UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

BACHARELADO EM ENGENHARIA DE SOFTWARE

VITÓRIA MOREIRA DE SOUZA

ANÁLISE DE DESEMPENHO ENTRE BANCOS DE DADOS EMPREGANDO A FERRAMENTA YAHOO CLOUD SERVING BENCHMARKING TOOL EM CONSULTAS GEOESPACIAIS

QUIXADÁ

2023

VITÓRIA MOREIRA DE SOUZA

ANÁLISE DE DESEMPENHO ENTRE BANCOS DE DADOS EMPREGANDO A FERRAMENTA YAHOO CLOUD SERVING BENCHMARKING TOOL EM CONSULTAS GEOESPACIAIS

Trabalho de Conclusão de Curso apresentado ao Bacharelado em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientadora: Profa. Dra. Lívia Almada Cruz

QUIXADÁ

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S236a Souza, Vitória Moreira de.
Análise de desempenho entre bancos de dados empregando a ferramenta Yahoo Cloud Serving Benchmarking Tool em consultas geoespaciais / Vitória Moreira de Souza. – 2023.
56 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2023.
Orientação: Prof. Dr. Lívia Almada Cruz.

1. Consultas Geoespaciais. 2. Banco de Dados Relacionais. 3. Banco de Dados Não-Relacionais. 4. Ferramentas de Benchmarking. 5. Yahoo Cloud Serving Benchmark. I. Título.

CDD 005.1

VITÓRIA MOREIRA DE SOUZA

ANÁLISE DE DESEMPENHO ENTRE BANCOS DE DADOS EMPREGANDO A FERRAMENTA YAHOO CLOUD SERVING BENCHMARKING TOOL EM CONSULTAS GEOESPACIAIS

Trabalho de Conclusão de Curso apresentado ao Bacharelado em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em:__/__/____.

BANCA EXAMINADORA

Profa. Dra. Lívia Almada Cruz (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Regis Pires Magalhães
Universidade Federal do Ceará (UFC)

Prof. Dr. Victor Aguiar Evangelista de Farias
Universidade Federal do Ceará (UFC)

Aos meus amigos e futuros colegas na área de tecnologia que me acompanham desde a infância e sempre me deram forças ao continuar a investir em mim como pessoa. À minha família, por sempre me apoiar em minhas decisões.

AGRADECIMENTOS

Agradeço primeiramente a minha resiliência em sempre continuar, não importando o obstáculo. Em segundo, a minha orientadora, a professora Lívia Almada Cruz, docente da Universidade Federal do Ceará, sem ela a existência desse trabalho nem seria possível.

Agradeço também a minha mãe, avó, irmãos e tias, que nos momentos de minha ausência dedicados ao estudo superior, sempre me fizeram entender que o futuro é feito a partir da constante dedicação no presente!

Por fim, agradeço a todos os professores por quem já tive o prazer de ser ensinada, por me proporcionar o conhecimento não apenas racional, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender.

"Não tenhamos pressa, mas não percamos tempo."

(José Saramago)

RESUMO

Quantidades massivas de dados geoespaciais são geradas atualmente pelas mais diversas aplicações, as quais são acessíveis a uma grande quantidade de usuários. Junto ao volume de dados, cresceu também a necessidade por administração eficiente deles. Como a necessidade de acompanhar esses dados cada vez mais em tempo real, surgiram os banco de dados não-relacionais dos mais diversos tipos (chave-valor, baseado em grafos e baseado em documento são alguns exemplos), agora concorrentes dos bancos relacionais. Este trabalho efetua uma análise comparativa de desempenho dos bancos de dados em consultas geoespaciais, análise essa auxiliada pela ferramenta de avaliação de desempenho *Yahoo Cloud Serving Benchmark* (YCSB).

Palavras-chave: Dados geoespaciais; Banco de dados; Banco de dados relacional; Banco de dados não-relacionais; Ferramenta de benchmarking.

ABSTRACT

Massive amounts of geospatial data are currently generated by various applications, which are accessible to a large number of users. Alongside the data volume, the need for efficient management of these data has also grown. With the increasing demand to track this data in real-time, various types of non-relational databases (such as key-value, graph-based, and document-based) have emerged as competitors to relational databases. This work will perform a comparative performance analysis of databases in geospatial queries, aided by the benchmarking tool Yahoo Cloud Serving Benchmark (YCSB).

Keywords: Geospatial datas; Database; Relational database; Non-relational database; Benchmarking tool.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

CRUD	*Create, Read, Update and Delete*/Criação, Consulta, Atualização e Apagamento de dados
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
GIS	sistema de informação geográfica
IoT	*Internet of Things*/Internet das Coisas
NoSQL	*Not Only SQL*/Não somente SQL
POI	*Point of Interest*/Ponto de Interesse
SQL	*Standard Query Language*/Linguagem de Consulta Estruturada
XHR	XMLHttpRequest
YCSB	*Yahoo Cloud Serving Benchmark*

SUMÁRIO

1 INTRODUÇÃO

O acesso à tecnologia vem se difundindo e aumentando entre as populações ao longo dos anos, o efeito disso é a produção massiva de aplicações e consequentemente a geração abundante de dados dos mais variados tipos, dentre eles destacam-se os dados categorizados como geoespaciais. As plataformas mais conhecidas de geração desses dados são sistemas de GPS, como Waze ou Google Maps, sistemas para transporte coletivo, como Uber e 99, e até mesmo em sistemas de encomenda e entrega de alimentos como IFood. Segundo IBGE (2017), "um dado geográfico ou geoespacial, ou georreferenciado é um dado espacial onde a dimensão espacial está associada à sua localização na superfície terrestre, em determinado instante ou período de tempo".

Esses dados são gerados por aplicações em nuvem, em computadores de mesa, celulares e até em objetos cotidianos em versões digitais conectados à internet *Internet of Things*/Internet das Coisas (IoT), eles podem ser gerados de maneira ativa, com escolha manual do usuário, ou passiva, que necessita apenas do consentimento do utilizador do sistema. Para Santos *et al.* (2015) os numerosos dados gerados precisam ser guardados e gerenciados de maneira eficiente e eficaz, garantindo a sua segurança, integridade e disponibilidade, visto que aplicações que tratam dados geográficos são muito rentáveis e úteis muito socialmente (não atoa possuem empresas milionárias operando-as).

Para garantir o gerenciamento eficiente e eficaz dessas informações existem os bancos de dados. O modelo relacional é um dos mais aceitos para armazenamento e gerenciamento de dados, criado como uma forma de padronizar a estrutura de consultas e representação dos dados, esse modelo se apoia no uso de tabelas e na linguagem de consulta conhecida como *Standard Query Language*/Linguagem de Consulta Estruturada (SQL) (ORACLE, 2020a). Por muitos anos o modelo relacional dominou o mundo da tecnologia, porém novos modelos com características únicas surgiram, com habilidade de armazenarem estruturas de dados diferentes entre si nos mesmos conjuntos, possuindo ainda uma alta habilidade de leitura e escrita em simultâneo dos mesmos dados e o que é chamado consistência eventual, se tornando rapidamente popular, conhecidos como *Not Only SQL*/Não somente SQL NoSQL ou bancos de dados não relacionais.

Existem muitos tipos documentados e reconhecidos de modelos de banco de dados NoSQL (entre eles estão chave-valor, baseado em documento e baseado em grafos). Neles o esquema de armazenamento não precisa ser definido previamente, mas pode ser moldado

à medida que o desenvolvimento da aplicação avança, o que pode aumentar o desempenho em detrimento da consistência (ORACLE, 2020b). Ao se observar um crescente aumento de armazenamento de dados geoespaciais e a necessidade de banco de dados capazes de lidar rapidamente com eles em tempo real, ou quase real, a adoção do modelo de banco de dados não relacional se tornou cada vez mais comum e atrativa nas aplicações que utilizam esses dados.

No entanto, a escolha de qual sistema de banco de dados NoSQL usar pode variar consoante a proposta da aplicação, conhecer as características e os fundamentos de cada um pode não ser suficiente para seleção mais adequada. Davoudian *et al.* (2018) relata que uma das principais dificuldades de *designers* e arquitetos de aplicativos que desejam utilizar este modelo banco de dados em suas empresas habita na grande lista de bancos, com mais de 200 bancos de dados NoSQLs diferentes existentes, os quais são adequados para cenários e aplicações específicas.

Existe uma gama de trabalhos que comparam o desempenho de sistemas de bancos de dados, relacionais ou não, como os trabalhos de Pereira *et al.* (2018), Santos *et al.* (2015), Laksono (2018). Porém, a maioria deles não costuma focar em consultas e dados geográficos e, dentre eles, poucos usam uma gama variada de tipos de bancos não-relacionais em suas comparações.

1.1 Objetivos

Este trabalho se apoia na ideia de guiar o desenvolvedor ou arquiteto de *software* na escolha de sistema de banco de dados em aplicações que utilizam dados geoespaciais com Pontos de Interesse (POI, do inglês *Point of Interest*). O objetivo principal desse trabalho é comparar bancos de dados, fazendo um aparato de suas características, funcionalidades e desempenho ao tratar dados geoespaciais. Seus objetivos específicos são:

- Encontrar entre os bancos de dados comparados aquele que é mais adequado para inserção de dados geoespaciais;
- Encontrar entre os bancos de dados comparados aquele que é mais adequado para consultas geoespaciais de intervalo;
- Encontrar entre os bancos de dados comparados aquele que é mais adequado para consultas geoespaciais de k-vizinho mais próximo;
- Encontrar entre os bancos de dados comparados aquele que é mais adequado para consultas geoespaciais de região.

Destarte, para atingir esses objetivos foi escolhida uma ferramenta de avaliação de desempenho de bancos de dados conhecida como *Yahoo Cloud Serving Benchmark* YCSB[1], que possui um sistema próprio de métricas para análise de desempenho e consultas próprias, contudo para este trabalho ele será adaptado especialmente para inserir e consultas dados geoespaciais.

Para o conjunto de dados em que essas consultas atuarão foi escolhido um conjunto de dados geográficos reais *US Accidents (2016 – 2021)* [2] (recolhido de uma plataforma que possuí um acervo de conjunto de dados para uso gratuito conhecida como *Kaggle*), que representam a geolocalização de acidentes de carro nos Estados Unidos no período de 2016 a 2021.

Por fim, foram escolhidos quatro tipos de banco de dados que suportam consultas geoespaciais, um relacional e três não-relacionais (chave-valor, baseado em documentos e baseado em grafos).

1.2 Estrutura do Trabalho

O presente documento está estruturado de maneira que segue. No Capítulo 2 são apresentados e fundamentados vários dos conceitos que serão usados ao longo do trabalho. No Capítulo 3 estão presentes os trabalhos que alicerçaram e estão intrinsecamente ligados a proposta desse documento, bem como a contribuição deste trabalho sobre eles. No Capítulo 4 são apontados os procedimentos metodológicos usados para atingir o objetivo. No Capítulo 5 estão listados os resultados obtidos com os experimentos. Por fim, no Capítulo 6 encontra-se a conclusão.

[1] https://github.com/brianfrankcooper/YCSB
[2] https://www.kaggle.com/datasets/sobhanmoosavi/us-accidents

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo trata da fundamentação teórica necessária para a melhor compreensão do trabalho contido neste documento. Nele são apresentados os modelos de bancos de dados e como eles costumam armazenar dados comuns e geoespaciais, bem como as explicações de como são aplicadas neles as consultas geoespaciais abordadas no trabalho.

2.1 Bancos de Dados

É imprescindível para a compreensão de banco de dados não relacionais a introdução ao conceito de bancos de dados relacionais, visto que o primeiro surgiu como uma alternativa para o segundo na manipulação de grandes dados. A principal causa da origem de bancos de dados relacionais foi a necessidade de padronização no armazenamento de dados, resolvido nesse banco pelo conceito de tabelas (com linhas e colunas) cujos dados se relacionam, pelos pilares Atomicidade, Consistência, Isolamento e Durabilidade (ACID) que alicerçam a sua definição e, mais tarde, por SQL, linguagem que auxilia nas consultas no banco (ORACLE, 2020a).

Conforme Davoudian *et al.* (2018), após a utilização massiva de bancos com linguagem SQL ao longo dos anos, devido aos avanços tecnológicos, maior acesso à tecnologia e maior uso de aplicativos por muitas pessoas, seja devido a redes sociais, sistemas com Internet das coisas (IOTs, do inglês *Internet of Things*) ou outras aplicações para dispositivos móveis; os modelos de dados no desenvolvimento de sistemas se diversificaram e passaram a ser classificados como estruturados, semi-estruturados ou mesmo não-estruturados. Consequentemente, também cresceu a quantidade de dados que precisam de armazenamento e a necessidade de melhor gerenciamento desses dados. Graças a essa demanda, o uso do modelo NoSQL para armazenamento de dados foi amplamente difundido, tornando-se uma alternativa ao modelo relacional vigente.

Segundo Vera-Olivera *et al.* (2021), a ampla utilização de banco de dados NoSQL se deve porque o banco de dados relacional tradicional é menos equipado para gerenciar grandes volumes de dados em rápido crescimento com estruturas muito complexas e visa a consistência quando consulta e escrita estão ocorrendo no mesmo dado, em simultâneo. Já os bancos de dados NoSQL priorizam a disponibilidade sobre a consistência e possuem flexibilidade de estruturação de dados dentro de seus tipos, categorizando bancos nesse modelo. Com a popularidade de bancos de dados não relacionais, os antigos problemas de formas variadas de armazenamento

voltaram. A seguir são esplanadas todos os modelos, ou tipos, reconhecidos dos bancos de dados não relacionais até a data de publicação deste trabalho, são eles, chave-valor, modelo orientado a coluna, modelo orientado a documento e modelo orientado a grafos, bem como o modelo relacional tradicional.

2.1.1 Modelo Relacional

O modelo de banco de dados relacional é um dos mais difundidos e usados entre os profissionais da área de tecnologia para armazenamento de dados consistentemente. No modelo relacional a principal construção para representação dos dados é a relação, uma tabela com linhas não ordenadas e colunas. A Figura 1 representa bem esse modelo. Os dados guardados nesse modelo podem ser espalhados por diversas tabelas e ligados por meio de uma chave primária ou estrangeira. No contexto da figura, os dados são guardados em apenas uma tabela com uma chave primária "ID". É possível notar que são guardados dados de tipos diferentes, sejam eles numerais, caracteres ou de temporais.

Figura 1 – Representação gráfica de dado armazenado em bancos de dados relacional

ID	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	Distance(mi)	Description
A-1	3	2016-02-08 05:46:00	2016-02-08 11:00:00	39.865147	-84.058723	0.01	Right lane blocked due to accident on I-70 Eastbound at Exit 41 OH-235 State Route 4.
A-2	2	2016-02-08 06:07:59	2016-02-08 06:37:59	39.928059	-82.831184	0.01	Accident on Brice Rd at Tussing Rd. Expect delays.
A-3	2	2016-02-08 06:49:27	2016-02-08 07:19:27	39.063148	-84.032608	0.01	Accident on OH-32 State Route 32 Westbound at Dela Palma Rd. Expect delays.
A-4	3	2016-02-08 07:23:34	2016-02-08 07:53:34	39.747753	-84.2055819	0.01	Accident on I-75 Southbound at Exits 52 52B US-35. Expect delays.

Fonte: elaborado pela autora (2023).

O modelo relacional costuma contar também com uma linguagem estruturada própria (SQL e suas variações). Para este trabalho foi escolhido o PostgreSQL v.15.3[1] como sistema de gerenciamento de banco de dados desta categoria.

2.1.2 Modelo Chave-Valor

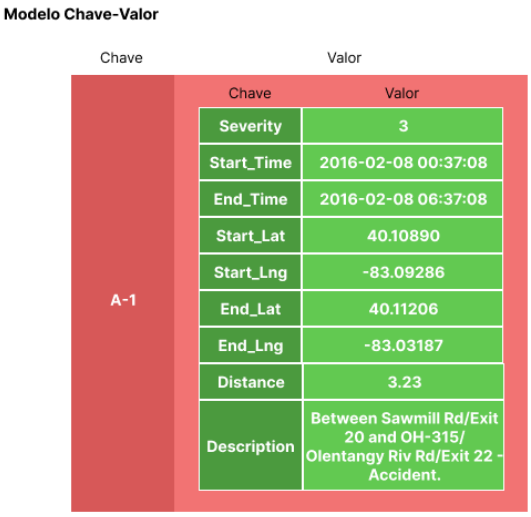
O modelo de banco de dados NoSQL chave-valor talvez seja o mais conhecido de seu gênero, esse modelo se baseia no conceito simples de organização de armazenamento de dados por meio de uma "chave" que representa o identificador do "valor" (dado armazenado).

[1] https://www.postgresql.org/

Segundo Bugiotti *et al.* (2014), no armazenamento chave-valor a chave controla a distribuição de dos dados, os quais são divididos em pares de chave-valor sem esquemas prévios definidos, possuindo operações de acesso de dados de pares individuais ou de grupos selecionados.

O banco de dados escolhido pode afetar a representação chave-valor, alguns bancos aceitam uma estrutura complexa como “chave”, bem como uma estrutura complexa como “valor”. Este trabalho recorreu ao banco de dados Redis v.3.2.100[2], cuja representação gráfica de armazenamento de dados pode ser observado na Figura 2. Nesta figura a chave principal, “A-1”, é simples e representa o identificador da estrutura de dados complexa que possui pares de chaves e valores, também é possível que esse conjunto de valores esteja dentro de outra chave que represente o conjunto de dados referente a está estrutura, como a chave "*Severity*", por exemplo.

Figura 2 – Representação gráfica de dado armazenado em bancos NoSQL chave-valor



Fonte: elaborado pela autora (2023).

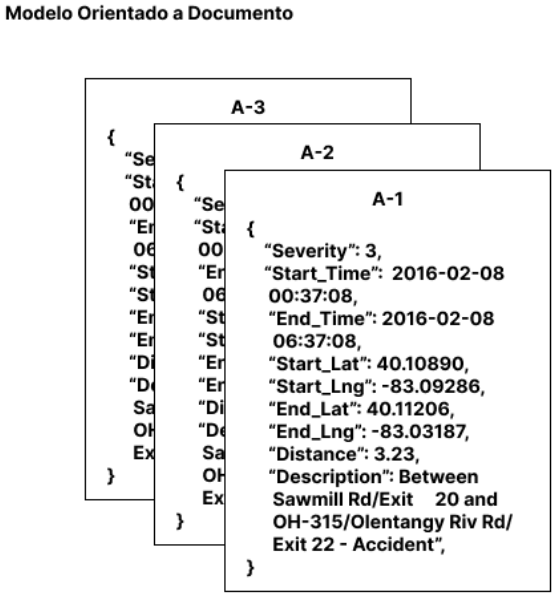
2.1.3 Modelo Orientado a Documentos

O modelo orientado a documentos segue uma estrutura de armazenamento que, como o nome sugere, se guia por documentos. Cada documento possui um nome e um conjunto de atributos comumente representados por chave-valor, um conjunto de documentos pode ser chamado coleção, um banco de dados costuma ter várias coleções, os documentos na coleção podem ou não seguir uma mesma estrutura. Alguns bancos mais complexos aceitam que documentos possuam coleções.

[2] https://redis.io

Conforme Amirian *et al.* (2014), uma busca em coleções que possuem documentos com a mesma estrutura costuma ser mais eficiente do que buscas em banco de dados chave-valor com estruturas semelhantes de armazenamento entre os dados. Na Figura 3 está representado o armazenamento dos dados usado neste documento em um banco de dados não relacional baseado em documentos chamado MongoDB v.6.0.6 [3]. Nesta figura, o nome de cada documento representa o identificador da estrutura de dados contido nele, está estrutura é representada por pares de chaves e valores estruturados da mesma maneira em cada documento, a coleção que contém estes documentos pode ser chamada "acidentes".

Figura 3 – Representação gráfica de dado armazenado em bancos NoSQL baseado em documentos



Fonte: elaborado pela autora (2023).

2.1.4 Modelo Orientado a Grafos

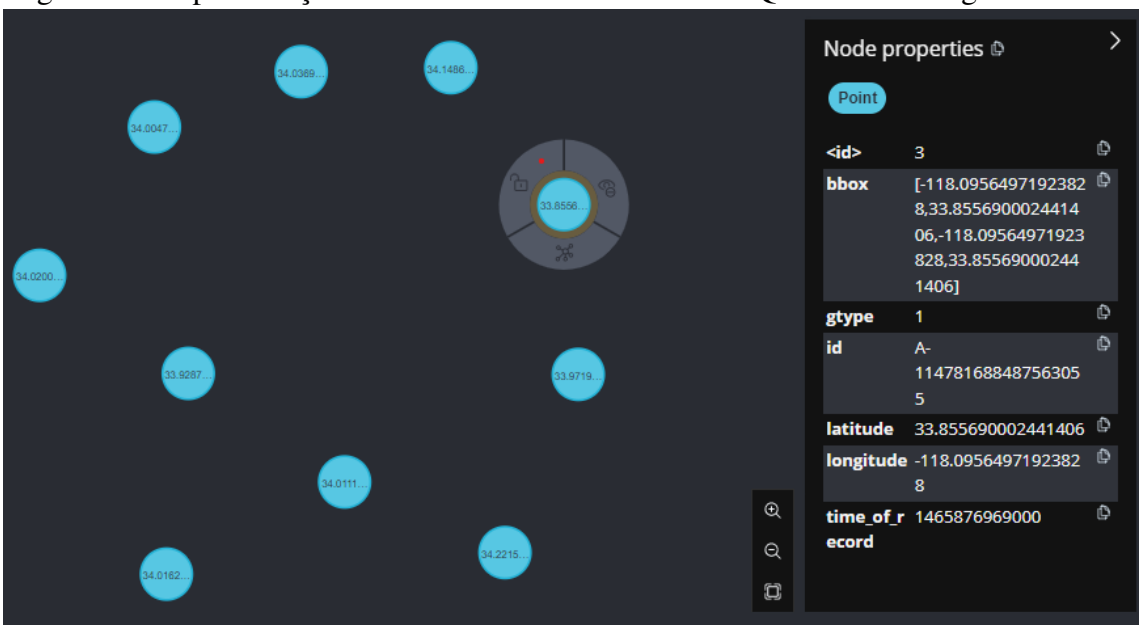
Os bancos de dados orientados a grafos possuem suas entidades representadas por vértices e os relacionamentos entre eles são representados por arestas. Esse é um modelo construído com intuito de ser o mais visual possível e seu alicerce está no relacionamento entre os vértices nas aplicações comuns. Grandes empresas como Twitter, Facebook, Google e LinkedIn tem seus dados modelados naturalmente usando grafos (GUDIVADA *et al.*, 2014).

A Figura 4 demonstra a representação dos dados em banco de dados NoSQL de

[3] https://www.mongodb.com/pt-br

modelo baseado escolhido, o Neo4J v.4.4.3 [4]. Na Figura 4 estão representados pontos, neles estão contidos as informações de referência geográfica. Este trabalho visa focar no uso dos modelos de bancos de dados voltados para dados geoespaciais e os relacionamentos que existem entre os pontos geográficos apenas podem ser abstraídos quando sobrepostos à terra, um exemplo de relacionamento entre eles seria a distância entre cada nó, porém para simplificar a inserção no banco, apenas os vértices foram deixados.

Figura 4 – Representação do armazenado em bancos NoSQL baseado em grafos



Fonte: elaborado pela autora (2023).

2.2 Armazenamento de Dados e Consultas Geoespaciais

Aplicações que utilizam dados espaciais (geoespaciais) possuem o ponto em comum de que em seu conjunto de dados existirá pelo menos uma indicação de latitude, longitude e, por muitas vezes, de tempo em que essa informação foi armazenada, assim é possível ter a ciência de onde um objeto estava em determinado tempo. Isto pode ser muito útil e usado em vários setores da sociedade. Por conseguinte, o armazenamento de dados de localização vem aumentando drasticamente e o gerenciamento de grandes conjuntos de dados costuma ser altamente custoso, principalmente no que tange a grande conjunto de dados geoespaciais.

Para o gerenciamento dos dados coletados, transformando-os em informações relevantes para ou sobre os objetos observados, existem conjuntos diversificados de consultas. Os tipos de consultas mais utilizadas pela aplicação dependerá notavelmente de qual a finalidade

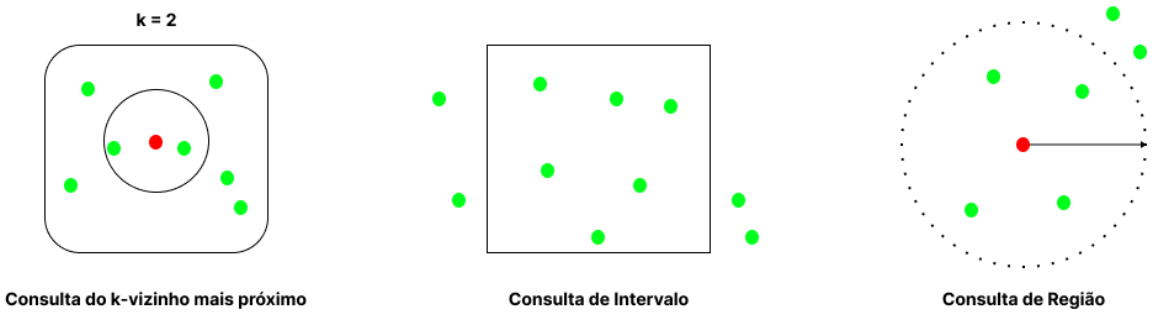
[4] https://neo4j.com/

da aplicação. Em aplicações geoespaciais, segundo Santos *et al.* (2015), essas consultas podem ser categorizadas como ponto de interesses próximos, visão de mapa, roteamento urbano e rastreamento de posição. Para Domingues *et al.* (2020), "A detecção de pontos de interesse é questão fundamental no estudo da mobilidade humana e sua evolução com o tempo. Pontos de interesse em dados geoespaciais podem representar residências, lojas, escritórios de trabalho e até semáforos e regiões de congestionamento frequente".

Neste trabalho, são consideradas algumas consultas de pontos de interesse próximo que podem ser encontradas no trabalho de Niyizamwiyitira e Lundberg (2017), suas representações podem ser observadas na Figura 5, sendo descritas abaixo como:

- Consulta do K-vizinho Mais Próximo: Essa consulta tem o objetivo de encontrar os "k" pontos de interesse que estão mais próximos a um ponto delimitado. Nela é fornecida as coordenadas do ponto e a quantidade de pontos a serem retornados.
- Consulta de Intervalo: Essa consulta encontra pontos de interesse em determinada região delimitada por um polígono escolhido. Nela é fornecida as coordenadas de cada vértice do polígono em ordem para retornar os pontos dentro dele.
- Consulta de Região: Essa consulta é usada para encontrar pontos que estão a uma determinada distância do ponto especificado, é bem representada pelo raio de um círculo. Nela é fornecido as coordenadas do ponto e a distância para se retornar os pontos dentro dessa distância.

Figura 5 – Representação das consultas de POI



Fonte: elaborado pela autora (2023).

2.3 *Benchmarking*: Medição e Análise de Desempenho sobre Consultas

Existem algumas ferramentas para medição de desempenho de consultas em banco de dados, elas podem ser chamadas ferramentas de *benchmarking* e auxiliam na escolha de

sistemas sobre determinados cenários, visto que podem avaliar esses sistemas sobre diversos critérios, fazendo comparações no que tange a desempenho de hardware. Geralmente, quando de código aberto, são escritas em uma linguagem de fácil modificação e possuem cargas de trabalho, consultas e sistemas que podem ser acoplados já pré-definidos, todas essas configurações podem ser alteradas para gerar versões diferentes de uma mesma ferramenta.

Entre algumas das ferramentas de *benchmarking* existentes está o BenchBase[5], utilizado para medir desempenho de bancos relacionais, infelizmente esta ferramenta não possui naturalmente metrificação de consultas geográficas ou análise de bancos não relacionais. Apesar de já existirem ferramentas que medem o desempenho de consultas geográficas como o Jackpine, de acordo com Ray *et al.* (2011), ainda não existe uma ferramenta de benchmarking espacial padrão que seja amplamente utilizada, principalmente sobre consultas e bancos NoSQL.

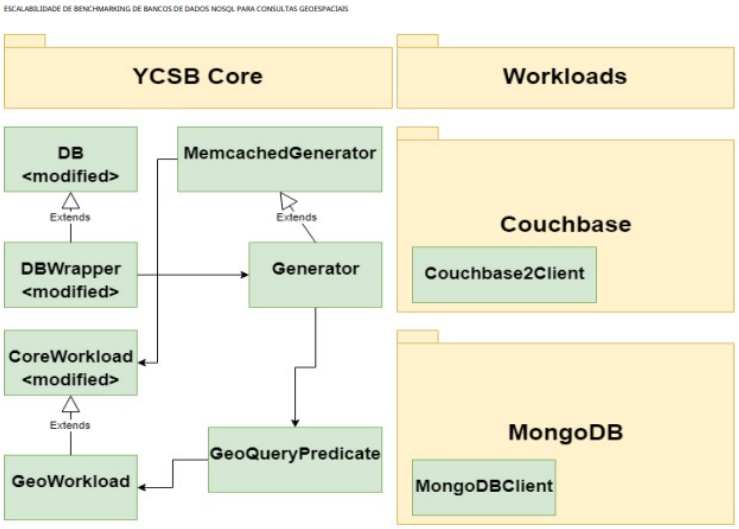
Uma das ferramentas de *benchmarking* existentes para banco de dados não relacionais é o The Yahoo Cloud Serving Benchmark (YCSB), ele faz a medição de tempo de execução, vazão e latência (mínima, máxima, média, 95 percentil e 99 percentil) sobre as consultas de leitura de um registro, leitura de um conjunto de registros, atualização de um registro, inserção de um registro ou apagamento de um registro, dependendo da configuração da carga de trabalho (BENCHANT, 2021). Seu código é escrito em Java, linguagem orientada a objetos, portanto, tanto suas métricas, consultas e cargas de trabalho podem ser estendidas por outras classes e modificadas.

Para Kanwar (2019) é possível implementar uma arquitetura sobre YCSB para suporte geoespacial de consultas específicas conforme mostrado na Figura 6. Nesta figura está representado um diagrama de classes usado na modificação do projeto nativo da ferramenta de *benchmarking* para adaptá-lo para consultas geoespaciais nos bancos Couchbase e MongoDB.

Nela é possível observar que na pasta “YCSB Core” (responsável por guardar o código-fonte da ferramenta) modificações foram feitas nas classes abstratas “BD” (interfaces das consultas) e “DBWrapper” (implementação das consultas), bem como em “CoreWorkload” (responsável pela execução das cargas de trabalho), também é notável o acréscimo de classes como “GeoWorkload” e “GeoQueryPreficate” (classes para o auxílio de cargas de trabalho geoespaciais, bem como consultas geográficas, nessa ordem) (KANWAR, 2019). Este trabalho visa fazer uma implementação semelhante sobre o YCSB, com pequenas modificações nas consultas e nas métricas, conseguindo assim a ferramenta necessária para medir o desempenho

[5] https://github.com/cmu-db/benchbase

Figura 6 – Adaptação da ferramenta YCSB para consultas geoespaciais



Fonte: Kanwar (2019)

dos bancos de dados sobre as consultas já citadas.

3 TRABALHOS RELACIONADOS

Neste capítulo estão listados os principais trabalhos relacionados, os quais alicercearam o presente neste documento, além das similaridades também estão presentes as características únicas de cada um, um breve resumo da contribuição deste trabalho sobre os demais e, por fim, o Quadro 1 que os compara de maneira sucinta.

3.1 NoSQL Real-Time Database Performance Comparison

O trabalho de Pereira *et al.* (2018) entende que o modelo de dados NoSQL pode variar conforme a demanda da aplicação que o usa e ressalta que o modelo de bancos não relacional costuma tratar de acesso e escrita de dados em tempo real, ou o mais próximo possível do tempo real. Nele é feita a análise de três bancos de dados, dois deles tratando de bancos baseados em documento (RethinkDB e MongoDB Enterprise) e um multi-modelo (Couchbase Server), aceitando dados NoSQL tanto baseado em documentos quanto em formato chave-valor. Assim como no trabalho de Pereira *et al.* (2018), este trabalho também propões analisar esses modelos (orientado a documentos e chave-valor). Entretanto, para o modelo chave-valor será usado o banco de dados Redis.

A análise do desempenho do trabalho de Pereira *et al.* (2018) é feita sobre simples operações de *Create, Read, Update and Delete*/Criação, Consulta, Atualização e Apagamento de dados (*CRUD*) e medida em relação à média, moda e 90 percentil sobre a latência e vazão. Para as rodadas de testes no experimento (a primeira, usando de *thread* única, fazendo 100 vezes uma requisição por segundo e a segunda, usando de múltiplas *thread*, fazendo cinco vezes 1000 solicitações por segundo) fora usado para coletar resultados em cima das métricas estabelecidas o Apache JMeter.

Por conseguinte, os resultados obtidos por Pereira *et al.* (2018) foram que na maioria das operações Couchbase Server se saiu melhor, salvo pela operação de leitura de vários documentos e a operação de criação usando múltiplas *thread*, nas quais MongoDB Enterprise se mostrou mais rápido.

3.2 Comparative Performance Evaluation of Relational and NoSQL Databases for Spatial and Mobile Applications

Seguindo a proposta de análise de desempenho em banco de dados, no trabalho de Santos *et al.* (2015) não é medido apenas o desempenho dos bancos não relacionais, mas também de relacionais, em um contexto de aplicações para dispositivos móveis que recorrem a dados espaciais. Há nele a ideia de que escolher um banco de dados que trata de dados espaciais (ou geoespaciais) para aplicações depende, não somente de quão bem o gerenciador trata o modelo de dados usado, mas também de quais funções são mais executadas e necessitam de mais atenção no ambiente da aplicação. Por isso, é levantada a ideia de que na análise de desempenho de bancos de dados que utilizam dados geoespaciais as funcionalidades que precisam ser consideradas e o formatos de dados são mais importantes do que a análise feita por *benchmarks* genéricos.

No trabalho são usados três bancos de dados, um relacional (PostgreSQL4) e dois não relacionais (MongoDB, baseado em documentos, e Neo4j, baseado em grafos), todos possuindo suporte interno ou externo para consultas espaciais. A análise de desempenho foi feita por uma variedade de consultas comuns a aplicações móveis espaciais (pontos de interesse próximo, visão de mapa, roteamento urbano e acompanhamento de posição) sobre um conjunto de dados georreferenciado extraídos do mundo real, ou seja, dados que não foram gerados aleatoriamente, mas que são consequências reais de ações geralmente humanas.

Assim como no trabalho de Santos *et al.* (2015), este trabalho também visa comparar o desempenho entre MongoDB, Neo4j e PostgreSQL para dados geoespaciais, usando um conjunto de dados extraídos do mundo real sobre consultas baseadas em POI, porém adicionando também o banco chave-valor Redis.

No que tange a análise do desempenho no trabalho de Santos *et al.* (2015) os seguintes passos foram considerados: carregamento de dados (importando o conjunto de dados para cada banco), geração de carga de trabalho (indexando e garantindo consultas limpas), aquecimento (melhoria do desempenho dos bancos de dados), avaliação (execução das consultas pré-definidas) e desligamento (encerramento de todos os recursos abertos por todos os bancos). As métricas de avaliação usadas foram o tempo de execução de cada consulta e a métrica de vértice por segundos. Dessa forma, o trabalho de Santos *et al.* (2015) chegou a conclusão de que, no cenário de múltiplas consultas geoespaciais diversas, o PostgreSQL com PostGIS costuma ser melhor para aplicações, seguido logo após pelo MongoDB.

3.3 Testing Spatial Data Deliverance in SQL and NoSQL Database Using NodeJS Fullstack Web App

Entendendo a necessidade de analisar o desempenho de banco de dados em relação a *Big Data* geoespacial, Laksono (2018) em seu trabalho propõe uma análise voltada a aplicações *WEB* baseadas no framework MEAN/SEAN (usando NodeJS, AngularJS, ExpressJS e Mongoose ou Sequelize para conectar o aplicativo a um banco de dados) sobre carregamento de dados de POI. Os bancos de dados escolhidos para análise foram o PostgreSQL com o suporte PostGIS e o MongoDB com o suporte 2Dsphere Index para otimização espacial. Semelhante ao trabalho de Laksono (2018), este trabalho também tem a proposta de analisar o desempenho do MongoDB e o PostgreSQL com relação a POI em aplicações com sistema de informação geográfica (GIS).

Para tanto Laksono (2018) projetou e implementou uma aplicação *WEB* em seu trabalho que mediu o tempo de resposta XMLHttpRequest (XHR), API JavaScript para enviar requisições entre o navegador e o servidor com números diferentes de testes de carregamento de dados e chegou a conclusão que, neste cenário de apenas carregamentos(ou *INSERT*) de pontos de interesse no banco de dados, a medida que a quantidade de dados sobe, a otimização do MongoDB se sobressai sobre o desempenho do PostgreSQL.

3.4 Performance Evaluation of NoSQL Systems Using Yahoo Cloud Serving Benchmarking Tool

O foco do trabalho de Chakraborttii (2019), bem como neste trabalho, é fazer uma comparação de desempenho entre bancos de bancos NoSQL com categorias de armazenamento distintos, (MongoDB, Apache Cassandra, Redis e OrientDB), porém em ambientes de consultas comuns para descobrir qual apresenta melhor execução. No trabalho, é ressaltado a importância da pesquisa e testes de desempenho em banco de dados não relacionais, visto sua grande variedade.

Outra similaridade com este trabalho está na utilização da ferramente de *benchmarking* Yahoo Cloud Serving Benchmark Client (YCSB) para realização do experimento, com cargas de trabalho personalizadas e baseadas na utilização real, salvo que no trabalho de Chakraborttii (2019), a análise de desempenho é feita sobre consultas comuns realizadas em bancos dados (leitura, escrita, buscas por identificadores e atualizações) sobre dados gerados aleatoriamente e não sobre consultas de pontos de interesse em dados geográficos reais.

3.5 Comparação Entre Trabalhos

Como é possível observar no quadro da Quadro 1, os bancos de dados usados nos trabalhos relacionados encontrados sobre análise de desempenho geralmente fazem isso sobre dados não espaciais, costumeiramente abrangendo bancos relacionais, contudo com poucos tipos de banco de dados não-relacionais diferentes.

Quadro 1 – Comparação entre trabalhos

	Tipos de bancos de dados	**Banco de dados**	**Métricas**	**Conjunto de dados**	**Consultas**
Este trabalho	Orientado a documento, orientado a grafos chave-valor e relacional	Neo4j, Postgres SQLRedis e MongoDB	Tempo de execução, latência e vazão	US Accidents (2016-2021)	Pontos de Interesse (k-vizinho mais próximo, região e intervalo) e *Insert*
[Santos et al., 2015]	Orientado a documento, orientado a grafos e relacional	Neo4j, PostgreSQL e MongoDB	Tempo de execução e vértices por segundo	2013 TIGER	Pontos de Interesse (k-vizinho mais próximo e região)
[Laksono, 2018]	Orientado a documento e relacional	MongoDB e PostgreSQL	Latência	Gerado aleatoriamente	*Insert*
[Pereira et al., 2018]	Orientado a documento, chave-valor	MongoDB Enterprise CouchBase Server e RethinkDB	Latência e vazão	Gerado aleatoriamente	*Post, Patch, Get, Get by Id e Delete*
[Chakraborttii, 2019]	Orientado a documento, orientado a grafos, orientado a coluna e chave-valor	Apache Cassandra, OrientDB, MongoDB e Redis	Latência e vazão	Gerado aleatoriamente	*Read, Scan, Update e Insert*

Fonte: Fonte: elaborado pela autora (2023).

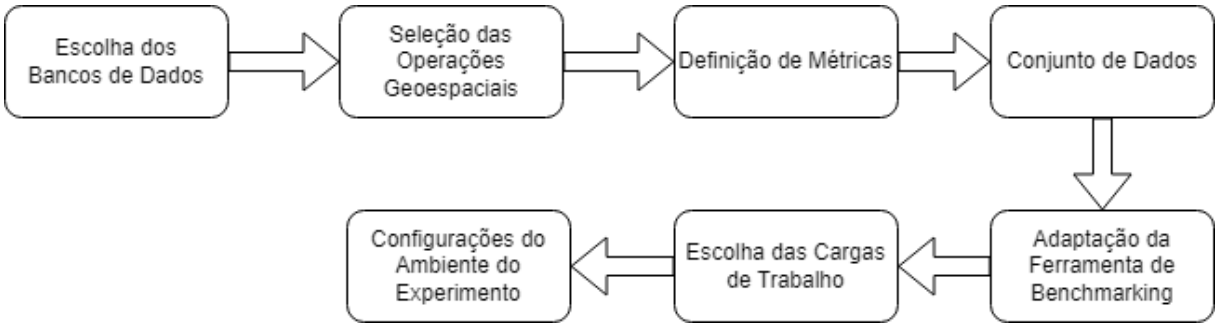
A contribuição deste trabalho está alicerceada em realizar uma análise de desempenho sobre três tipos de bancos NoSQL diferentes e um banco de dados SQL sobre carregamento e consultas geoespaciais de pontos de interesse. Ademais, também foi implementada uma extensão da ferramenta YCSB que usa consultas geoespaciais e disponibilizada na ferramenta de hospedagem de código aberto GitHub[1].

[1] https://github.com/vivi-more/YCSB-GeoPoints

4 METODOLOGIA

Este capítulo explana sobre os procedimentos utilizados durante o desenvolvimento da comparação entre os bancos de dados deste trabalho. Inicialmente foram definidos quais sistemas de bancos utilizar. Em seguida, foram selecionadas quais consultas seriam realizadas sobre eles. Por terceiro, foram definidas as métricas para avaliação. Por conseguinte, o conjunto de dados que seria utilizado. Após isso, foi adaptado e implementado a ferramenta de benchmarking e as cargas de trabalho foram preparadas. Por fim, é dada uma breve explicação das configurações da máquina que será usada no experimento. Os passos podem ser observados na Figura 7 abaixo.

Figura 7 – Passos da metodologia



Fonte: elaborado pela autora (2023).

4.1 Escolha dos Sistemas de Banco de Dados

Foram escolhidos bancos de dados de código aberto para representar as categorias de banco de dados relacional e bancos de dados não-relacional, deste último, modelos de uso comum como orientado a documentos, chave-valor e orientado a grafos foram escolhidos. Sua escolha foi baseada nos trabalhos relacionados a este e em sua capacidade de gerir dados geográficos, naturalmente ou por *plugins* e extensões. O Quadro 2 mostra um quadro que faz uma sucinta comparação entre os bancos de dados escolhidos: PostgreSQL, MongoDB, Neo4J e Redis.

O banco de dados relacional escolhido foi o PostgreSQL 15 com sua extensão PostGIS para dados geoespaciais. Entrando no tópico de bancos de dados não-relacionais, o Redis foi usado como representante do modelo chave-valor por possuir suporte nativo a armazenamento e tratamento de dados geográficos, assim também o MongoDB, representante do modelo orientado a documento. Para o modelo orientado a grafos, foi escolhido em sua versão 4, o Neo4j, por suportar o *plugin* Neo4j Spatial. A possibilidade de usar o Apache Cassandra como

Quadro 2 – Comparação entre os bancos de dados

	Modelo	**Linguagem**	**Suporte geoespacial**
PostgreSQL	Relacional	*Structured Query Language (SQL)*	PostGIS
MongoDB	Orientado a documento	*MongoDB Query Language (MQL)*	Nativo
Neo4j	Orientado a grafos	Cypher	Neo4j Spatial
Redis	Chave-valor	*Sem Linguaguem Definida*	Nativo

Fonte: Fonte: elaborado pela autora (2023).

representante do modelo orientado a coluna assim como em um dos trabalhos relacionados foi ponderada, porém, este não possui suporte nativo a dados espaciais em sua versão de código aberto e nas pesquisas não foi encontrado outro banco do mesmo modelo que possuísse.

4.2 Seleção das Operações Geoespaciais

Após a escolha dos bancos de dados, outros pontos que precisaram ser levantados foram sobre o carregamento de dados e sobre as consultas. Para uma comparação mais simples entre os bancos, índices espaciais foram adicionados apenas se estritamente necessários para efetuar buscas, como foi o caso do MongoDB e do Neo4j, que precisavam de um índice ligado a um dado para indicar que ele era geoespacial.

As consultas foram focadas em buscas em pontos de interesses (região, intervalo e k-vizinhos mais próximo), explicadas no Capítulo 2 deste documento. Os bancos previamente escolhidos possuem suporte a estas consultas, salvo pelo Redis que precisa performar a pesquisa de k-vizinhos mais próximo e adaptar a busca por intervalo. Por questões de padronização e suporte dos bancos, a consulta de intervalo será feita sobre quadrados.

O carregamento de dados e as consultas em linguagem SQL para o PostgreSQL podem ser observadas no Apêndice A.

No apêndice B é apresentado o carregamento de dados geoespaciais no Redis e a consulta nativa por região, a busca por k-vizinho mais próximo (que é adaptada passando o maior número permitido como raio e limitando o retorno). Ainda no Redis, a consulta por intervalo foi adaptada e é feita utilizando a fórmula haversine da trigonometria (para pegar distância entre pontos), descobrindo a maior distância entre pontos de um polígono e fazendo uma consulta por região, em seguida é usado o algoritmo *Winding Number* (Número de Enrolamento) para saber se o ponto pertence ou não ao polígono, conforme mostrado no apêndice. Todos esses

algoritmos utilizaram coordenadas de latitude e longitude.

Por causa dessa implementação no cliente ao se utilizar o Redis, é possível que o desempenho dele caia significativamente em relação aos demais bancos. Outro fator que pode implicar em perda de desempenho é o fato de não ser possível guardar mais dados junto a dados geográficos, obrigando o desenvolvedor a fazer conexões entre diferentes lugares por meio de uma representação de "chave estrangeira" para buscar eventuais dados a mais, visto que o dado geográfico nesse banco suporta apenas latitude e longitude.

No Apêndice C é mostrado como os dados são carregados e buscados utilizando a linguagem Cypher do Neo4j, bem como é feita a criação do índice espacial que deve ser inserido antes do carregamento dos dados geográficos. Por fim, o Apêndice D possui representações em Json das operações feitas com o MongoDB.

4.3 Definição de Métricas

As métricas escolhidas foram três métricas comuns na avaliação de desempenho entre sistemas de banco de dados:

- Latência Média: Média dos períodos do início da consulta até o retorno da resposta de sua conclusão.
- Vazão: Quantidade total de operações realizadas por segundo (sobre a carga de trabalho).
- Tempo de Execução: Por quanto tempo a operação executou na CPU.

4.4 Conjunto de dados

O conjunto de dados usado neste trabalho foi um conjunto de dados reais, visto que muitas vezes dados geográficos gerados aleatoriamente não tendem a seguir uma distribuição que se pareça com a realidade. O conjunto utilizados no experimento *US Accidents (2016 – 2021)*[1] foram obtidos na plataforma que possui um acervo de conjunto de dados para uso gratuito conhecida como Kaggle, ele representa a geolocalização de acidentes de carro nos Estados Unidos no período de 2016 a 2021.

Os dados contidos no conjunto de dados estão no formato de arquivo ".CSV" que possui mais de 1 GB em registros. O Quadro 3 é um retrato dos atributos abstraídos e utilizados

[1] https://www.kaggle.com/datasets/sobhanmoosavi/us-accidents

Quadro 3 – Detalhamento do conjunto de dados

	Descrição	**Tipo de dados**
id	Identificador do registro	Conjunto de caracteres
latitude	Latitude do acidente	Numeral
longitude	Longitude do acidente	Numeral
time_of_record	Data que ocorreu o acidente	Temporal

Fonte: Fonte: elaborado pela autora (2023).

desse conjunto. Sua representação em cada sistema de banco de dados pode ser observada nas Figuras 1, 2, 3 e 4 citadas no Capítulo 2.

4.5 Adaptação da Ferramenta de Benchmarking

Para fazer a apuração das métricas foi escolhida Yahoo Cloud Serving Benchmarking (YCSB) [2], uma ferramenta de análise de desempenho entre sistemas de bancos de dados não relacionais, que suporta também bancos relacionais, desenvolvida usando a linguagem JAVA. Essa ferramenta é amplamente usada por possuir fácil manuseio e acoplamento de bancos diversos serem permitidas nela.

Algumas classes dessa ferramenta foram alteradas como fez Kanwar (2019) para que o desempenho das operações geoespaciais pudessem ser analisado. O YCSB por padrão possui a capacidade de medir o desempenho de banco de dados para as operações denominadas *read*, *scan*, *update*, *insert* e *delete* contidas em uma classe abstrata “DB”. Essa classe foi alterada para estender uma classe abstrata “GeoDB” que possui as operações denominadas *scanKnn* (para consulta de k-vizinho mais próximo), *scanByDistance* (para consulta de região) e *scanByPolygon* (para consulta de intervalo).

Essas operações são chamadas na classe indicada no arquivo com as variáveis de carga de trabalho a ser executada. Essa classe deve obrigatoriamente estender a classe abstrata “Workload” que possui as operações padrão de carregamento de dados e execução das operações. Para tanto, foi criada e implementada a classe “GeoPointWorkload”, ela faz a preparação das operações usando dados reais carregados previamente do arquivo “.CSV” do conjunto de dados utilizado, escolhendo aleatoriamente pontos para as consultas e os transformando em objetos do contidos na classe "GeoPoint".

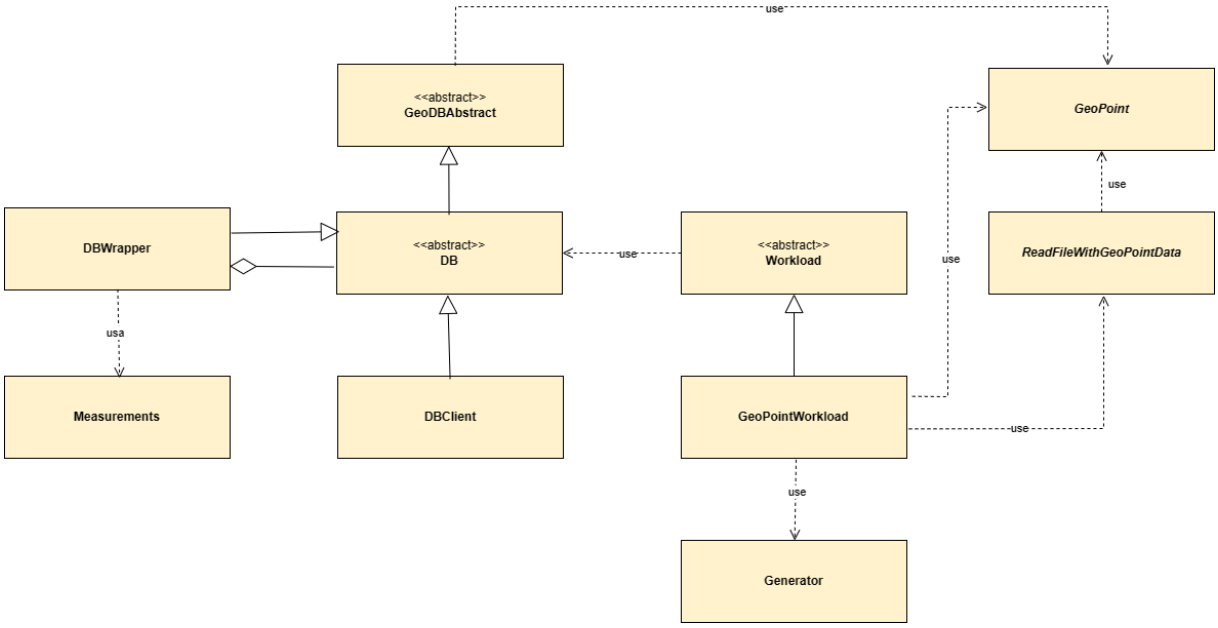
O carregamento de dados é feito sequencialmente nessa mesma classe. Além disso, ela foi implementada de forma que possa ser atribuído no arquivo propriedades de carga de trabalho referentes a cada consulta, tal como número mínimo e máximo de vizinhos na busca

[2] https://github.com/brianfrankcooper/YCSB

do k-vizinho mais próximo, máximo e mínimo da distância na busca por região, e máximo e mínimo de largura do quadrado na busca por intervalo. Também é indicada a propriedade de proporção que cada consulta será executada sobre o número de operações indicadas no arquivo.

As classes que estendiam a classe abstrata “DB” também foram alteradas para suportarem as demais funções estendidas da classe "GeoDB". Ademais, a integridade do sistema foi mantido nas classes que faziam controle de *Threads* e geravam as métricas padrões da ferramenta, sendo elas latência (mínima, máxima, média, 95 percentil e 99 percentil), vazão e tempo de execução. Os bancos de dados utilizaram diferentes bibliotecas para se comunicarem com a ferramenta por meio da linguagem Java, o PostgreSQL utilizou "org.postgresql.driver” na v42.5.4, o Redis utilizou "jerdis” v2.0.0, o Neo4j usou "neo4j-java-driver” v4.3.6 e, por fim, o MongoDB utilizou o "mongo-java-driver” v3.0.3. É possível observar todas essas modificações no repositório público do Github denominado YCSB- GeoPoints[3]. A Figura 8 ilustra bem as modificações feitas.

Figura 8 – Classe modificadas da ferramenta YCSB



Fonte: elaborado pela autora (2023).

4.6 Escolha das Cargas de Trabalho

Assim como Pereira *et al.* (2018) fez em seu trabalho, os experimentos aconteceram em duas rodadas, uma sequencial de apenas uma *thread* e outra utilizando cinco *threads*,

[3] https://github.com/vivi-more/YCSB-GeoPoints/tree/main

respeitando a capacidade do ambiente de experimento. Como Laksono (2018) e Chakraborttii (2019) a quantidade e operações realizadas foi uma escala progressiva de 1000, 10000 e 100000. Cada operação foi executada 5 vezes. Na consulta de região o raio, bem como Santos *et al.* (2015) fez, foi limitando a variação de 1 a 100 quilômetro, bem como a largura do quadrado na consulta por intervalo. Também foi seguido o exemplo de seu trabalho ao escolher o número variantes de vizinhos na consulta de k-vizinhos mais próximos (de 1 a 10). Seguindo essas variações as cargas de trabalho se dividiu em:

- A: 100% consultas de k-vizinho mais próximo;
- B: 100% consultas de intervalo;
- C: 100% consultas de região;
- E: 100% operação de inserção de dados geoespaciais;
- D: 33% consultas de região, 33% consultas de intervalo, 33% consultas de k-vizinho mais próximo.

Os bancos de dados foram todos instalados na mesma máquina, executavam simultaneamente, e as consultas por banco ocorreram sequencialmente, uma após a outra, exceto quando se utilizava *threads*, que executaram consultas em paralelo, mesmo assim, apenas um banco por vez foi usado. A execução de testes dos experimentos foi coletado localmente. As execuções das *threads* serão explicadas nos capítulos posteriores.

4.7 Configurações do Ambiente do Experimento

A máquina escolhida para a realização do experimento possui as seguintes configurações:

- CPU: Intel(R) Core(TM) i5-8265U CPU 1.60GHz 1.80 GHz;
- RAM: 16 GB;
- SSD: 500 GB;
- Sistema Operacional: Windows 11.

As versões dos bancos foram escolhidas considerando as configurações, bem como o número de *threads* usar nas operações em paralelo.

5 RESULTADOS

Este capítulo apresenta o resultado de desempenho dos bancos de dados para cada carga de trabalho já discutidas.

5.1 Carregamento dos Dados (Operação de *INSERT*)

O carregamento dos dados geográficos consistia em operações de inserção de dados no formato exigido por cada banco para que ele fosse reconhecido como geoespacial. A Tabela 1 representa a latência média, o tempo de execução médio e a vazão média no cenário com uma única e no cenário com cinco *threads*. É possível notar que o Redis se saiu melhor que os demais bancos, como está destacado em negrito.

Tabela 1 – Operações de inserção de dados nos bancos

Operação de Inserção				
Thread Única				
Latência (milissegundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	0,56	**0,23**	0,76	4,02
10000	0,45	**0,16**	0,46	4,87
100000	0,37	**0,11**	0,23	24,76
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	1,24	**0,31**	1,78	5,43
10000	5,29	**1,74**	6,19	50,62
100000	37,44	**11,10**	27,80	2.483,62
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	814,24	**3.190,30**	561,95	184,95
10000	1.890,81	**5.770,48**	1.620,98	199,23
100000	2.679,39	**9.012,72**	3.662,56	40,29
Cinco Threads				
Latência (milissegundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	1,31	**0,72**	2,10	58,00
10000	0,75	**0,47**	0,74	24,78
100000	0,62	**0,29**	0,52	125,59
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	1,04	**0,22**	1,43	27,45
10000	2,38	**1,06**	2,82	51,33
100000	13,17	**6,25**	14,90	2.519,40
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	998,06	**4.722,29**	707,41	36,43
10000	4.210,76	**9.484,07**	3.613,82	197,19
100000	7.675,17	**16.031,27**	6.725,73	39,70

Fonte: elaborado pela autora (2023).

5.2 Consulta de K-Vizinhos Mais Próximo

A consulta de K-vizinhos mais próximos consistia em procurar uma quantidade aleatória de vizinhos de um ponto geográfico no banco com as métricas já pré-programadas. A Tabela 2 possui a representação dos resultados no cenário de uma única *thread* e no cenário de cinco *threads*. Nela é possível observar que o MongoDB se sobressaiu sobre os demais bancos.

Tabela 2 – Consulta de K-vizinhos mais próximos nos dados nos bancos

Consulta de K-vizinhos mais próximos				
Thread Única				
Latência (milissegundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	89,45	77,42	**3,63**	259,85
10000	99,72	74,33	**2,13**	193,37
100000	102,35	73,28	**1,95**	194,99
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	90,20	77,60	**4,69**	260,75
10000	998,31	744,00	**22,53**	1.934,89
100000	10.240,29	7.333,05	**198,39**	19.504,43
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	11,09	12,93	**213,48**	3,84
10000	10,02	13,44	**444,68**	5,17
100000	9,77	13,64	**504,52**	5,13
Cinco Threads				
Latência (milissegundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	325,57	433,92	**8,00**	1.438,50
10000	265,96	366,81	**5,04**	963,21
100000	287,06	375,03	**4,05**	981,47
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	66,43	88,29	**2,63**	289,22
10000	5.337,79	738,14	**11,27**	1.927,55
100000	5.748,62	7.524,02	**83,11**	19.631,72
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	15,07	11,35	**380,34**	3,46
10000	18,73	13,55	**887,56**	5,19
100000	17,48	13,29	**1.203,26**	5,09

Fonte: elaborado pela autora (2023).

5.3 Consulta de Região

A consulta de região se focava em buscar todos os pontos em uma determinada distância de outro ponto. A Tabela 3 mostra os resultados dessa consulta. Mais uma vez o

MongoDB se sobressaí sobre os demais em todas as métricas e cenários conforme destacado em negrito.

Tabela 3 – Consulta de Região nos dados nos bancos

Consulta de Região				
Thread Única				
Latência (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	0,07	0,06	**0,01**	0,19
10000	0,10	0,12	**0,02**	0,15
100000	0,05	0,08	**0,01**	0,13
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	74,53	57,00	**15,42**	189,32
10000	958,09	1.203,85	**218,77**	1.486,81
100000	4.647,05	11.764,50	**1.367,31**	14.868,05
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	14,09	18,97	**65,04**	5,65
10000	10,44	8,36	**45,72**	6,73
100000	21,53	8,16	**18,47**	7,06
Cinco Threads				
Latência (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	0,13	0,11	**0,03**	0,64
10000	0,22	0,33	**0,05**	0,78
100000	0,11	0,25	**0,03**	0,71
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	26,33	23,28	**6,46**	128,67
10000	442,72	677,42	**99,62**	1.553,74
100000	2.178,59	6.096,75	**688,11**	14.943,34
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	37,98	43,02	**154,81**	7,77
10000	22,59	14,77	**100,47**	6,45
100000	45,90	24,77	**52,60**	7,34

Fonte: elaborado pela autora (2023).

5.4 Consulta de Intervalo

A consulta por intervalo consistia em fazer uma busca numa região determinada por um polígono, neste trabalho sempre foram usados quadrados, apenas variando seu tamanho. As Tabela 3 apresentam os resultados nessa consulta. Nela o MongoDB continua a apresentar melhor resultado, porém os bancos Redis e Neo4j disputam o pior desempenho nessas consultas. Salvo pela execução de *thread* única com 100000 operações, onde o PostgreSQL se saiu ligeiramente melhor.

Tabela 4 – Consulta de Intervalo nos dados nos bancos

Consulta de Intervalo				
Thread Única				
Latência (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	0,07	0,04	**0,01**	0,06
10000	0,09	0,10	**0,01**	0,06
100000	0,04	0,14	**0,02**	0,07
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	73,20	41,56	**8,81**	59,04
10000	926,71	1.024,87	**101,85**	604,95
100000	**4.335,95**	14.035,25	17.923,68	6.910,89
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	15,66	24,13	**113,90**	16,94
10000	10,79	9,77	**98,47**	16,53
100000	24,96	7,12	**55,79**	14,47
Cinco Threads				
Latência (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	0,25	0,15	**0,02**	0,29
10000	0,20	0,24	**0,02**	0,30
100000	0,09	0,33	**0,03**	0,29
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	50,15	32,86	**4,53**	59,12
10000	399,45	491,25	**47,05**	603,33
100000	1.803,91	6.722,05	**687,29**	6.873,05
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	19,95	30,55	**221,47**	16,93
10000	25,04	20,37	**212,55**	16,57
100000	49,72	14,88	**145,50**	14,55

Fonte: elaborado pela autora (2023).

5.5 Consulta Mista (k-Vizinhos Mais Próximos, Região e Intervalo)

A consulta mista também foi feita sequencialmente ao se usar uma única *thread*. Porém, ela foi dividida em 33.3% entre as operações. Isso significa que se existissem 100 operações, 33 seriam de um tipo de consulta (k-vizinhos mais próximos), 33 de outro tipo (região) e 34 da que sobrou (intervalo). A carga de trabalho apenas atua em paralelo quando existem *threads* adicionadas a ela. Assim, a latência dessa carga de trabalho foi abstraída, visto que seriam individuais para cada consulta, sendo tratadas apenas as medidas de tempo de execução e vazão.

A Tabela 5 apresenta o tempo de execução dessa carga de trabalho usando apenas uma única *thread* e utilizando cinco *threads*. Nas operações com uma única *thread*, o PostgreSQL se saiu melhor, possuindo e menor tempo de execução, entretanto, nas operações com cinco

threads, o MongoDB possui o menor tempo. Já o maior tempo de execução ficou com Neo4j, exceto pela execução com 10000 operações e cinco *threads*, na qual o PostgreSQL se saiu pior.

Tabela 5 – Consulta de Mista nos dados nos bancos

Consulta de Mista				
Thread Única				
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	96,99	**62,58**	**62,58**	187,74
10000	**940,33**	952,19	960,57	1.310,03
100000	**9.863,85**	11.783,48	12.010,18	14.063,63
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	10,36	**16,00**	**16,00**	5,33
10000	**10,63**	10,32	10,49	7,63
100000	**10,14**	8,00	8,33	7,11
Cinco Threads				
Tempo de Execução (segundos)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	56,13	44,03	**4,77**	891,30
10000	477,98	550,78	**54,38**	2.445,79
100000	5.431,50	6.782,88	**621,01**	10.957,99
Vazão (operações/segundo)				
Operações/Bancos	**PostgreSQL**	**Redis**	**MongoDB**	**Neo4j**
1000	17,82	22,73	**210,12**	5,99
10000	20,92	18,16	**184,14**	11,22
100000	18,41	14,74	**161,03**	9,13

Fonte: elaborado pela autora (2023).

Finalmente, a Tabela 5 também apresenta os resultados da vazão nessa carga de trabalho sem uso e com uso de *threads*. Aqui a leitura dos dados não apresenta realmente um banco que se saiu melhor nas operações com uma única *thread*, apenas aquele que apresentou pior desempenho (Neo4j). Já no caso de uso de cinco *threads*, o MongoDB apresentou mais operações por segundo e o Neo4J seguiu com menos operações por segundo.

5.6 Discussão de Resultados

Nos experimentos, na operação de inserção de dados geoespaciais, o banco de dados não-relacional Redis apresentou desempenho melhor nas métricas de latência, tempo de execução e vazão, o motivo disso está provavelmente contido no modo simplista deste banco representar seus dados.

Para as demais operações, o MongoDB apresentou resultados melhores na maioria dos cenários apresentados, nele, provavelmente o motivo de seu exito também está no modo simplista de tratar os dados contidos nesse banco e como ele já é um banco robusto o suficiente

para realizar consultas geoespaciais complexas nativamente.

Entre os possíveis motivos do mal desempenho do Neo4j está o uso do *plugin* Neo4j Spatial, que não é nativo desse banco, mas sim desenvolvido por um grupo de pesquisadores para tornar pesquisas geoespaciais possíveis nele. Isso não significa que banco de dados orientados a grafos sempre terão um desempenho parecido em consultas geoespaciais.

O desempenho do PostgreSQL com a extensão PostGIS nesses cenários foi mediano se comparado aos demais bancos, há a possibilidade disso ter diferido caso indexes espaciais fossem usados nele.

6 POSSÍVEIS MELHORIAS PARA TRABALHOS FUTUROS

Alguns pontos do trabalho podem ser aperfeiçoados para melhor desempenhos dos bancos de dados e gerar mais precisões de resultados. Por exemplo, neste trabalho não foi utilizado índices espaciais no PostgreSQL, por conseguinte, o desempenho desse banco pode ter sido afetado por isso. No banco Redis as consultas de K-vizinhos mais próximos e de Intervalo podem ser aperfeiçoados para performar melhor tirando as repetições quadráticas.

Os bancos foram também executados em simultâneo, mesmo que as operações realizadas neles fossem sequenciais, um banco pode ter interferido no outro, o cenário ideal seria parar todos os serviços desses bancos e executá-los um por vez apenas quando os experimentos no banco em específico começasse. Outro ponto a se levantar é que para os dados serem obtidos de um modo mais próximo do uso do banco de dados no mundo real é coletar apenas os dados do meio do experimento, entre os primeiros minutos e últimos minutos.

7 CONSIDERAÇÕES FINAIS

Para o planejamento da execução deste trabalho foi preciso levantar se era viável, ou mesmo preciso, uma avaliação de sistemas de bancos de dados sobre dados geográficos. Chegada a conclusão de que sim, foi preciso fazer um levantamento dos principais tipos de bancos e de seus respectivos sistemas, bem como de que maneira era feita a avaliação e comparação sobre banco de dados e a adaptação de uma ferramenta de *benchmarking* para permitir avaliação sobre consultas geoespaciais.

A confiabilidade do experimento precisou ser apoiada em alguns pilares como todos os bancos estarem instalados na mesma máquina, receberem os mesmos dados, usado a mesma ferramenta de coleta em bancos com as mesmas métricas e cargas de trabalho. Além disso, apesar dos bancos possuírem diferentes formas de armazenamento e consulta, os dados foram modelados para se aproximarem ao máximo do modelo conceitual de cada tipo, bem como as consultas (nativas ou com apoio de bibliotecas e *plugins*), replicando um padrão em cada um dos bancos.

Em suma, este documento serve como uma base para a execução de trabalhos de avaliação de desempenho em banco de dados usando consultas geoespaciais de forma prática e concisa, podendo a vir ser usada para escolha de sistemas de banco de dados deste tipo no futuro.

REFERÊNCIAS

AMIRIAN, P.; BASIRI, A.; WINSTANLEY, A. Evaluation of data management systems for geospatial big data. In: SPRINGER. **International Conference on Computational Science and Its Applications**. [S.l.], 2014. p. 678–690.

BENCHANT. **The Ultimate YCSB Benchmark Guide (2021)**. 2021. Disponível em: https://benchant.com/blog/ycsb. Acesso em: 01 jun. 2022.

BUGIOTTI, F.; CABIBBO, L.; ATZENI, P.; TORLONE, R. Database design for nosql systems. In: SPRINGER. **International Conference on Conceptual Modeling**. [S.l.], 2014. p. 223–231.

CHAKRABORTTII, C. Performance evaluation of nosql systems using yahoo cloud serving benchmarking tool. 2019.

DAVOUDIAN, A.; CHEN, L.; LIU, M. A survey on nosql stores. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 2, p. 1–43, 2018.

DOMINGUES, A.; SILVA, F.; SANTOS, L.; SOUZA, R.; COIMBRA, G.; LOUREIRO, A. A. F. Dados geoespaciais: Conceitos e técnicas para coleta, armazenamento, tratamento e visualização. **Sociedade Brasileira de Computação**, 2020.

GUDIVADA, V. N.; RAO, D.; RAGHAVAN, V. V. Nosql systems for big data management. In: IEEE. **2014 IEEE World congress on services**. [S.l.], 2014. p. 190–197.

IBGE - INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATíSTICA. **manuais técnicos em geociências**: avaliação da qualidade de dados geoespaciais. Rio de Janeiro, 2017.

KANWAR, Y. S. Benchmarking scalability of nosql databases for geospatial queries. 2019.

LAKSONO, D. Testing spatial data deliverance in sql and nosql database using nodejs fullstack web app. In: IEEE. **2018 4th International Conference on Science and Technology (ICST)**. [S.l.], 2018. p. 1–5.

NIYIZAMWIYITIRA, C.; LUNDBERG, L. Performance evaluation of sql and nosql database management systems in a cluster. **International Journal of Database Management Systems**, v. 9, n. 6, p. 1–24, 2017.

ORACLE. **O que É um Banco de Dados Relacional?** 2020. Disponível em: https://www.oracle.com/br/database/what-is-a-relational-database. Acesso em: 01 jun. 2022.

ORACLE. **O que é NoSQL?** 2020. Disponível em: https://www.oracle.com/br/database/nosql/what-is-nosql. Acesso em: 01 jun. 2022.

PEREIRA, D. A.; MORAIS, W. Ourique de; FREITAS, E. Pignaton de. Nosql real-time database performance comparison. **International Journal of Parallel, Emergent and Distributed Systems**, Taylor & Francis, v. 33, n. 2, p. 144–156, 2018.

RAY, S.; SIMION, B.; BROWN, A. D. Jackpine: A benchmark to evaluate spatial database performance. In: IEEE. **2011 IEEE 27th International Conference on Data Engineering**. [S.l.], 2011. p. 1139–1150.

SANTOS, P. O.; MORO, M. M.; DAVIS, C. A. Comparative performance evaluation of relational and nosql databases for spatial and mobile applications. In: SPRINGER. **Database and Expert Systems Applications**. [S.l.], 2015. p. 186–200.

VERA-OLIVERA, H.; GUO, R.; HUACARPUMA, R. C.; SILVA, A. P. B. D.; MARIANO, A. M.; HOLANDA, M. Data modeling and nosql databases-a systematic mapping review. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 54, n. 6, p. 1–26, 2021.

APÊNDICE A – CONSULTAS GEOESPACIAIS NO POSTGRESQL

Este apêndice possui a representação das consultas usadas no PostgreSQL.

Código-fonte 1 – Instrução SQL para Inserção de ponto geográfico

```
INSERT INTO TABLE_NAME (PRIMARY_KEY,LONGITUDE_COLUMN,
   LATITUDE_COLUMN,TIME_OF_RECORD_COLUMN) VALUES(?,?,?,?)
```

Código-fonte 2 – Consulta SQL de K-vizinho mais próximo utilizando o PostGIS

```
SELECT * FROM TABLE_NAME ORDER BY ST_SetSRID(ST_MakePoint(
   LONGITUDE_COLUMN,LATITUDE_COLUMN), 4326)  <-> ST_SetSRID
   (ST_MakePoint(?, ?), 4326)  LIMIT ?
```

Código-fonte 3 – Consulta SQL de região utilizando o PostGIS

```
SELECT * FROM TABLE_NAME WHERE ST_DWithin(ST_SetSRID(
   ST_MakePoint(LONGITUDE_COLUMN, LATITUDE_COLUMN), 4326),
   ST_SetSRID(ST_MakePoint(?, ?), 4326), ?)
```

Código-fonte 4 – Consulta SQL de intervalo utilizando o PostGIS

```
SELECT * FROM TABLE_NAME WHERE ST_Within(ST_SetSRID(
   ST_MakePoint(LONGITUDE_COLUMN, LATITUDE_COLUMN), 4326),
   ST_GeomFromText(POLYGON(? ?, ? ?, ? ?, ? ?, ? ?), 4326))
```

APÊNDICE B – CONSULTAS GEOESPACIAIS NO REDIS

Este apêndice possui a representação das consultas usadas no Redis.

Código-fonte 5 – Instrução Redis CLI para Inserção de ponto geográfico

```
GEOADD LOCATION_COLUNM <longitude> <latitude> ID
```

Código-fonte 6 – Consulta Redis CLI de região

```
GEORADIUS LOCATION_COLUNM <longitude> <latitude> <distance>
    KM WITHCOORD
```

Código-fonte 7 – Consulta Redis CLI perfomando k-vizinho mais próximo

```
# Neste caso <distance> em Java seria Double.MAX_VALUE
GEORADIUS LOCATION_COLUMN <longitude> <latitude> <distance>
    KM WITHCOORD ASC COUNT <k>
```

Código-fonte 8 – Algortmo da Fórmula de Haversine para ditância máxima entre os vértices

```
public static double calculateMaxDistanceInKM(ArrayList<
   GeoPoint> gp) {
    double maxDistance = 0.0; // Distancia maxima
       inicializada como zero

    // Iterar sobre todos os pares de pontos
    for (int i = 0; i < gp.size() - 1; i++) {
        for (int j = i + 1; j < gp.size(); j++) {
            double lon1 = gp.get(i).getLongitude(); //
               Longitude do primeiro ponto
            double lat1 = gp.get(i).getLatitude(); //
               Latitude do primeiro ponto
```

```
double lon2 = gp.get(j).getLongitude(); //
   Longitude do segundo ponto
double lat2 = gp.get(j).getLatitude(); //
   Latitude do segundo ponto

double earthRadius = 6371.0; // Raio medio da
   Terra em quilometros

// Converter as diferencas de latitude e
   longitude para radianos
double dLat = Math.toRadians(lat2 - lat1);
double dLon = Math.toRadians(lon2 - lon1);

// Calcular a distancia entre os dois pontos
   usando a formula de Haversine
double a = Math.sin(dLat / 2) * Math.sin(dLat /
    2) +
           Math.cos(Math.toRadians(lat1)) *
              Math.cos(Math.toRadians(lat2)) *
           Math.sin(dLon / 2) * Math.sin(dLon /
               2);

double c = 2 * Math.atan2(Math.sqrt(a), Math.
   sqrt(1 - a));

double distance = earthRadius * c;

// Atualizar a distancia maxima, se a distancia
    atual for maior
if (distance > maxDistance) {
    maxDistance = distance;
}
```

```
        }
    }

    return maxDistance;
}
```

Código-fonte 9 – Algortmo Winding Number

```
public static boolean isCoordinateInsidePolygon(double x,
   double y, List<GeoPoint> polygon) {
    int windingNumber = 0; // Numero de vezes que o
       poligono "enrola" em torno do ponto
    int numPoints = polygon.size(); // Numero de pontos no
       poligono

    // Iterar sobre cada segmento de linha do poligono
    for (int i = 0; i < numPoints; i++) {
        double x0 = polygon.get(i).getLongitude(); //
           Coordenada x do ponto atual
        double y0 = polygon.get(i).getLatitude(); //
           Coordenada y do ponto atual
        double x1 = polygon.get((i + 1) % numPoints).
           getLongitude(); // Coordenada x do proximo ponto
            (levando em conta a ultima volta)
        double y1 = polygon.get((i + 1) % numPoints).
           getLatitude(); // Coordenada y do proximo ponto
           (levando em conta a ultima volta)

        // Verificar se o ponto atual esta abaixo ou acima
           do ponto y
        if (y0 <= y) {
            // Verificar se o proximo ponto esta acima do
```

```
                ponto y e se o ponto est      esquerda da
                linha
            if (y1 > y && ((x1 - x0) * (y - y0)) - ((x - x0
                ) * (y1 - y0)) > 0) {
                windingNumber++; // Incrementar o n mero
                    de voltas
            }
        } else {
            // Verificar se o proximo ponto esta abaixo ou
                no mesmo nivel do ponto y e se o ponto esta
                a direita da linha
            if (y1 <= y && ((x1 - x0) * (y - y0)) - ((x -
                x0) * (y1 - y0)) < 0) {
                windingNumber--; // Decrementar o numero de
                    voltas
            }
        }
    }

    return windingNumber != 0; // Retornar verdadeiro se o
        numero de voltas for diferente de zero, indicando
        que o ponto est  dentro do poligono
}
```

Código-fonte 10 – Perfomando consulta de intervalo com o Redis

```
public Status scanByPolygon(String table, ArrayList<
    GeoPoint> polygonVertices) {
    // Calcular a distancia maxima entre os pontos do
        poligono
    double maxDistance = calculateMaxDistanceInKM(
        polygonVertices);
```

```
    // Obter todos os pontos dentro do poligono usando o
       comando GEORADIUS
    GeoRadiusParam geoRadiusParam = GeoRadiusParam.
       geoRadiusParam().withCoord().sortAscending();
    List<GeoRadiusResponse> results = jedis.georadius(
       LOCATION_COLUMN, polygonVertices.get(0).getLongitude
       (),
            polygonVertices.get(0).getLatitude(),
               maxDistance, GeoUnit.KM, geoRadiusParam);

    ArrayList<String> uniqueIdentifiers = new ArrayList<>()
       ;

    for (GeoRadiusResponse point : results) {
        String id = point.getMemberByString();
        GeoCoordinate gc = point.getCoordinate();

        // Verificar se o ponto esta dentro do poligono
        if (GeoUtils.isCoordinateInsidePolygon(gc.
           getLongitude(), gc.getLatitude(),
           polygonVertices)) {
            result.add(point);
        }
    }
}
```

APÊNDICE C – CONSULTAS GEOESPACIAIS NO NEO4J

Este apêndice possui a representação das consultas usadas no Neo4j.

Código-fonte 11 – Instrução Cypher para criação de índice geográfico

```
CALL spatial.addPointLayer(<indice>);
```

Código-fonte 12 – Instrução Cypher para Inserção de ponto geográfico

```
CREATE (n:Point {PRIMARY_KEY: $PRIMARY_KEY,LONGITUDE_COLUMN
   : $LONGITUDE_COLUMN,LATITUDE_COLUMN: $LATITUDE_COLUMN,
   TIME_OF_RECORD_COLUMN: $TIME_OF_RECORD_COLUMN}) WITH n
   CALL spatial.addNode(<indice>, n) YIELD node RETURN node
```

Código-fonte 13 – Consulta Cypher de K-vizinho mais próximo

```
MATCH (p:Point) WHERE p.id IS NOT NULL AND p.time_of_record
    IS NOT NULL RETURN p.PRIMARY_KEY,p.LATITUDE_COLUMN,p.
   LONGITUDE_COLUMN,p.TIME_OF_RECORD_COLUMN ORDER BY point.
   distance(point({latitude: p.LATITUDE_COLUMN,longitude: p
   .LONGITUDE_COLUMN}),point({latitude: $LATITUDE_COLUMN,
   longitude: $LONGITUDE_COLUMN})) LIMIT $k
```

Código-fonte 14 – Consulta Cypher de região

```
MATCH (p:Point) WHERE p.id IS NOT NULL AND p.time_of_record
    IS NOT NULL AND point.distance(point({latitude: p.
   LATITUDE_COLUMN,longitude: p.LONGITUDE_COLUMN}),point({
   latitude: $LATITUDE_COLUMN,longitude: $LONGITUDE_COLUMN
   })) <= $maxDistance RETURN p.PRIMARY_KEY,p.
   LATITUDE_COLUMN,p.LONGITUDE_COLUMN,p.
   TIME_OF_RECORD_COLUMN
```

Código-fonte 15 – Consulta Cypher de intervalo

```
WITH 'POLYGON((<coodenates>))' as polygon CALL spatial.
   intersects(<indice>, polygon) YIELD node AS p p.
   PRIMARY_KEY,p.LATITUDE_COLUMN,p.LONGITUDE_COLUMN,p.
   TIME_OF_RECORD_COLUMN
```

APÊNDICE D – CONSULTAS GEOESPACIAIS NO MONGODB

Este apêndice possui a representação das consultas usadas no MongoDB.

Código-fonte 16 – Instrução representada em Json da instrução MQL para criação de índice geográfico

```
{
  "createIndex": "collection",
  "keys": {
    "LOCATION_COLUMN": "2dsphere"
  }
}
```

Código-fonte 17 – Instrução representada em Json da instrução MQL para Inserção de ponto geográfico

```
{
  "PRIMARY_KEY": "key",
  "LOCATION_COLUMN": {
    "type": "Point",
    "coordinates": [longitude, latitude]
  },
  "TIME_OF_RECORD_COLUMN": timeOfRecord
}
```

Código-fonte 18 – Consulta representada em Json da instrução MQL de K-vizinho mais próximo

```
{
  "find": "collection",
  "filter": {
    "LOCATION_COLUMN": {
      "$near": {
        "$geometry": {
```

```
          "type": "Point",
          "coordinates": [longitude, latitude]
        }
      }
    }
  },
  "limit": k
}
```

Código-fonte 19 – Consulta representada em Json da instrução MQL de região

```
{
  "find": "collection",
  "filter": {
    "LOCATION_COLUMN": {
      "$near": {
        "$geometry": {
          "type": "Point",
          "coordinates": [longitude, latitude]
        },
        "$maxDistance": distanceInMeters
      }
    }
  }
}
```

Código-fonte 20 – Consulta representada em Json da instrução MQL de intervalo

```
{
  "find": "collection",
  "filter": {
    "LOCATION_COLUMN": {
```

```
      "$geoWithin": {
        "$geometry": {
          "type": "Polygon",
          "coordinates": [
            [
              [longitude1, latitude1],
              [longitude2, longitude2],
              ...
            ]
          ]
        }
      }
    }
  }
}
```