  _id: Int32
  nome: String
  ufNascimento: String
  idLegislaturaInicial: Int32
  idLegislaturaFinal: Int32
}

Coleção: deputadosPartidos
{
  _id: Int32
  dados_id_legislatura: Int32
  dados_nome: String
  dados_sigla_partido: String
  dados_sigla_uf: String
  dados_id_partido: String
}

Coleção: frentes
{
  _id: Int32
  titulo: String
  dataCriacao: String
}

Coleção: frentesDeputados
{
  _id: ObjectId
  titulo: String
  frente_id: Int32
}

Coleção: legislaturas
{
  _id: Int32
  dataInicio: String
  dataFim: String
}

Coleção: partidos
{
  _id: Int32
  dados_nome: String
  dados_sigla: String
}

Coleção: proposicoes
{
  _id: Int32
  siglaTipo: String
  numero: Int32
  ano: Int32
  ementa: String
  ementaDetalhada: String
  keywords: String
  dataApresentacao: String
}

Coleção: votacoes
{
  _id: Int32
  proposicao: {
    Sigla: String
    Numero: String
    Ano: String
    Votacoes: (Mixed) {
      Votacao: [
        @Resumo: String
        @Data: String
        @Hora: String
        @ObjVotacao: String
        @codSessao: String
        votos: {
          Deputado: [
            @Nome: String
            @IdeCadastro: String
            @Partido: String
            @UF: String
            @Voto: String
          ]
        }
      ]
    }
  }
  orientacaoBanca: {
    bancada: [
      @Sigla: String
      @Orientacao: String
    ]
  }
}

```

Figura 5 – Exemplo dados não estruturados (BERRUTTI, 2019).

2.2 Armazém de dados

Em 1988, Barry Devlin e Paul Murphy, arquitetos da IBM, lançaram as bases do que hoje chamamos de armazém de dados empresariais, conforme ilustrado na Figura 6. Em seu trabalho seminal (DEVLIN; MURPHY, 1988), eles relataram que cada empresa possuía diferentes formas de lidar com dados, tornando desafiador para um usuário novato estruturar e criar relatórios a partir dessas informações. Para abordar esses problemas, Devlin e Murphy propuseram um modelo que oferecia uma solução para a complexidade e a dificuldade de gestão de dados nas organizações, introduzindo assim o conceito de armazém de dados empresariais.

Anos depois, Ralph Kimball e Margy Ross, em (KIMBALL; ROSS, 2013), que teve sua primeira edição lançada em 1996, observaram problemas semelhantes. Empresas enfrentavam dificuldades no acesso a informações relevantes, decisões baseadas em suposições e perda de tempo com dados irrelevantes. Como resposta a esses desafios, Kimball e Ross propuseram uma abordagem de modelagem dimensional dos dados, que facilita o acesso à informação e garante a apresentação de dados consistentes e confiáveis.

De forma complementar, (INMON, 1996) que é conhecido como o Pai do armazém de dados define *Data Warehouse* como uma coleção de dados orientada a assuntos, que contém dados corporativos granulares e serve como suporte às decisões de negócio. No passado, segundo Inmon, ao desenhar aplicações, as partes responsáveis raramente consideravam que os dados de uma solução precisariam ser integrados com outros dados. Isto resultava na falta de padronização entre as aplicações e na dificuldade de correlacionar informações. O armazém de dados, portanto, surgiu como uma fonte única e integrada para armazenar os dados de forma eficiente.

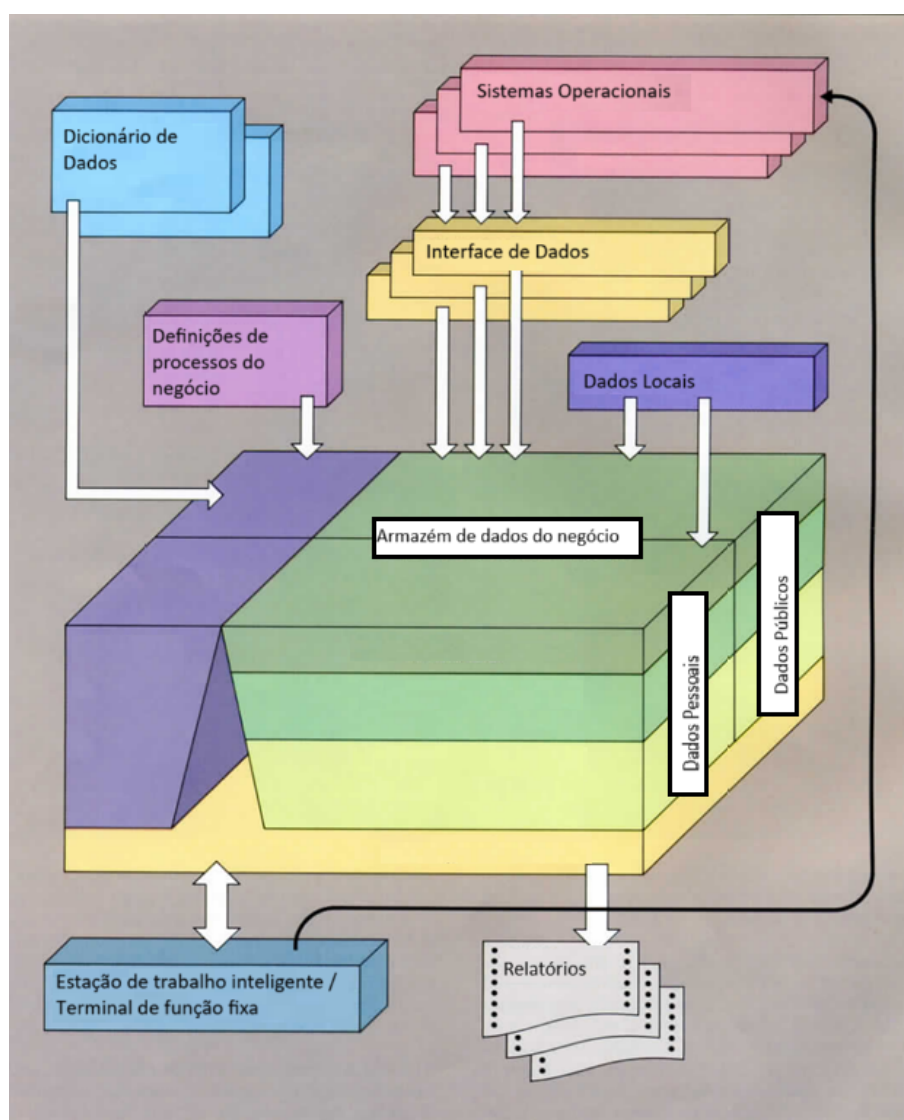


Figura 6 – Modelo descrito por Barry Devlin e Paul Murphy, imagem traduzida de (DEVLIN; MURPHY, 1988).

Os armazéns de dados modernos podem ser baseados em dois modelos principais: SQL e NoSQL. Esta monografia abordará brevemente o modelo SQL para facilitar a compreensão das diferenças, com uma análise mais aprofundada do modelo NoSQL ao longo do trabalho.

2.2.1 Armazém de Dados Baseado em SQL

Os armazéns de dados baseados em modelos SQL são projetados para organizar e gerenciar grandes volumes de dados, permitindo que eles sejam relacionados e que o conhecimento seja extraído. Esses sistemas seguem uma definição rígida de esquemas e padronização dos dados (KHAN et al., 2023). A modelagem dimensional e relacional é amplamente utilizada, estruturando os dados de modo a facilitar a análise e a geração de relatórios, apoiando a tomada de decisões nas empresas. É fundamental possuir conhecimento básico sobre o funcionamento dessas modelagens para compreender a crescente relevância dos bancos NoSQL no mercado. Neste trabalho, apresenta-se apenas um resumo do modelo dimensional.

2.2.1.1 Dimensional

A modelagem dimensional representada na Figura 7 é uma das formulas mais populares em se organizar um banco de dados. Focada em simplificar o banco de dados, essa técnica visa entregar os dados de maneira que seja fácil para o usuário compreender, além de garantir um desempenho eficiente (KIMBALL; ROSS, 2013).

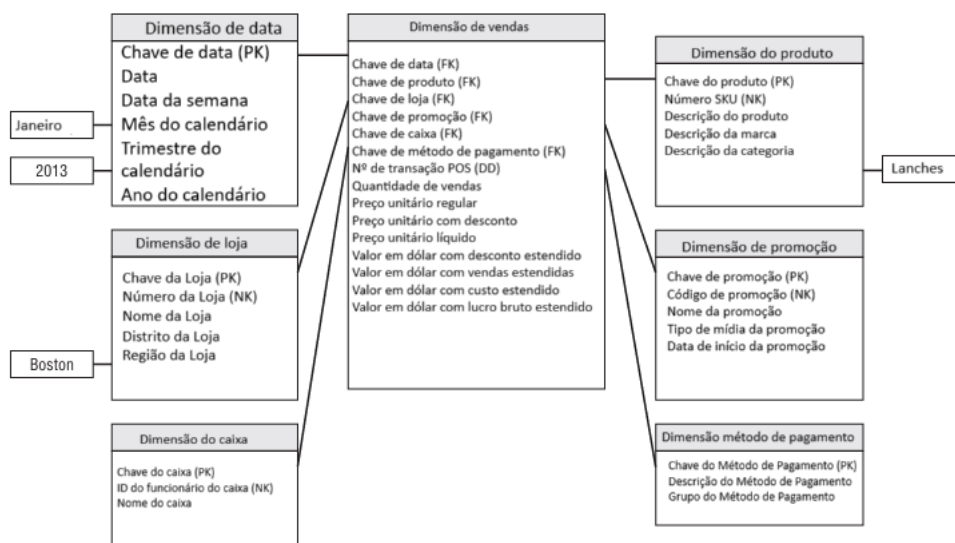


Figura 7 – Exemplo de modelagem dimensional, imagem traduzida de (KIMBALL; ROSS, 2013).

As tabelas fatos e dimensões em um armazém de dados são a forma como os dados são organizados dentro da estrutura dimensional, fazendo com que facilite a busca e correlação dos dados por analistas, tal modelagem utiliza dos dados estruturados que foram apresentados anteriormente no trabalho, com as estruturas previamente montadas é possível relacionar as diversas tabelas do sistema por meio de suas chaves primárias e assim gerar uma fonte de verdade.

2.2.2 Armazém de Dados Baseado em NoSQL

Após compreender os conceitos fundamentais de *Data Warehouse* em SQL, é importante destacar que a adoção de um armazém de dados em NoSQL visa alcançar benefícios similares para o negócio, como acesso rápido e organizado às informações, além de maior agilidade no processamento de dados. Diferente dos modelos SQL, que seguem uma rígida definição de esquema de dados e correlação entre tabelas, os bancos de dados NoSQL oferecem maior flexibilidade, permitindo que empresas lidem com diferentes tipos de dados e volumes de maneira eficiente.

2.2.2.1 Documentos

O modelo orientado a documentos é projetado para armazenar e gerenciar dados em formatos de documentos, com esquemas de dados flexíveis e escalabilidade horizontal. Diferentemente dos bancos de dados relacionais, que organizam dados em tabelas e linhas, os bancos de dados orientados a documentos armazenam informações em documentos individuais, frequentemente no formato JSON ou XML. Cada documento é uma unidade de informação auto-suficiente, capaz de conter estruturas de dados complexas, como arrays e listas (CHEVALIER et al., 2016). O modelo é o mais citado na literatura, como mostrado em (OLIVEIRA; VICTORINO; HOLANDA, 2021).

Esta abordagem oferece uma maior flexibilidade em comparação aos modelos tradicionais de *Data Warehouse*, onde mudanças no esquema exigem a modificação de várias tabelas. No modelo orientado a documentos, a facilidade de gerenciar documentos torna desnecessárias alterações extensivas em caso de mudanças nos dados armazenados, o que simplifica a manutenção do armazém de dados (ODITIS et al., 2018).

Algumas das principais vantagens de utilizar o modelo orientado a documentos incluem:

- **Flexibilidade de Esquema:** O modelo orientado a documentos permite que os dados sejam armazenados sem um esquema fixo, o que significa que a estrutura dos documentos pode variar. Isto facilita a adaptação às mudanças nos requisitos e a inclusão de novos tipos de dados sem necessidade de reestruturação e longo tempo de desenvolvimento.
- **Armazenamento de Dados Complexos:** Documentos podem conter dados complexos, como arrays e listas, o que torna a modelagem de dados mais natural e alinhada com a forma como as aplicações modernas manipulam informações.
- **Escalabilidade Horizontal:** A escalabilidade horizontal permite adicionar novos nós ao sistema para aumentar a capacidade de armazenamento e processamento.

Em bancos de dados NoSQL, como MongoDB, os dados podem ser facilmente particionados e replicados entre diferentes nós, o que facilita o manuseio de grandes volumes de dados e alta demanda. Isso proporciona maior flexibilidade e eficiência, reduzindo custos em comparação com a escalabilidade vertical dos bancos de dados SQL.

- **Modelo Natural para Desenvolvimento:** A estrutura dos documentos pode se mapear de forma mais intuitiva para a forma como os dados são manipulados em muitas linguagens de programação, como JSON .

Algumas das principais desvantagens de utilizar o modelo orientado a documentos incluem:

- **Consistência e Normalização:** A flexibilidade do esquema pode levar a desafios na manutenção da consistência. Dados relacionados podem ser armazenados em documentos separados, o que pode resultar em inconsistências se não forem cuidadosamente gerenciados.
- **Consultas Complexas:** Pode haver limitações na realização de consultas complexas que envolvem múltiplos documentos ou que exigem operações JOIN.
- **Gerenciamento de Transações:** A capacidade de realizar transações complexas e garantir a integridade dos dados pode ser limitada em comparação com bancos de dados relacionais que garantem ACID.

Quando se trata da organização de documentos em uma base de dados orientada a documentos, existem diferentes abordagens. (CHEVALIER et al., 2016) identifica quatro modelos principais:

Modelo Plano: O modelo demonstrado na Figura 8 corresponde a um modelo desnormalizado, onde um fato e todos os atributos das dimensões correspondentes a este fato são armazenados juntos em um único documento. Isso simplifica a consulta, mas pode gerar redundância de dados.

```
{
  "id": 1,
  "l_quantidade": 4,
  "l_modulo_envio": "correio",
  "l_preco": 400.0,
  "c_nome": "João",
  "c_cidade": "Roma",
  "c_nacao_nome": "Itália"
}
```

Figura 8 – Modelo Plano, imagem adaptada de (CHEVALIER et al., 2016).

1. **Modelo Deco:** O modelo da Figura 9 ainda é um modelo desnormalizado, mas com mais estrutura. Fatos e dimensões são agrupados em subdocumentos, proporcionando uma organização mais clara e estruturada dos dados, também gerando redundância.

```
{
  "id": 1,
  "PedidoLinha": {
    "l_quantidade": 4,
    "l_modulo_envio": "correio",
    "l_preco": 400.0
  },
  "Cliente": {
    "c_nome": "João",
    "c_cidade": "Roma",
    "c_nacao_nome": "Itália"
  }
}
```

Figura 9 – Modelo Deco, imagem adaptada de (CHEVALIER et al., 2016).

2. **Modelo Fragmentado:** O modelo da Figura 10 é um modelo equivalente ao normalizado em que os registros de fatos são armazenados separadamente das dimensões, evitando redundância. Os documentos de fatos contêm chaves estrangeiras que referenciam as dimensões.

```
{
  "id": 1,
  "l_quantidade": 4,
  "l_modulo_envio": "correio",
  "l_preco": 400.0,
  "c_id": 4
}

{
  "id": 4,
  "c_nome": "João",
  "c_cidade": "Roma",
  "c_nacao_nome": "Itália"
}
```

Figura 10 – Modelo Fragmentado, imagem adaptada de (CHEVALIER et al., 2016).

3. **Modelo Híbrido:** Um modelo que combina elementos dos outros modelos, permitindo o armazenamento de documentos com diferentes esquemas em uma única coleção. Este modelo oferece flexibilidade, permitindo consultas eficientes com o uso de índices apropriados, modelo demonstrado na Figura 11.

```
{
  "id": 1,
  "l_quantidade": 4,
  "l_modulo_envio": "correio",
  "l_preco_extendido": 400.0,
  "c_codigo_cliente": 2,
  "c_codigo_data": 3
}

{
  "id": 2,
  "codigo_cliente": 4,
  "c_nome": "João",
  "c_cidade": "Roma",
  "c_nacao_nome": "Itália",
  "c_regiao_nome": "Europa"
}

{
  "id": 3,
  "codigo_data": 1,
  "d_dia": 10,
  "d_mes": "Janeiro",
  "d_ano": 2014
}
```

Figura 11 – Modelo Híbrido, imagem adaptada de (CHEVALIER et al., 2016).

2.3 MongoDB

Foi ao perceberem as limitações dos modelos relacionais em termos de escalabilidade e disponibilidade de dados que Dwight Merriman, Eliot Horowitz e Kevin Ryan decidiram criar o MongoDB, um banco de dados orientado a documentos projetado para resolver esses problemas, permitindo a escalabilidade da solução na nuvem em diversos nós (MONGODB, s.d.). Para construir um armazém de dados baseado em documentos, é necessário compreender dois dos principais componentes do MongoDB:

- **Coleções (Collections):** O MongoDB armazena os documentos em estruturas chamadas coleções, representadas na Figura 12, que podem ser comparadas às tabelas do modelo tradicional ([MONGODB, s.d.](#)).



Figura 12 – Exemplo de coleções, imagem traduzida de ([MONGODB, s.d.](#)).

- **Banco de dados (Database):** No MongoDB, um banco de dados é responsável pelo armazenamento de diversas coleções. Na Figura 13, é possível ver a estrutura do MongoDB.



Figura 13 – Organização da estrutura do MongoDB, imagem traduzida de ([studyopedia, s.d.](#)).

Uma das principais diferenças entre o MongoDB e os bancos de dados baseados em SQL é a forma como as consultas são realizadas. Enquanto os bancos de dados SQL utilizam a Linguagem de Consulta Estruturada (SQL), como mostrado na Figura 14, para manipular dados organizados em tabelas, o MongoDB emprega a MongoDB Query Language (MQL), ilustrada na Figura 15, permitindo trabalhar com documentos nos formatos mencionados anteriormente.

```
SELECT nome, email
FROM clientes
WHERE cidade = 'Uberlandia';
```

Figura 14 – Exemplo de SQL.

```
db.clientes.find(
  { cidade: 'Uberlandia' },
  { nome: 1, email: 1 }
);
```

Figura 15 – Exemplo de MQL.

2.4 Necessidade de uma Abordagem NoSQL

À medida que o volume e a complexidade dos dados nas organizações continuam a crescer, os modelos tradicionais de bancos de dados relacionais podem se tornar insuficientes para atender às demandas modernas. Surge, então, a necessidade de uma abordagem NoSQL como resposta a essas demandas e às limitações dos sistemas relacionais tradicionais na análise de dados não estruturados (OLIVEIRA; VICTORINO; HOLANDA, 2021).

2.4.1 Situações em que NoSQL é mais apropriado

O uso de bancos de dados NoSQL se destaca em diversas situações nas quais os bancos de dados relacionais podem se mostrar inadequados. Uma dessas situações é o desenvolvimento ágil, no qual iterações rápidas e mudanças frequentes são essenciais. NoSQL permite que os desenvolvedores tenham maior controle sobre a estrutura dos dados, eliminando a necessidade de ajustes complexos e demorados no esquema do banco

de dados, o que torna o processo de desenvolvimento mais ágil e eficiente (YANGUI; NABLI; GARGOURI, 2017).

Em termos de escalabilidade, os bancos de dados NoSQL utilizam uma estratégia de "scale-out", permitindo adicionar novos servidores à medida que a necessidade de armazenamento e processamento aumenta, o que reduz os custos e melhora a capacidade de lidar com volumes massivos de dados e tráfego elevado. Além disso, a arquitetura de "scale-out" facilita atualizações e mudanças na estrutura sem interrupções no serviço (AUTHOR, 2011).

2.4.2 Comparação com o modelo SQL

Nos bancos de dados relacionais, os dados são organizados em tabelas estruturadas com linhas e colunas, seguindo um esquema rígido e previamente definido. Essa estrutura impõe uma uniformidade que, embora útil em muitos casos, pode limitar a flexibilidade na manipulação de dados não estruturados ou semiestruturados, especialmente em comparação com o modelo baseado em documentos.

No modelo orientado a documentos, cada documento é autodescritivo, ou seja, além dos valores dos dados, ele inclui informações sobre a natureza desses dados, como ilustrado na Figura 16. Isto permite que os documentos sejam compreendidos com facilidade, pois tanto o conteúdo quanto a estrutura são apresentados de forma completa, simplificando a recuperação e interpretação dos dados Papiernik e Drake (2021).

A Figura 17 apresenta um exemplo de esquema dinâmico, destacando a flexibilidade do modelo orientado a documentos—uma característica que seria mais complexa de implementar em um banco de dados relacional.

```
{
  "id_usuario": "001",
  "nome": "Carlos Silva",
  "email": "carlos.silva@example.com"
}
```

Figura 16 – Exemplo de schema no modelo baseado em documentos

```
{
  "id_usuario": "002",
  "nome": "Mariana Souza",
  "email": "mariana.souza@example.com",
  "telefone": [
    "+55-11-99999-1234",
    "+55-11-98888-5678"
  ],
  "endereço": {
    "rua": "Rua das Palmeiras, 123",
    "cidade": "São Paulo",
    "estado": "SP",
    "cep": "01000-000"
  }
}
```

Figura 17 – schema dinâmico no modelo baseado em documentos

Para criar um modelo similar ao do banco de dados orientado a documentos em um banco de dados relacional, é necessário seguir os passos descritos nas Figuras 18, 19 e 20 para garantir que as informações estejam bem estruturadas e normalizadas. A seguir, estão os passos necessários para criar as tabelas correspondentes:

Primeiro, é necessário criar a tabela `Usuarios`, que armazenará as informações básicas de cada usuário, como nome e e-mail.

```
CREATE TABLE Usuarios (
  id_usuario VARCHAR(10) PRIMARY KEY,
  nome VARCHAR(100),
  email VARCHAR(100) UNIQUE
);
```

Figura 18 – Exemplo de criação da tabela usuário

Em seguida, criamos uma tabela `Telefones`. Dessa forma, a tabela `Telefones` armazena os números de telefone e os relaciona com a tabela `Usuários` através de `id_usuario`.

```
CREATE TABLE Telefones (  
    id_usuario VARCHAR(10),  
    telefone VARCHAR(20),  
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario)  
);
```

Figura 19 – Exemplo de criação da tabela telefones

Por fim, a tabela Endereços, onde o endereço de cada usuário é armazenado e relacionado com a tabela Usuários através de id_usuario.

```
CREATE TABLE Enderecos (  
    id_usuario VARCHAR(10),  
    rua VARCHAR(200),  
    cidade VARCHAR(100),  
    estado VARCHAR(2),  
    cep VARCHAR(10),  
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario)  
);
```

Figura 20 – Exemplo de criação da tabela endereços

3 Trabalhos Correlatos

Nesta seção, apresento e analiso alguns dos trabalhos realizados na área de armazém de dados baseados em NoSQL. O objetivo é explorar como diferentes estudos têm abordado e relatado as vantagens e desvantagens do uso de abordagens NoSQL, especialmente no contexto de armazém de dados baseados em documentos. Esses trabalhos fornecem informações valiosas sobre as experiências que outros pesquisadores tiveram ao utilizar esta abordagem. Os trabalhos aqui mencionados foram escolhidos por sua sinergia com o tema aqui apresentado.

O trabalho realizado em (OLIVEIRA; VICTORINO; HOLANDA, 2021) foi crucial para a definição do modelo que seria utilizado nesta monografia, pois fez o mapeamento da literatura sobre armazéns de dados NoSQL. Esse mapeamento possibilitou compreender que o modelo orientado a documentos é o que oferece a maior quantidade de estudos, também demonstrando que, nesses estudos, o MongoDB era amplamente utilizado. O trabalho apresentado por (ODITIS et al., 2018) destaca várias vantagens associadas ao uso de armazém de dados baseados em NoSQL. No estudo, eles se referenciam a um outro estudo que pode ser visto em (LEAVITT, 2010) onde demonstra que os bancos NoSQL oferecem suporte para as funções agregadas (AVG, SUM e etc), porém sempre de uma forma inferior quando se comparado com os armazém de dados tradicionais, mas também demonstram que esses sistemas oferecem benefícios significativos em termos de flexibilidade, o que elimina grande parte da complexidade dos modelos relacionais, onde diversas tabelas precisam ser criadas para chegar ao mesmo resultado. O trabalho realizado por (MONTEIRO, 2023) explora uma abordagem alternativa, utilizando relacionamentos entre coleções. Essa abordagem contribui para a compreensão de diferentes formas de organizar e estruturar dados em bancos de dados orientados a documentos. Entre as principais desvantagens destacadas por (ODITIS et al., 2018), está a falta de suporte robusto para ferramentas de relatórios e análise de dados. Ao contrário dos sistemas tradicionais de armazém de dados, que oferecem uma ampla gama de soluções para este fim. O artigo de (BOUAZIZ; NABLI; GARGOURI, 2019) apresenta cinco passos cruciais para o desenvolvimento de um esquema de armazém de dados a partir de bancos de dados orientados a documentos. Os passos descritos são: escolha do banco de dados NoSQL, extração do esquema, identificação do gráfico de estrutura, definição dos conceitos multidimensionais e modelagem do esquema do armazém de dados. O trabalho realizado em (DABBÈCHI et al., 2021) foi de grande ajuda para a compreensão do mapeamento e desenvolvimento de um modelo de armazém de dados baseado em documentos. No estudo, os autores realizaram a transição de um Data Lake para um armazém de dados NoSQL. Para o desenvolvimento, foram utilizados os databases MongoDB e Cassandra. Os autores também apresentam de forma

breve como foi feito o processo de integração destes dados e como foi implementada a solução. O trabalho realizado por ([CHEVALIER et al., 2016](#)) apresenta de forma clara as diferentes maneiras de estruturar os arquivos em um banco de dados NoSQL. Além disso, mostra diversos testes realizados, o que ajudou a entender melhor como criar um banco em NoSQL e sua estrutura.

4 Método de Trabalho

O método de trabalho descrito na Figura 21 demonstra como foi realizado o desenvolvimento do estudo, detalhando as seguintes etapas: sistema de gerenciamento de banco de dados NoSQL escolhido, detalhes sobre as configurações do ambiente utilizado para os testes, definição do modelo de dados utilizado, como foi realizado o processo de carga dos dados, as informações obtidas através do processo de carga dos dados, a criação das consultas utilizadas para os testes, informações gerais sobre a execução das consultas e, por fim, as considerações finais.

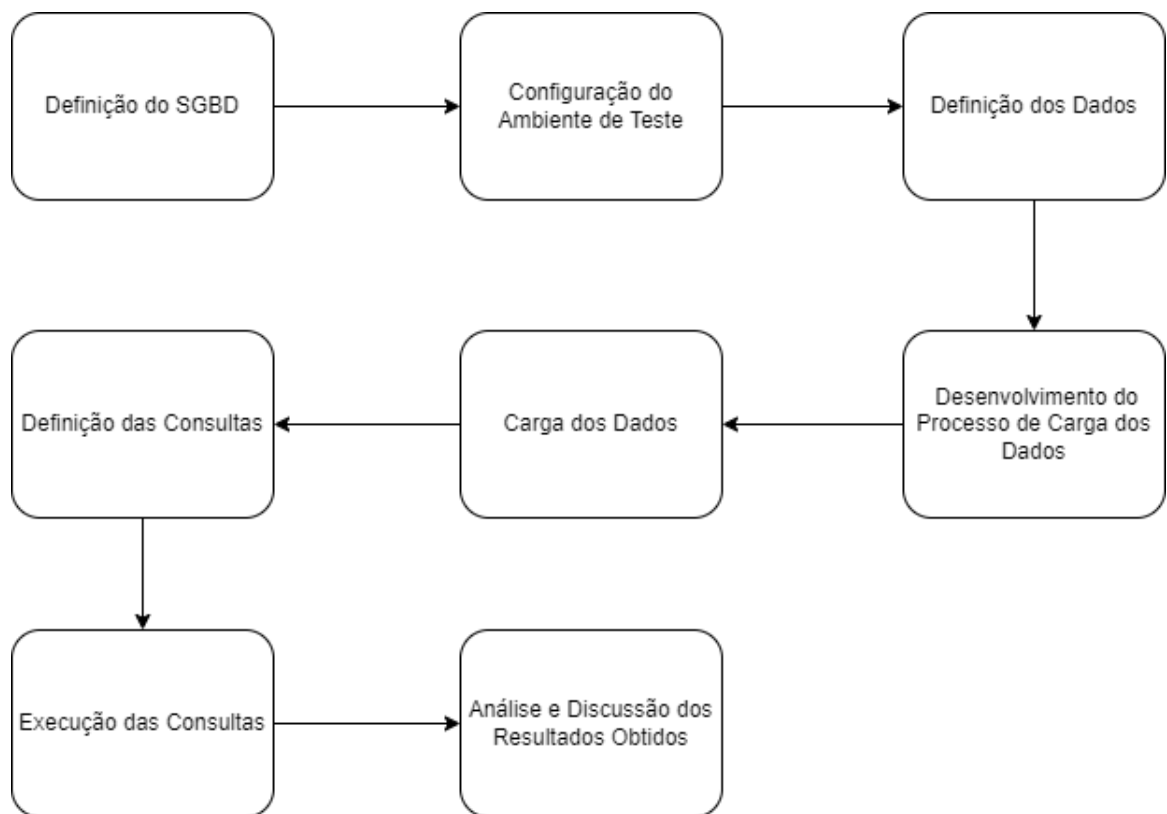


Figura 21 – Diagrama do método de trabalho.

4.1 Definição do SGBD

Como mencionado anteriormente, o SGBD escolhido para este projeto foi o MongoDB, devido à sua popularidade e à sua capacidade de lidar com grandes volumes de dados de forma flexível.

4.2 Configuração do Ambiente de Teste

MongoDB oferece diferentes opções para se configurar o seu ambiente para fins de desenvolvimento:

- **MongoDB Atlas:** Um serviço de banco de dados totalmente gerenciado e baseado em nuvem. Oferece funcionalidades como backups automatizados, monitoramento em tempo real, segurança aprimorada e escalabilidade flexível, ajustando o desempenho conforme a demanda da carga de trabalho.
- **MongoDB Compass:** Uma interface gráfica avançada que facilita a exploração e gestão dos bancos de dados MongoDB. A ferramenta permite executar consultas e gerenciar coleções. O Compass também oferece visualização do esquema dos dados, tornando mais intuitivo o gerenciamento do banco de dados.
- **MongoDB Enterprise:** Versão comercial da ferramenta, com funções extras que visa atender as diversas necessidades dos usuários.
- **MongoDB Community:** Versão para a comunidade com algumas funções limitadas, também é oferecida para os usuários a possibilidade de se utilizar um cluster de 512MB gerenciado na nuvem.

Neste projeto, foi utilizada a versão Community do MongoDB junto com o MongoDB Compass. O Compass foi especialmente útil para uma visualização clara das coleções, facilitando o entendimento dos dados armazenados. A seguir, são apresentadas as especificações da versão do MongoDB, o hardware da máquina e outros detalhes relevantes do ambiente de desenvolvimento.

- **MongoDB:** MongoDB Community versão 7.0.14 para plataforma Windows x64 com package MSI. Também foi utilizado do MongoDB Compass para facilitar a visualização do desenvolvimento e resultados durante o decorrer do projeto.
- **Hardware:**
 - CPU: Intel Core i7 12700K
 - Memória RAM: DDR4 32GB 3600MHz
 - Vídeo: NVIDIA GeForce RTX 3060
 - Placa-mãe: B660M AORUS PRO DDR4
 - SSD M.2: SeaFargate FireCude 520 2TB Gen4 (4850/4750MB/s)
- **Linguagem de desenvolvimento:** Python 3.12

Após a instalação, foi conectado ao MongoDB na *localhost* (Figura 22) para uma visualização inicial do ambiente. Esta configuração permite explorar as funcionalidades básicas e garantir que tudo esteja pronto para o desenvolvimento e testes subsequentes.

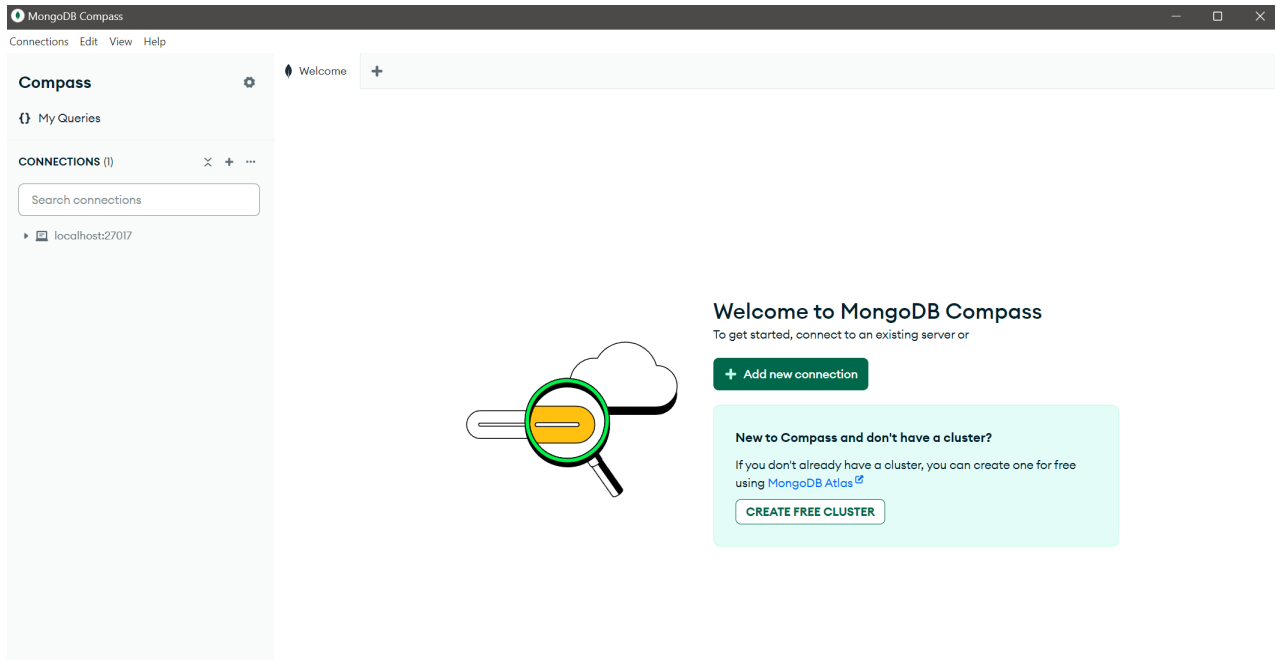


Figura 22 – Tela inicial MongoDBCompass.

4.3 Definição dos Dados

Para a realização dos testes, adotou-se a abordagem de Documentos Embutidos (*Embedded Documents*), conforme descrito em (CHEVALIER et al., 2016), que se refere à modelagem plana dos dados também apresentada no Capítulo 2 deste trabalho. Este modelo apresenta várias vantagens, como a facilidade de leitura e visualização dos dados. No entanto, uma desvantagem notável é a possibilidade de duplicação de dados, que podem ser replicados em múltiplos documentos. Um exemplo disto são os produtos contidos na lista `produtos_vendidos`, que aparecerão em todos os clientes que tiverem comprado os mesmos produtos, como ilustrado na Figura 23. Outro ponto a ser observado é o limite de tamanho de 12 MB por arquivo, conforme descrito em (MongoDB, s.d.).

Neste projeto, a biblioteca Faker do Python (JOKE2K, 2023) foi utilizada para gerar dados fictícios de maneira eficiente, simulando transações de vendas de produtos. Estes dados foram armazenados em uma coleção única, chamada `registro_de_vendas`, cuja estrutura está representada na Figura 23.

Os dados simulam cenários de transações de vendas, onde cada cliente possui um cadastro com informações pessoais, que podem estar completas ou parcialmente registradas para que também seja possível realizar os testes referentes a flexibilidade do esquema

de dados. Em cada transação, o cliente é atendido por um vendedor, e os produtos adquiridos são detalhados, compondo um histórico de vendas.

```
{
  "nome_cliente": "Ana Machado",
  "endereco_cliente": "Rua Teste",
  "telefone_cliente": "81 1965-6674",
  "emprego_cliente": "Paparazzo",
  "email_cliente": "diasana-laura@example.com",
  "nome_vendedor": "Brito da Luz S/A",
  "data": "1970-12-18",
  "valor_total_venda": 1233.7,
  "produtos_vendidos": [
    {
      "nome_produto": "Alto-falante Borgonha 1475542692",
      "preco_unitario": 67.95,
      "quantidade": 9,
      "valor_total": 611.55
    },
    {
      "nome_produto": "VideoGame Azul turquesa brilhante 7607862449",
      "preco_unitario": 55.64,
      "quantidade": 9,
      "valor_total": 500.76
    },
    {
      "nome_produto": "Liquidificador Turquesa média 449539850",
      "preco_unitario": 37.75,
      "quantidade": 3,
      "valor_total": 113.25
    },
    {
      "nome_produto": "Tablet Pardo escuro 1287380003",
      "preco_unitario": 4.07,
      "quantidade": 2,
      "valor_total": 8.14
    }
  ]
}
```

Figura 23 – Modelo dos dados definidos para o projeto.

É importante destacar que, ao utilizarmos dados fictícios, pode não haver coerência entre algumas das informações presentes, como por exemplo os nomes dos clientes e os e-mails gerados. A biblioteca Faker apresenta uma limitação: após 1.000 iterações sem encontrar um nome que ainda não foi utilizado, ela retorna um erro. Para contornar essa questão, foi adicionado um número inteiro de 10 dígitos à frente de alguns itens. Apesar disso, garantimos que cada cliente possui seu próprio documento e que as informações sobre os produtos são consistentes em todas as vendas, assim como os valores totais das

compras realizadas. Esta estrutura de dados é suficiente para realizar consultas e testar operações como *sum*, *avg*, *min* e *max*. Os dados foram definidos da seguinte maneira:

- **nome_cliente**: Gerado aleatoriamente e único para cada cliente.
- **endereco_cliente**: O endereço é registrado de forma opcional, aproveitando a flexibilidade do modelo de documentos. Assim, os clientes podem ou não ter o endereço registrado.
- **telefone_cliente**: Gerado aleatoriamente para cada cliente.
- **emprego_cliente**: Gerado aleatoriamente e, de forma opcional, pode ou não ser fornecido, testando a flexibilidade do esquema.
- **email_cliente**: Gerado aleatoriamente para cada cliente.
- **nome_vendedor**: Para fins de teste de agregação, o número de vendedores foi determinado pela proporção de clientes multiplicada por 0,01, com um limite de 10.000 vendedores distintos, garantindo a repetição de vendedores entre os clientes.
- **data**: Gerada aleatoriamente para cada transação.
- **valor_total_venda**: Representa o valor total das compras realizadas pelo cliente.
- **produtos_vendidos**: Uma lista de produtos é gerada aleatoriamente, permitindo avaliar a flexibilidade do esquema. O número de produtos varia entre 1 e 10, com a possibilidade de o usuário definir um intervalo diferente para testar volumes de dados maiores.
- **nome_produto**: Gerado aleatoriamente para cada produto vendido.
- **preco_unitario**: Gerado aleatoriamente e mantido consistente entre as vendas para o mesmo produto.
- **quantidade**: Quantidade de unidades do produto adquiridas, com valor aleatório entre 1 e 10 (o intervalo pode ser ajustado pelo usuário para testar cargas de dados mais intensas).
- **valor_total**: Calculado como o produto entre o **preco_unitario** e a **quantidade**.

4.4 Desenvolvimento do Processo de Carga dos Dados

Para gerar e carregar os dados do projeto, foi desenvolvido um código em Python que cria os dados e os carrega no MongoDB no formato JSON, seguindo as regras descritas

anteriormente. Todos os códigos utilizados no projeto estão disponíveis ao final deste trabalho.

Para realizar a carga dos dados, utilizamos a classe `CriarBaseDeDadosClientes`, conforme demonstrado na Figura 24. Para utilizá-la, basta passar uma lista contendo a quantidade de clientes que deseja simular. No caso descrito na imagem estamos criando dados fictícios para 1.000, 10.000, 100.000, 1.000.000 e 10.000.000 de clientes.

```
import pymongo
from no_sql_data_test.one_collection_document_database import CriarBaseDeDadosClientes

def main(): 1 usage  ⬆ Matheus Andrade
    numero_de_clientes_para_gerar = [1000, 10000, 100000, 1000000, 10000000]
    gerador_dados = CriarBaseDeDadosClientes()

    gerador_dados.conectar_ao_mongodb(db_host="localhost:27017")
    gerador_dados.definir_banco_de_dados(nome_banco_de_dados="vendas")

    for num_clientes in numero_de_clientes_para_gerar:
        gerador_dados.definir_colecao(nome_colecao=f"registro_de_vendas_{num_clientes}")
        gerador_dados.gerar_base_clientes(numero_de_clientes=num_clientes)

if __name__ == "__main__":
    main()
```

Figura 24 – Código para ingestão dos dados.

No projeto, os bancos de dados e coleções são gerados automaticamente utilizando Python, através da classe apresentada na Figura 25. Esta classe facilita a execução de operações essenciais no MongoDB, incluindo a conexão ao servidor, a criação de bancos de dados e coleções, inserção de dados em coleções e consultas.

```
import pymongo

class ManipuladorMongoDB: 3 usages  ⚡ Matheus Andrade
    def __init__(self):  ⚡ Matheus Andrade
        self.client = None
        self.database = None
        self.collection = None

    def conectar_ao_mongodb(self, db_host: str) -> None: 1 usage  ⚡ Matheus Andrade
        self.client = pymongo.MongoClient(db_host)
        print("Conectado ao MongoDB em: ", db_host)

    def definir_banco_de_dados(self, nome_banco_de_dados: str) -> None: 1 usage  ⚡ Matheus Andrade
        if not self.client:
            raise ConnectionError("Cliente MongoDB não está conectado. Por favor, conecte primeiro.")

        self.database = self.client[nome_banco_de_dados]
        print("Banco de dados definido: ", nome_banco_de_dados)

    def definir_colecao(self, nome_colecao: str) -> None: 1 usage  ⚡ Matheus Andrade
        if not self.client:
            raise ConnectionError("Cliente MongoDB não está conectado. Por favor, conecte primeiro.")

        self.collection = self.database[nome_colecao]
        print("Coleção definida: ", nome_colecao)

    def inserir_um_na_colecao(self, payload: dict) -> None:  ⚡ Matheus Andrade
        resultado_insercao = self.collection.insert_one(payload)

    def inserir_muitos_na_colecao(self, payload: list) -> None: 1 usage  ⚡ Matheus Andrade
        resultado_insercao = self.collection.insert_many(payload)

    def executar_consulta_agg(self, consultas: list) -> None:  ⚡ Matheus Andrade
        start_time = time.time()

        resultados = collection.aggregate(pipeline)

        end_time = time.time()
        exec_time = end_time - start_time

        print(f"Tempo de execução: {exec_time:.4f} segundos \n")

        return resultados
```

Figura 25 – Classe para interação com o MongoDB.

4.5 Carga dos Dados

Nesta seção, é apresentado o processo de carga dos dados no MongoDB. A Figura 26 ilustra uma visão geral dos dados ingeridos.

```
Conectado ao MongoDB em: localhost:27017
Banco de dados definido: vendas
Coleção definida: registro_de_vendas_1000
1000 clientes criados.
10 vendedores criados.
100 produtos diferentes criados.
Tempo total de execução: 0.12 segundos.

Coleção definida: registro_de_vendas_10000
10000 clientes criados.
100 vendedores criados.
500 produtos diferentes criados.
Tempo total de execução: 0.79 segundos.

Coleção definida: registro_de_vendas_100000
100000 clientes criados.
1000 vendedores criados.
1000 produtos diferentes criados.
Tempo total de execução: 7.95 segundos.

Coleção definida: registro_de_vendas_1000000
1000000 clientes criados.
10000 vendedores criados.
2000 produtos diferentes criados.
Tempo total de execução: 81.93 segundos.

Coleção definida: registro_de_vendas_10000000
10000000 clientes criados.
10000 vendedores criados.
2000 produtos diferentes criados.
Tempo total de execução: 951.65 segundos.
```

Figura 26 – Informação geral sobre a carga dos dados.

Durante o processo de carga, o código criou cinco diferentes coleções, conforme ilustrado na Figura 27.



Figura 27 – Coleções criadas após a carga dos dados.

Além disso, a Figura 28 fornece um exemplo de como os dados estão estruturados em uma das coleções.

```
_id: ObjectId('6723469ad1daa858058fb1bb')
nome_cliente: "Agatha Siqueira 1998912"
endereco_cliente: null
telefone_cliente: "+55 98 501407550"
emprego_cliente: "Gestor ambiental"
email_cliente: "costakamilly@example.net"
nome_vendedor: "Barros 3492002795"
data: "2006-01-16"
valor_total_venda: 4410.44
▼ produtos_vendidos: Array (3)
  ▼ 0: Object
    nome_produto: "Liquidificador Vermelho violeta 9696215430"
    preco_unitario: 66.39
    quantidade: 6
    valor_total: 398.34
  ▼ 1: Object
    nome_produto: "Mouse Azul manteiga 6990323188"
    preco_unitario: 566
    quantidade: 7
    valor_total: 3962
  ▼ 2: Object
    nome_produto: "Relógio Carmim clássico 2704958138"
    preco_unitario: 8.35
    quantidade: 6
    valor_total: 50.1
```

Figura 28 – Exemplo dos dados nas coleções.

4.6 Definição das Consultas

Nesta seção, são apresentadas consultas desenvolvidas para testar diferentes operações de agregação no MongoDB, como as funções de máximo (MAX), mínimo (MIN), soma (SUM), média (AVG) e contagem (COUNT). O objetivo é avaliar a eficiência dessas operações e identificar possíveis desafios ou limitações que possam surgir durante o processo de desenvolvimento.

A consulta mostrada na Figura 29 utiliza a operação de agregação MAX para identificar os produtos com o maior valor de venda em uma única transação por vendedor, retornando o nome do vendedor e o produto correspondente.

```
1  [
2  {
3    $unwind: "$produtos_vendidos"
4  },
5  {
6    $group: {
7      _id: "$nome_vendedor",
8      maxValorVendido: {
9        $max: "$produtos_vendidos.valor_total"
10     },
11     produtos: { $push: "$produtos_vendidos" }
12   }
13 },
14 {
15   $project: {
16     nome_vendedor: "$_id",
17     produtosMax: {
18       $filter: {
19         input: "$produtos",
20         as: "produto",
21         cond: {
22           $eq: [
23             "$$produto.valor_total",
24             "$maxValorVendido"
25           ]
26         }
27       }
28     }
29   }
30 }
31 ]
32
```

Figura 29 – Consulta para identificar os produtos com maiores valores de venda de um vendedor.

Além do cálculo do valor máximo, outra consulta explorada é a de cálculo do valor mínimo (MIN), ilustrada na Figura 30. Essa consulta permite identificar os produtos com menor valor total de venda em uma única transação por vendedor.

```

1  ▾ [
2  ▾ {
3      $unwind: "$produtos_vendidos"
4  },
5  ▾ {
6  ▾   $group: {
7       _id: "$nome_vendedor",
8   ▾   minValorVendido: {
9       $min: "$produtos_vendidos.valor_total"
10    },
11    produtos: { $push: "$produtos_vendidos" }
12  },
13  },
14  ▾ {
15  ▾   $project: {
16       nome_vendedor: "$_id",
17   ▾   produtosMin: {
18       $filter: {
19         input: "$produtos",
20         as: "produto",
21         cond: {
22   ▾   $eq: [
23       "$$produto.valor_total",
24       "$minValorVendido"
25     ]
26   }
27   }
28   }
29   }
30 }
31 ]
32

```

Figura 30 – Consulta para identificar o cliente com menor valor em compras.

A consulta apresentada na Figura 31 agrupa os dados pelo nome do produto e calcula a quantidade total de unidades vendidas, o valor total das vendas e o preço unitário de cada item. Isso permite analisar quais produtos tiveram o maior número de unidades vendidas, o valor total gerado por cada um e o preço unitário de cada produto.

```

1  ▾ [
2  ▾ {
3      "$unwind": "$produtos_vendidos"
4  },
5  ▾ {
6  ▾   "$group": {
7       "_id": "$produtos_vendidos.nome_produto",
8       "quantidadeVendida": { "$sum": "$produtos_vendidos.quantidade" },
9       "precoUnitario": { "$first": "$produtos_vendidos.preco_unitario" },
10      "valorTotalVenda": { "$sum": "$produtos_vendidos.valor_total" }
11    },
12  },
13  ▾ {
14  ▾   "$project": {
15       "precoUnitario": 1,
16       "quantidadeVendida": 1,
17       "valorTotalVenda": 1
18    }
19  },
20  ▾ {
21      "$sort": { "quantidadeVendida": -1 }
22  }
23 ]
24

```

Figura 31 – Consulta para analisar a quantidade de cada produto vendido e o preço total.

As consultas apresentadas nas Figuras 33 e 32 mostram duas abordagens diferentes para calcular o número de clientes distintos atendidos por cada vendedor, fornecendo insights sobre a distribuição de clientes entre os vendedores

```
1 ▼ [
2 ▼   {
3 ▼     "$group": {
4       "_id": "$nome_vendedor",
5       "totalClientes": { "$sum": 1 }
6     }
7   },
8 ▼   {
9     "$sort": { "totalClientes": -1 }
10  }
11 ]
12
```

Figura 32 – Consulta para contar o número de clientes atendidos por cada vendedor.

```
1 ▼ [
2 ▼   {
3 ▼     "$group": {
4       "_id": "$nome_vendedor",
5       "clientesAtendidos": { "$addToSet": "$nome_cliente" }
6     }
7   },
8 ▼   {
9 ▼     "$project": {
10      "nome_vendedor": "$_id",
11      "totalClientes": { "$size": "$clientesAtendidos" }
12    }
13  },
14 ▼   {
15     "$sort": { "totalClientes": -1 }
16  }
17 ]
18
```

Figura 33 – Diferente maneira de se consultar quantos clientes cada vendedor possui.

A consulta mostrada na Figura 34 calcula o gasto médio de todos os clientes.

```
1 ▼ [
2 ▼   {
3 ▼     "$group": {
4       "_id": null,
5       "gasto_medio": { "$avg": "$valor_total_venda" }
6     }
7   }
8 ]
9
```

Figura 34 – Consulta para calcular o gasto médio de todos os clientes.

4.7 Execução das Consultas

Para a realização desta tarefa, foi criado um Jupyter Notebook, no qual foi executada uma série de passos para garantir a execução das consultas em todas as coleções do banco de dados, além de possibilitar o monitoramento do tempo de execução de cada consulta.

Inicialmente, foi importada a classe `ManipuladorMongoDB` demonstrada na Figura 25 e estabelecemos a conexão com o MongoDB, definindo também o banco de dados no qual as operações serão realizadas.

```
from no_sql_data_base.mongo_functions import ManipuladorMongoDB
✓ [34] 128ms

mongodb = ManipuladorMongoDB()

mongodb.conectar_ao_mongodb(db_host="localhost:27017")
mongodb.definir_banco_de_dados(nome_banco_de_dados="vendas")
✓ [35] < 10 ms

Conectado ao MongoDB em: localhost:27017
Banco de dados definido: vendas
```

Figura 35 – Código utilizado para estabelecer a conexão com o MongoDB.

Após a conexão, listamos todas as coleções presentes no banco de dados para garantir que todas foram criadas corretamente durante o processo de carga dos dados. Além disso, criamos uma lista com os nomes das coleções para utilizá-los posteriormente na execução das consultas.

```
collections = [coll for coll in mongodb.database.list_collection_names()]
collections
✓ [44] < 10 ms

['registro_de_vendas_1000000',
 'registro_de_vendas_1000',
 'registro_de_vendas_100000',
 'registro_de_vendas_10000',
 'registro_de_vendas_10000000']
```

Figura 36 – Código utilizado para listar as coleções no banco de dados.

Em seguida, foi criada uma variável que armazena todas as consultas definidas no Capítulo 6, para que possam ser utilizadas na execução a seguir.

```
> max_pipeline = [...]
> min_pipeline = [...]
> sum_unidade_prod = [...]
> sum_primeira_opc = [...]
> sum_segunda_opc = [...]
> avg_pipeline = [...]

pipelines = [max_pipeline, min_pipeline, sum_unidade_prod, sum_primeira_opc, sum_segunda_opc, avg_pipeline]
[4]
```

Figura 37 – Código utilizado para armazenar todas as consultas de interesse.

A função `executar_consulta_agg`, ilustrada na Figura 38, faz parte da classe `ManipuladorMongoDB` e executa uma agregação em uma coleção do MongoDB. Ela recebe como parâmetro o pipeline de agregação a ser aplicado.

O processo inicia com a medição do tempo de execução, seguido pela execução do método `aggregate` na coleção. Após isso, o tempo total de execução é calculado e exibido em segundos, com 4 casas decimais. O resultado da operação é retornado e pode ser manipulado conforme os requisitos do usuário.

```
def executar_consulta_agg(self, consultas: list) -> None:
    start_time = time.time()

    resultados = collection.aggregate(pipeline)

    end_time = time.time()
    exec_time = end_time - start_time

    print(f"Tempo de execução: {exec_time:.4f} segundos \n")

    return resultados
```

Figura 38 – Método utilizado para executar as consultas e registrar o tempo de execução.

Após estes passos, utilizamos o código abaixo para garantir que todas as consultas sejam executadas em todas as coleções.

```
for nome_colecao in collections:
    mongodb.definir_colecao(nome_colecao)

    for pipeline in pipelines:
        resultados = mongodb.executar_consulta_agg(pipeline)
```

Figura 39 – Trecho de código para executar todas as consultas

5 Análise e Discussão dos Resultados Obtidos

Neste capítulo, são apresentados e discutidos os resultados obtidos a partir dos testes realizados no MongoDB, com ênfase nas operações de agregação e nos desafios enfrentados durante o desenvolvimento do projeto.

Uma das principais vantagens dessa solução é a facilidade na criação do banco de dados por meio de documentos JSON. A flexibilidade do esquema, característica fundamental do MongoDB, permitiu lidar de forma ágil e eficiente com diferentes tipos de dados. Isso eliminou a necessidade de um modelo rígido e pré-definido, proporcionando uma abordagem dinâmica para integração e armazenamento de dados, o que permitiu que o banco estivesse rapidamente em operação.

Adicionalmente, as operações de agregação se mostraram eficientes, especialmente em cenários com grandes volumes de dados. As tabelas abaixo demonstram o desempenho do MongoDB em diferentes volumes de dados e com diversas funções de agregação, evidenciando que, embora o tempo de execução aumente conforme a quantidade de registros cresce, o sistema mantém uma boa performance nas operações de agregação.

Por outro lado, à medida que o volume de dados alcança milhões de registros, o tempo de execução também cresce, algo esperado em qualquer sistema de banco de dados. Esse comportamento reflete a capacidade do MongoDB de lidar com grandes volumes de dados, mas também indica a importância de técnicas de otimização, como a definição estratégica de índices e a adaptação do modelo de dados, para manter a eficiência em cenários de larga escala.

Coleção	Tempo médio de execução em segundos (10 execuções)
registro_de_vendas_1000	0.0114
registro_de_vendas_10000	0.1347
registro_de_vendas_100000	1.5193
registro_de_vendas_1000000	16.3310
registro_de_vendas_10000000	166.2533

Tabela 1 – Produtos com os maiores valores de venda de um vendedor. Consulta definida em [29](#).

Coleção	Tempo médio de execução em segundos (10 execuções)
registro_de_vendas_1000	0.0104
registro_de_vendas_10000	0.1359
registro_de_vendas_100000	1.5871
registro_de_vendas_1000000	16.5551
registro_de_vendas_10000000	164.8178

Tabela 2 – Consulta para identificar o cliente com menor valor em compras. Consulta definida em 30

Coleção	Tempo médio de execução em segundos (10 execuções)
registro_de_vendas_1000	0.0131
registro_de_vendas_10000	0.1338
registro_de_vendas_100000	0.7754
registro_de_vendas_1000000	8.1007
registro_de_vendas_10000000	78.954

Tabela 3 – Consulta para analisar a quantidade de cada produto vendido e o preço total. Consulta definida em 31

Coleção	Tempo médio de execução em segundos (10 execuções)
registro_de_vendas_1000	0.0024
registro_de_vendas_10000	0.0087
registro_de_vendas_100000	0.0553
registro_de_vendas_1000000	0.9297
registro_de_vendas_10000000	9.3021

Tabela 4 – Consulta para contar o número de clientes atendidos por cada vendedor. . Consulta definida em 32

Coleção	Tempo médio de execução em segundos (10 execuções)
registro_de_vendas_1000	0.0024
registro_de_vendas_10000	0.0185
registro_de_vendas_100000	0.1324
registro_de_vendas_1000000	2.1141
registro_de_vendas_10000000	25.9693

Tabela 5 – Segunda forma de se consultar quantos clientes cada vendedor possui. Consulta definida em [33](#)

Coleção	Tempo médio de execução em segundos (10 execuções)
registro_de_vendas_1000	0.0013
registro_de_vendas_10000	0.0158
registro_de_vendas_100000	0.0645
registro_de_vendas_1000000	0.7205
registro_de_vendas_10000000	7.2561

Tabela 6 – Consulta para calcular o gasto médio de todos os clientes. Consulta definida em [34](#)

6 Conclusões e recomendações para trabalhos futuros

Com o aumento do volume de dados não estruturados gerados por diversas fontes, torna-se cada vez mais necessário que as empresas se adaptem para extrair e utilizar todo o potencial dos dados disponíveis. Neste contexto, o MongoDB, um banco de dados NoSQL orientado a objetos, foi testado ao longo deste projeto.

As conclusões apresentadas neste trabalho são baseadas especificamente no escopo deste estudo e não devem ser generalizadas para todos os casos. Antes de tomar decisões sobre como organizar os dados ou qual sistema de gerenciamento de banco de dados adotar, a equipe ou pessoa responsável deve realizar uma análise detalhada das necessidades específicas e das características do ambiente de dados em questão.

Por meio das consultas desenvolvidas e dos testes de desempenho realizados com o MongoDB, foi possível analisar diversas operações de agregação. A eficiência das operações foi avaliada com base no tempo de execução das consultas, que demonstraram resultados satisfatórios mesmo ao lidar com altas cargas de dados.

No entanto, uma das principais desvantagens dessa solução é a complexidade no desenvolvimento de consultas, especialmente quando se trata de operações que exigem lógicas de agregação avançadas. A ausência de um modelo relacional pode tornar o processo de criação de consultas mais desafiador e menos intuitivo para desenvolvedores e equipes de TI acostumados com bancos de dados relacionais. Além disso, para pessoas do negócio que estão familiarizadas com SQL, a transição para o modelo NoSQL pode representar uma curva de aprendizado significativa. Isso pode levar a um aumento no tempo de desenvolvimento e à necessidade de treinamento específico para as equipes envolvidas, o que pode impactar a adoção e o uso contínuo da tecnologia.

Para trabalhos futuros, é recomendado que novas formas de organização dos arquivos dentro do MongoDB sejam testadas, assim como otimizações, como a criação de índices. Além disso, é recomendado testar queries mais complexas, que envolvam regras de negócio específicas, a fim de avaliar o comportamento do MongoDB em cenários mais desafiadores e garantir que o sistema atenda adequadamente às necessidades de dados mais avançadas.

Todos os códigos utilizados ao longo deste projeto estão disponíveis no apêndice, para que possam ser acessados por aqueles que desejarem utilizá-los ou adaptá-los a suas próprias necessidades, facilitando a replicação e a aplicação dos testes realizados.

Referências

- AKSHAY M. MANU, R.; RAJ, R.; KT, S.; K, S. Hybrid approaches for handling imbalanced structured and unstructured data. **Springer Science+Business Media**, Springer Nature, 2024. <<https://doi.org/10.1007/s11042-024-20247-2>>. Citado 2 vezes nas páginas 14 e 15.
- AUTHOR, S. Nosql databases: A step to database scalability in web environment. **International Journal of Web Information Systems**, DBLP, v. 9, n. 1, p. 278–283, 2011. <<https://doi.org/10.1145/2095536.2095583>>. Citado na página 26.
- BARROSO, I. **Banco de dados relacional: fundamentos e aplicações**. 2023. Acessado em: 26 setembro 2023. Disponível em: <<https://www.dio.me/articles/banco-de-dados-relacional-fundamentos-e-aplicacoes>>. Citado 2 vezes nas páginas 3 e 14.
- BERRUTTI, W. W. **Um Estudo Comparativo da Modelagem e Avaliação de Desempenho entre Bancos de Dados Relacional e de Documentos para os Dados Abertos da Câmara dos Deputados do Brasil**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2019. <<https://doi.org/10.183/198694>>. Citado 2 vezes nas páginas 3 e 16.
- BOUAZIZ, S.; NABLI, A.; GARGOURI, F. Design a data warehouse schema from document-oriented databases. In: **23rd International Conference on Knowledge-Based and Intelligent Information Engineering Systems**. [S.l.]: Elsevier, 2019. v. 159, p. 221–230. <<https://doi.org/10.1016/j.procs.2019.09.177>>. Citado na página 29.
- CHEVALIER, M.; MALKI, M. E.; KOPLIKU, A.; TESTE, O.; TOURNIER, R. Document-oriented models for data warehouses: Nosql document-oriented for data warehouses. 2016. <<https://doi.org/10.5220/0005830801420149>>. Citado 8 vezes nas páginas 3, 19, 20, 21, 22, 23, 30 e 33.
- DABBÈCHI, H.; HADDAR, N. Z.; ELGHAZEL, H.; HADDAR, K. Social media data integration: From data lake to nosql data warehouse. In: **International Conference on Intelligent Systems Design and Applications**. Sfax, Tunisia: Springer, 2021. (Lecture Notes in Networks and Systems, v. 1351). <https://doi.org/10.1007/978-3-030-71187-0_64>. Citado na página 29.
- DB-Engines. **DB-Engines Ranking**. 2024. Acesso em: 06 de outubro de 2024. Disponível em: <<https://db-engines.com/en/ranking>>. Citado 3 vezes nas páginas 3, 10 e 11.
- DEHALWAR, K.; SHARMA, S. N. Exploring the distinctions between quantitative and qualitative research methods. **Think India**, v. 27, n. 1, p. 7–15, 2024. <<https://doi.org/10.5281/zenodo.10553000>>. Citado na página 13.
- DEVLIN, B.; MURPHY, P. **An Architecture for a Business and Information System**. [S.l.]: IBM, 1988. <<https://doi.org/10.1147/sj.271.0060>>. Citado 3 vezes nas páginas 3, 16 e 17.

- FOOTE, K. D. **A Brief History of the Data Warehouse**. 2023. Disponível em: <www.dataversity.net/brief-history-data-warehouse/>. Citado na página 9.
- FOWLER, A. **NoSQL For Dummies**. 1st. ed. [S.l.]: John Wiley & Sons, 2015. ISBN 978-1118905623. Citado na página 9.
- INMON, W. H. **Building the Data Warehouse**. [S.l.]: Wiley, 1996. Citado na página 17.
- JOKE2K. **Faker: Python package for generating fake data**. [S.l.], 2023. Acesso em: 29 set. 2024. Disponível em: <<https://faker.readthedocs.io/en/master/>>. Citado na página 33.
- KHAN, W.; KUMAR, T.; ZHANG, C.; RAJ, K.; ROY, A. M.; LUO, B. Sql and nosql database software architecture performance analysis and assessments—a systematic literature review. **Big Data and Cognitive Computing**, Multidisciplinary Digital Publishing Institute, v. 7, n. 2, p. 97, 2023. ISSN 2504-2289. <<https://doi.org/10.3390/bdcc7020097>>. Citado na página 18.
- KIMBALL, R.; ROSS, M. **The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling**. 3rd. ed. [S.l.]: Wiley, 2013. ISBN 978-1118530801. Citado 5 vezes nas páginas 3, 9, 10, 16 e 18.
- LEAVITT, N. Will NoSQL databases live up to their promise? **Computer**, IEEE, v. 43, n. 2, p. 12–14, 2010. ISSN 0018-9162. <<https://doi.org/10.1109/MC.2010.58>>. Citado na página 29.
- MELLO, R. d. S.; DORNELES, C. F.; KADE, A.; BRAGANHOLO, V.; HEUSER, C. A. Dados semi-estruturados. **XV Simpósio Brasileiro de Banco de Dados**, 2000. Citado 2 vezes nas páginas 3 e 15.
- MONGODB. **Databases and Collections**. s.d. Disponível em: <<https://www.mongodb.com/docs/manual/core/databases-and-collections>>. Citado 2 vezes nas páginas 3 e 24.
- MongoDB. **embedded-mongodb**. s.d. Disponível em: <<https://www.mongodb.com/resources/products/fundamentals/embedded-mongodb>>. Citado na página 33.
- MONGODB. **Our Story**. s.d. Disponível em: <<https://www.mongodb.com/company/our-story>>. Citado na página 23.
- MONTEIRO, T. A. F. **Modelagem de banco de dados orientados a documentos com AML**. Recife: Universidade Federal de Pernambuco, 2023. Citado na página 29.
- NALLA, L. N.; REDDY, V. M. The role of nosql databases in scaling e-commerce platforms. **International Journal of Advanced Engineering Technologies and Innovations**, Miami, FL, v. 01, n. 03, p. 262, 2024. Citado na página 10.
- ODITIS, I.; BICEVSKA, Z.; BICEVSKIS, J.; KARNITIS, G. Implementation of nosql-based data warehouses. **BJMC**, DIVI Group and University of Latvia, January 2018. <<https://doi.org/10.22364/bjmc.2018.6.1.04>>. Citado 2 vezes nas páginas 19 e 29.

- OLIVEIRA, B. F. P. de; VICTORINO, M. d. C.; HOLANDA, M. Data warehouse based on nosql: a literature mapping. In: **2021 16th Iberian Conference on Information Systems and Technologies (CISTI)**. [S.l.: s.n.], 2021. p. 1–6. <<https://doi.org/10.23919/CISTI52073.2021.9476293>>. Citado 3 vezes nas páginas 19, 25 e 29.
- PAPIERNIK, M.; DRAKE, M. **An Introduction to Document-Oriented Databases**. DigitalOcean, 2021. Disponível em: <<https://www.digitalocean.com/community/conceptual-articles/an-introduction-to-document-oriented-databases>>. Citado na página 26.
- PRAVEEN, S.; CHANDRA, U. Influence of structured, semi- structured, unstructured data on various data models. **International Journal of Scientific and Engineering Research**, v. 8, p. 67–69, 09 2020. Citado na página 13.
- SARKAR, A. Conceptual level design of semi-structured database system: Graph-semantic based approach. **International Journal of Advanced Computer Science and Applications**, February 2012. <<https://doi.org/10.14569/IJACSA.2011.021019>>. Citado na página 14.
- STATISTICS, A. B. of. **Data**. 2024. Disponível em: <<https://www.abs.gov.au/statistics/understanding-statistics/statistical-terms-and-concepts/data>>. Citado na página 13.
- studyopedia. **How to Create a Collection in MongoDB**. s.d. Disponível em: <<https://studyopedia.com/mongodb/create-a-collection/>>. Citado 2 vezes nas páginas 3 e 24.
- The Economist. **The world's most valuable resource is no longer oil, but data**. The Economist, 2017. Disponível em: <<https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data>>. Citado na página 13.
- YANGUI, R.; NABLI, A.; GARGOURI, F. ETL Based Framework for NoSQL Warehousing. In: **Proceedings of the Conference**. BP 1030, Tunisia: [s.n.], 2017. <https://doi.org/10.1007/978-3-319-65930-5_4>. Citado 2 vezes nas páginas 9 e 26.

A Apêndices

Listing 6.1 – Código para executar os métodos necessários das classes para gerar e carregar os dados

```
import pymongo
from no_sql_data_base.one_collection_document_database import
    CriarBaseDeDadosClientes

def main():
    numero_de_clientes_para_gerar = [1000, 10000, 100000,
        1000000, 10000000]
    gerador_dados = CriarBaseDeDadosClientes()

    gerador_dados.conectar_ao_mongodb(db_host="localhost:27017")
    gerador_dados.definir_banco_de_dados(nome_banco_de_dados="
        vendas")

    for num_clientes in numero_de_clientes_para_gerar:
        gerador_dados.definir_colecao(nome_colecao=f"
            registro_de_vendas_{num_clientes}")
        gerador_dados.gerar_base_clientes(numero_de_clientes=
            num_clientes)

if __name__ == "__main__":
    main()
```

Listing 6.2 – Classe utilizada para gerar os dados de clientes

```
import math
import sys
import time
from faker import Faker
from faker.generator import random
from faker.providers import DynamicProvider
from no_sql_data_base.mongo_functions import ManipuladorMongoDB
```

```
class CriarBaseDeDadosClientes(ManipuladorMongoDB):  
    def __init__(self):  
        super().__init__()   
        self.numero_de_clientes = None  
        self.fake = None  
  
        self.clientes = None  
        self.vendedores = None  
        self.produtos = None  
        self.dados_vendas = []  
  
    def __gerar_dados_clientes(self) -> None:  
        """  
        Gera uma lista de nomes de clientes aleatórios para  
        dados de vendas.  
  
        Este metodo cria uma lista de nomes de clientes únicos  
        com base no número  
        de clientes fornecido pelo usuário.  
        """  
        nomes_gerados = set()  
        telefones_gerados = set()  
  
        self.clientes = []  
        for _ in range(math.ceil(self.numero_de_clientes)):  
            while True:  
                nome_cliente = self.fake.name() + ' ' + str(  
                    random.randint(1, 10000000))  
                if nome_cliente not in nomes_gerados:  
                    nomes_gerados.add(nome_cliente)  
                    break  
  
            while True:  
                telefone_cliente = f'+55 {str(random.randint(1,  
                    99))} {str(random.randint(11111111,  
                    999999999))}'  
                if telefone_cliente not in telefones_gerados:  
                    telefones_gerados.add(telefone_cliente)  
                    break
```

```

        self.clientes.append(
            {
                "nome_cliente": nome_cliente,
                "endereco_cliente": self._gerar_endereco(),
                "telefone_cliente": telefone_cliente,
                "emprego_cliente": self._gerar_emprego(),
                "email_cliente": self.fake.ascii_safe_email
                    (),
            }
        )

def _gerar_endereco(self) -> dict:
    """
    Gera uma lista de endereços aleatórios.

    Este metodo cria uma lista de dicionários, onde cada
    dicionário representa
    um endereço com campos como 'país', 'cidade', 'rua', '
    numero_da_rua' e 'codigo_postal'.
    Para cada entrada, usa-se random.randint(0, 1) para
    determinar se o endereço é incluído:
    - Se o valor for 1, um endereço aleatório é gerado.
    - Se o valor for 0, um dicionário vazio é adicionado
      à lista, indicando nenhum endereço registrado.
    """
    return {
        "pais": self.fake.country(),
        "cidade": self.fake.city(),
        "rua": self.fake.street_name(),
        "numero_da_morada": self.fake.random_number(
            digits=4),
        "codigo_postal": self.fake.postcode()
    } if random.randint(0, 1) == 1 else None

def _gerar_emprego(self) -> str:
    """
    Metodo para criar uma lista contendo empregos aleatórios
    .

```

```

    """
    return self.fake.job() if random.randint(0, 1) == 1 else
        None

def _gerar_dados_vendedores(self) -> None:
    """
    Metodo para criar uma lista contendo nomes de empresas
        aleatórios.

    Isso será usado para criar vendas, então criaremos para
        x clientes
    x * 0,01 número de vendedores.

    Desta forma, podemos criar uma simulação mais realista
        em que cada empresa vende x produtos.
    """
    if self.numero_de_clientes > 1000000:
        quantidade_vendedores = 10000
    else:
        quantidade_vendedores = self.numero_de_clientes *
            0.01

    self.vendedores = [f"{self.fake.unique.company()} {self.
        fake.unique.random_number(digits=10)}" for i in range
        (math.ceil(quantidade_vendedores))]

def _gerar_nomes_produtos(self) -> None:
    """
    Metodo para criar uma lista contendo valores aleatórios
        para gerar produtos e preços aleatórios.

    Aqui estamos gerando os nomes como:
        - Nome do produto entre um dos descritos na lista a
            baixo, uma cor aleatória e um número aleatório de
            10
            digitos apenas para garantir que nenhum produto ira
            se repetir.
    """
    produtos_comuns = [

```

```

        "Fones de Ouvido",
        "Smartphone",
        "Laptop",
        "Tablet",
        "Camera",
        "Alto-falante",
        "Relógio",
        "Mochila",
        "Carregador",
        "Liquidificador",
        "VideoGame",
        "Mouse",
        "Monitor",
        "Teclado",
        "impressora",
        "Gabinete"
    ]

    nomes_produtos = set()

    if self.numero_de_clientes < 1000:
        quantidade_produtos = 10
    elif 1000 <= self.numero_de_clientes < 10000:
        quantidade_produtos = 100
    elif 10000 <= self.numero_de_clientes < 100000:
        quantidade_produtos = 500
    elif 100000 <= self.numero_de_clientes < 1000000:
        quantidade_produtos = 1000
    else:
        quantidade_produtos = 2000

    while len(nomes_produtos) < quantidade_produtos:
        nomes_produtos.add(
            f"{random.choice(produtos_comuns)} {self.fake.
                color_name()} {self.fake.unique.random_number(
                    digits=10)}"
        )

    # nomes_produtos_provider = DynamicProvider(

```

```

#         provider_name="produtos",
#         elements=list(nomes_produtos),
#     )
#
# self.fake.add_provider(nomes_produtos_provider)
nomes_produtos = list(nomes_produtos)

self.produtos = [
    (
        #self.fake.unique.random_number(digits=10),
        nomes_produtos[_],
        round(float(self.fake.pricetag().replace('R$', ' ',
            ').replace(', ', '')), 2),
    )
    for _ in range(math.ceil(quantidade_produtos))
]

def _gerar_vendas_produtos(self) -> dict:
    """
    Gera uma lista de vendas simuladas de produtos para vá-
    rios clientes.

    Este metodo gera dados de vendas selecionando produtos e
    quantidades aleatórios para cada cliente
    com base em uma lista predefinida de produtos. Para cada
    venda, um número aleatório de produtos é selecionado
    ,
    e a quantidade e o valor total de cada produto são
    calculados. O resultado é uma lista de
    dicionários, onde cada dicionário representa uma venda
    com detalhes do produto.

    Retorna:
        list: Uma lista onde cada elemento é uma lista de
        dicionários. Cada dicionário contém:
        - "id_produto" (int): O ID do produto.
        - "quantidade" (int): A quantidade do produto
        comprada pelo cliente.
        - "valor_total" (float): O custo total da
    """

```

quantidade comprada do produto.

Exemplo:

```
[
    [
        {"quantidade": 5, "valor_total": 49.95},
        {"quantidade": 2, "valor_total": 19.98}
    ],
    [
        {"quantidade": 1, "valor_total": 9.99}
    ]
]
"""
total = 0
produtos_cliente = []
registro_cliente = []

quantidade_itens_comprados = self.fake.random_int(min=1,
max=10)

for q in range(quantidade_itens_comprados):
    unidades_compradas = self.fake.random_int(min=1, max
=10)
    item = self.fake.random_int(min=0, max=len(self.
produtos) - 1)

    while item in registro_cliente:
        item = self.fake.random_int(min=0, max=len(self.
produtos) - 1)

    valor_total = float(f"{self.produtos[item][1] *
unidades_compradas:.2f}")

    produtos_cliente.append(
        {
            #"id_produto": self.produtos[item][0],
            "nome_produto": self.produtos[item][0],
            "preco_unitario": self.produtos[item][1],
            "quantidade": unidades_compradas,
```

```
        "valor_total": valor_total,
    }
)

total += valor_total
registro_cliente.append(item)

return {
    "data": self.fake.date(),
    "valor_total_venda": round(total, 2),
    "produtos_vendidos": produtos_cliente,
}

def gerar_vendas(self):
    todos_registros = []

    for _ in range(self.numero_de_clientes):
        dicionario_base = {}

        item = self.fake.random_int(min=0, max=len(self.
            clientes) - 1)
        dicionario_base.update(self.clientes[item])

        item = self.fake.random_int(min=0, max=len(self.
            vendedores) - 1)
        dicionario_base.update({"nome_vendedor": self.
            vendedores[item]})

        dicionario_base.update(self._gerar_vendas_produtos()
            )

        todos_registros.append(dicionario_base)

    self.inserir_muitos_na_colecao(payload=todos_registros)

def gerar_base_clientes(self, numero_de_clientes: int):
    """
    Metodo para executar todos os métodos para gerar dados
    aleatórios.
```

```

    """

    tempo_inicio = time.time()

    self.limpar_execucao_anterior()

    self.numero_de_clientes = numero_de_clientes

    self._gerar_dados_clientes()
    self._gerar_dados_vendedores()
    self._gerar_nomes_produtos()

    print(f"{len(self.clientes)} clientes criados.")
    print(f"{len(self.vendedores)} vendedores criados.")
    print(f"{len(self.produtos)} produtos diferentes criados
        .")

    self.gerar_vendas()

    tempo_fim = time.time()
    tempo_total = tempo_fim - tempo_inicio

    print(f"Tempo total de execução: {tempo_total:.2f}
        segundos. \n")

    def limpar_execucao_anterior(self):
        self.fake = Faker(['pt_BR']) # Alterado para português brasileiro
        self.clientes = None
        self.vendedores = None
        self.dados_vendas = []

```

Listing 6.3 – Classe utilizada para interagir com MongoDB

```

import pymongo
import time

class ManipuladorMongoDB:
    def __init__(self):

```

```
self.client = None
self.database = None
self.collection = None

def conectar_ao_mongodb(self, db_host: str) -> None:
    self.client = pymongo.MongoClient(db_host)
    print("Conectado ao MongoDB em: ", db_host)

def definir_banco_de_dados(self, nome_banco_de_dados: str)
-> None:
    if not self.client:
        raise ConnectionError("Cliente MongoDB não está
conectado. Por favor, conecte primeiro.")

    self.database = self.client[nome_banco_de_dados]
    print("Banco de dados definido: ", nome_banco_de_dados)

def definir_colecao(self, nome_colecao: str) -> None:
    if not self.client:
        raise ConnectionError("Cliente MongoDB não está
conectado. Por favor, conecte primeiro.")

    self.collection = self.database[nome_colecao]
    print("Coleção definida: ", nome_colecao)

def inserir_um_na_colecao(self, payload: dict) -> None:
    resultado_insercao = self.collection.insert_one(payload)

def inserir_muitos_na_colecao(self, payload: list) -> None:
    resultado_insercao = self.collection.insert_many(payload
)

def executar_consulta_agg(self, consulta: list) -> None:
    start_time = time.time()

    #print(f"Executando consulta: {consulta} \n")

    resultados = self.collection.aggregate(consulta)
```

```
end_time = time.time()
exec_time = end_time - start_time

print(f"Tempo de execução: {exec_time:.4f} segundos \n")

return resultados
```