UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

ANDRÉ LUIS PEREIRA VIEIRA

INVESTIGAÇÃO DO POSTGIS PARA APERFEIÇOAMENTO DO ARMAZENAMENTO E DAS OPERAÇÕES SOBRE DADOS ESPACIAIS: UM ESTUDO DE CASO NO LABORATÓRIO TERRALAB

Ouro Preto, MG
2024

ANDRÉ LUIS PEREIRA VIEIRA

INVESTIGAÇÃO DO POSTGIS PARA APERFEIÇOAMENTO DO ARMAZENAMENTO E DAS OPERAÇÕES SOBRE DADOS ESPACIAIS: UM ESTUDO DE CASO NO LABORATÓRIO TERRALAB

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Rodrigo César Pedrosa Silva
Coorientador: Prof. Dr. Guilherme Tavares de Assis

Ouro Preto, MG
2024

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

André Luis Pereira Vieira

INVESTIGAÇÃO DO POSTGIS PARA APERFEIÇOAMENTO DO ARMAZENAMENTO E DAS OPERAÇÕES SOBRE DADOS ESPACIAIS: UM ESTUDO DE CASO NO LABORATÓRIO TERRALAB

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 14 de Outubro de 2024.

Membros da banca

Rodrigo César Pedrosa Silva (Orientador) - Doutor - Universidade Federal de Ouro Preto
Tiago Garcia de Senna Carneiro (Examinador) - Doutor - Universidade Federal de Ouro Preto
Guilherme Tavares de Assis (Examinador) - Doutor - Universidade Federal de Ouro Preto

Rodrigo César Pedrosa Silva, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 14/10/2024.

Documento assinado eletronicamente por **Rodrigo Cesar Pedrosa Silva**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 15/10/2024, às 22:43, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0789782** e o código CRC **9006B43E**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Resumo

O objetivo deste trabalho é buscar melhorar um processo de busca e processamento de dados espaciais no Terralab. Para isso, foi feita uma pesquisa experimental com o SGBD espacial PostGis. Utilizando mais de 8.000.000 (oito milhões) de dados disponibilizados pelo laboratório, comparar-se-á o custo de memória para armazenamento dos dados, medir eficiência de operações de consulta de dados fazendo uso de indexações, visões e tabelas temporárias, modelando os dados de maneira que facilite seu aceso, além de utilizar funções nativas dos SGBD para calcular a distância entre pontos e pertinência de um ponto em um objeto, com intuito de reduzir o custo de mão de obra e melhorar o tempo de execução. Estes são processos que podem levar horas ou talvez nem consigam ser executados no Terralab, o que pode ser inviável dependendo da aplicação. O PostGis, com 95 segundos, criou uma base que ocupa menos de 10x de espaço (442MB) que o Postgre (5484MB) levou 61 segundos para criação, com PostGis ainda armazenando malhas de cidades no seu banco. Todas as operações de distância obtiveram ganho no tempo de até 75% e o maior ganho nas pertinências, chegando a quase 99% de ganho com todas as tarefas realizadas dentro do próprio SGBD, diferente da solução do TerraLab que necessita carregar os dados em memória.

Palavras-chave: SGBDG, SGBDE, Postgis, Banco de dados espacial.

Abstract

The objective of this work is to improve a data search and processing workflow at Terralab. To achieve this, an experimental study will be conducted with two spatial DBMSs, Oracle and PostGis. Using more than 8,000,000 (eight million) data points provided by the lab, the memory cost for data storage will be compared, as well as the efficiency of query operations using indexes, views, and temporary tables, modeling the data in a way that facilitates access. Additionally, native DBMS functions will be used to calculate the distance between points and the inclusion of a point within an object, aiming to reduce labor costs and improve execution time. These processes can take hours or may not even be executable at Terralab, which may be impractical depending on the application. PostGis, in 95 seconds, created a database that takes up less than 10 times the space (442MB) compared to Postgre (5484MB), which took 61 seconds to create, with PostGis still storing city meshes in its database. All distance operations achieved up to a 75% time gain, with the greatest improvement in point inclusion, reaching nearly 99% of gains, with all tasks performed within the DBMS itself, unlike Terralab, which needs to load the data into memory.

Keywords: DBMS, Postgis, Spatial database.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABNT	Associação Brasileira de Normas Técnicas
DECOM	Departamento de Computação
UFOP	Universidade Federal de Ouro Preto
SGBD	Sistemas de Gerenciamento de Banco de Dados
SGBDE	Sistemas de Gerenciamento de Banco de Dados Espaciais
IBGE	Instituto Brasileiro de Geografia e Estatística
Terralab	Laboratório para Pesquisa e Capacitação em Desenvolvimento de Software

Lista de Algorítimos

Sumário

1 Introdução

Os Sistemas de Gerenciamento de Banco de Dados Espaciais (SGBDE) são mecanismos de criação, monitoramento e gerenciamento de banco de dados espaciais que implementam funcionalidades específicas para escrita, leitura, remoção e manipulação dos dados espaciais.

O Laboratório para Pesquisa e Capacitação em Desenvolvimento de Software (Terralab) da Universidade Federal de Ouro Preto (UFOP) coleta mais de 500.000 (quinhentos mil) endereços geolocalizados mensais de cinco Aplicações de Georreferenciamento diferentes (Google, Here, TomTom, MapBox, OpenRouteService). Além disso, o Terralab ainda conta atualmente com mais de 12.000.000 (doze milhões) de dados. Cada um desses guarda informações sobre o endereço geolocalizado: rua, número, cidade, estado, latitude e longitude, entre outros.

Um dos objetivos com estes dados é analisar os resultados das diferentes APIs para um mesmo endereço. Para tal é necessário calcular a distância entre os pontos gerados, verificar se um ponto está dentro da cidade e estado do seu endereço, calcular a quantidade de pontos para cada um das cidades e estados e qualificar estas APIs de acordo com os resultados. A aplicação que mais diferir das demais recebe uma classificação menor. Com estas informações é possível qualificar qual API gera resultados mais consistentes e criar uma coleção dos melhores dados coletados.

Entretanto, essas operações levam muito tempo para serem executadas ou nem é possível executá-las, comprometendo as análises exploratórias e o desenvolvimento de aplicações utilizando os dados gerados. Acredita-se que o maior causador desses problemas seja o SGBD não espacial. A modelagem, tipos e funções utilizadas poderiam ser reestruturadas para tratar desses dados.

Uma modelagem bem feita é o primeiro passo para a concepção de um banco de dados. Este processo visa conservar a integridade dos dados, especificar os tipos e a organização dos dados, preparar os dados para serem utilizados em um SGBD, entre outros (CODD, 1980). O tipo de modelagem de dados utilizada pelo Terralab é a Modelagem Relacional. O conceito relacional dos dados fora proposto por Codd nos anos 80 (CODD, 1983). Além de uma modelagem bem desenvolvida, a tipagem dos dados é muito importante para uma boa representação de um problema.

Em (CÂMARA; MEDEIROS, 2005) somos apresentados aos diferentes tipos de dados que uma informação pode assumir num SGBD. Pontos, polígonos e linhas encapsulam os tipos mais básicos, inteiro e real, assumindo tipos autoexplicativos. Ao invés de ter-se apenas vários números armazenados ao acaso, é mais fácil identificar, por exemplo, dois números sendo a representação de um ponto se eles tiverem a identificação Ponto. A determinação desses dados e seu encapsulamento facilita a modelagem dos bancos de dados e sua utilização em funções.

Os bancos de dados espaciais necessitam de operações que diferem dos dados tradicionais, para otimizar e facilitar o seu uso. Um SGBD tradicional trata basicamente de números contínuos e *strings*. Ordenar esses valores, selecionar o maior ou menor, agrupar os dados são feitos de forma totalmente diferente do que aquela necessária para os dados espaciais. Os dados ordinais podem ser classificados como maior ou menor, pode-se somá-los, subtraí-los, e muito mais. Quando trata-se de pontos, polígonos, não é possível aplicar as mesmas operações ou agrupar estes dados sem um critério (CHOPRA, 2010).

O SGBD espacial traz soluções para este problema. Com operações de pertinência, cobertura, distância, é possível estabelecer relações entre os dados armazenados, sendo possível separar aqueles que satisfazem a um determinado critério, por exemplo: Identifique todos os objetos a até 10 unidades de distância do objeto X ou Identifique todos os dados que estão dentro do objeto Y. Sendo assim, as aplicações trazem, nativamente, os recursos necessários para responder as perguntas do Terralab: O ponto A está dentro da cidade Y? Qual a distância entre os pontos A e B? Tendo todas as distâncias, quais aplicações diferiram mais?

Possuir essas identificações permite a utilização de um mecanismo muito importante nos bancos espaciais: as indexações. Responsáveis por garantir divisões nos dados e diminuir o tempo gasto em busca de dados, as indexações são indispensáveis quando trabalha-se com uma grande quantidade de dados espaciais. Assim como os números e letras podem ser divididos em intervalos (1 a 100, ‘a’ a ‘h’), os objetos espaciais podem ser divididos de acordo com o espaço que os contêm; ao criar delimitações no espaço, pode-se incluir nas delimitações os objetos presentes na mesma (GÜTING, 1994).

Aprender, utilizar e explorar esses conceitos podem trazer grandes benefícios aos armazenamentos espaciais. Para isso, será utilizado o SGBD Espacial PostGIS, que já conta com representações de dados, indexação e funções para dados espaciais de forma nativa.

O PostGIS faz parte do maior sistema de gerenciamento em código aberto do mundo, o PostgreSQL. PostGIS, além de gratuito, conta com mais de 10 tipos de dados para representar informações, incluindo “Points”, “LineStrings”, “Polygons”, “MultiPoints”, “MultiLineStrings”, “MultiPolygons” e “GeometryCollections”. Possui 16 funções que relacionam dados (“Contains”, “Equal”, “Cover”, “Within”, etc) e mais de 50 funções para medição, sobreposição, processamento de geometria e muito mais.

1.1 Objetivos

O objetivo deste trabalho é mostrar que utilizar um banco de dados geográfico vai deixar a execução da operações do Terralab significativamente mais rápidas e com menor custo de espaço quando comparada com o sistema de armazenamento atual.

Para o aprimoramento dos processos de análise, leitura e escrita, decidiu-se aplicar um

SGBD que comportasse dados espaciais. A partir disso, o trabalho tem como objetivo, utilizando os dados do TerraLab, também:

- Objetivo específicos:
  - Modelar os dados para representação espacial;
  - Fazer uso de SGBDG (PostGis) para implementação do novo modelo;
  - Verificar o uso de espaço de armazenamento, tempo de leitura, tempo de processamento (distância entre os pontos e verificação de ponto dentro de cidade) e tempo de escrita na nova modelagem.

2 Bancos de Dados

O armazenamento de informação sempre foi uma prática necessária em grandes processos. Seja no inventário de uma loja ou nas fichas de um paciente no hospital, armazenar, organizar, buscar e processar estas informações são importantes para os mantenedores das mesmas. Com o advento da informática, surge uma forma digital de armazenamento: os banco de dados.

Banco de dados são representações de informações do mundo real. São criados a partir da necessidade de se guardar e processar um grande volume de informação. Para que um banco de dados seja eficiente, ele deve seguir uma estrutura fiel ao universo de sua origem. O nome dado a estas estruturas é Modelo de Banco de Dados. A modelagem é o primeiro e mais importante processo na construção de um banco de dados. É a partir dele que são derivadas as estruturas do banco de dados e onde podem ser feitas as primeiras otimizações no armazenamento (HARRINGTON, 2016).

Existem diversas formas de se modelar um banco de dados, como: Modelo Relacional, Modelo Orientado a Objetos, Modelo de Documentos, Modelo de Coluna. O tipo de modelagem utilizada atualmente no objeto de estudo será a mesma proposta para a nova solução: o Modelo Relacional. Como trata-se de um problema onde possuímos os mesmos atributos para cada dado armazenado e a existência de relações entre os pontos, cidades e estados, o modelo tende a ser uma boa opção para a representação dos dado (HARRINGTON, 2016)s.

Modelo relacional organiza os dados em tabelas (ou relações) que consistem em linhas (tuplas) e colunas (atributos). Ele utiliza chaves primárias para identificar unicamente cada linha e chaves estrangeiras para representar relacionamentos entre tabelas (HARRINGTON, 2016).

Relações podem ser facilmente entendidas como tabelas, que representam um, e somente um, aspecto particular de um problema, por exemplo, em uma loja as fichas de todos os clientes representam uma tabela, uma relação. Elas contêm os atributos, que são características daquela relação, em que um cliente pode ter: nome, endereço, telefone, entre outros (HARRINGTON, 2016).

Como algumas características podem coincidir entre um cliente e outro, as chaves primárias são introduzidas para trazer distinção entre todos os clientes registrados. Elas são um atributo de valor único para cada cliente, logo, mesmo que mais de um cliente tenha o mesmo nome, as chaves primárias os distinguem (HARRINGTON, 2016).

Também temos as chaves estrangeiras que são responsáveis por identificar um atributo que é apresentado em outra relação. Por exemplo, na loja pode haver uma relação que represente o histórico de vendas, de modo que ela possua os atributos: valor da venda, produto vendido, hora e cliente. Entretanto, para não haver a necessidade de colocar todas as informações de clientes

nesta relação, pode ser informado apenas qual foi o cliente, da relação cliente, responsável por esta compra. Como a relação Cliente possui uma chave primária que distingue todos os clientes, utiliza-se desta chave como Chave Estrangeira para representar os clientes em outras relações (HARRINGTON, 2016).

Os dados de um banco precisam ser armazenados, organizados, atualizados, excluídos, etc. Para isso, os Sistemas de Gerenciamento de Banco de Dados (SGBD) foram desenvolvidos. Estes sistemas contam com mecanismos para trabalhar com todos os dados contidos em um banco de dados. Diferentes SGBDs possuem diferentes funcionalidades implementadas. Para o propósito deste estudo, foi desenvolvido um banco de dados espacial utilizado um Sistema de Gerenciamento de Banco de Dados Espaciais (SGBDE) que possuísse ferramentas para operar sobre o mesmo.

Os bancos espaciais são caracterizados por fazerem uso de conceitos e representações espaciais. Dados como relevo, vegetação, coordenadas, locais, temperatura, pontos, rotas e espaços são recursos presentes nos banco espacias(CASANOVA et al., 2005). Os SGBDEs permitem implementar estes bancos e dispõe de operações que facilitam a escrita, leitura e processamento dos dados. Com isso, serão analisadas as principais características do SGBDE Postgis.

PostGis é a extensão espacial do SGBD PostgresSQL, considerado o maior banco de dados relacional de código aberto do mundo. A escolha dessa ferramenta se deve ao seu fácil acesso, à quantidade de conteúdo disponível e ao fato de ser uma extensão da atual base de dados utilizada.

2.1 Tipos de dados

Um banco de dados bem estruturado conta com representações de dados que sejam os mais fiéis possíveis à informação que este dado carrega. Isso facilita a leitura, manipulação e operação dos dados por um analista. Como têm-se um conjunto de dados espaciais, deve-se buscar uma forma que melhor represente as informações (coordenadas, malhas de cidades e estados).

Os tipos de dados estão presentes em todo processo de programação com representação de dados. Os mais básicos e sempre encontrados são o binário, booleano, decimal, inteiro, real, string. Estes facilitam a distinção das informações, além de impor restrições e requisitos para os dados representados. Todavia, os dados espaciais não conseguem ser representados diretamente por esses tipos; por exemplo, se forem colocados dez valores decimais agrupados, não é possível afirmar que eles delimitam uma forma bidimensional ou tridimensional, ou quantas formas eles delimitam, por isso, estabelecer uma classificação para este dado facilita sua interpretação e seu uso.

Pontos são pares de coordenadas x e y (abscissas e ordenadas) que podem ser utilizados para representar uma localização, delimitar polígonos, segmentos de retas ou formas não definidas.

Polígonos, por sua vez, representam áreas bidimensionais, sejam divisões políticas, espaçamento ambiental, formas geométricas, etc. Podem ser:

1. Simples - Se os limites forem apenas um conjuntos de linhas;
2. Convexos - Se qualquer segmento formado por dois pontos do polígono estiverem totalmente incluídos no polígono;
3. Formas disjuntas - Por exemplo: um conjunto de ilhas ou enclaves.

Para criar a nova modelagem dos dados, busca-se duas representações, uma que expresse pontos de coordenada (Point e Multipoints) e uma que expresse as delimitações (malha) de uma cidade (Multipolygon). Vale ressaltar que as representações de polígonos precisam representar polígonos vazados (com uma abertura em seu interior) devido ao fato de que algumas localidades do Brasil possuírem esta característica (vide Goiás).

Na Figura 2.1 têm-se os tipos de dados espaciais presentes no PostGis que satisfazem os requesitos previamente estavbelecidos. O SGBDE apresenta novos tipos de dados para poder incorporar e abstrair as informações dos dados espaciais, permitindo que possamos utilizá-los para análises e operações.

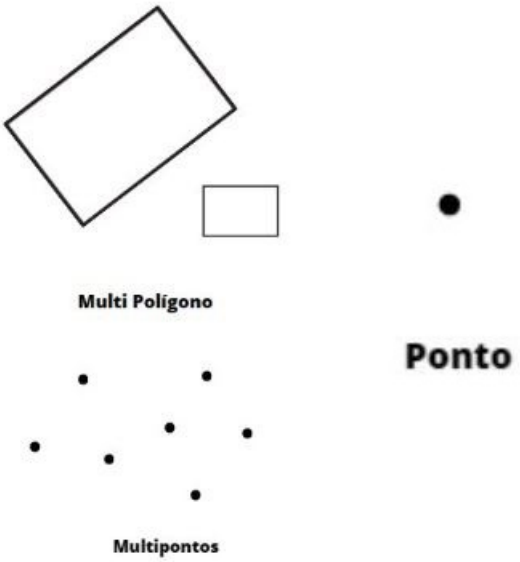

Figura 2.1 – Tipos de Dados PostGis

- Pontos - Uma geometria com dimensão 0;
  - POINT(NUMERIC NUMERIC);

- Multipontos - Um conjunto de pontos;
  - MULTIPOINT ((NUMERIC NUMERIC), (NUMERIC NUMERIC));
- Multi-Polígono - Um conjunto de polígonos não adjacentes delimitado por um conjunto de pontos/pares de valores numéricos.
  - MULTIPOLYGON(((NUMERIC NUMERIC, NUMERIC NUMERIC, NUMERIC NUMERIC,...)) , (NUMERIC NUMERIC, NUMERIC NUMERIC,...), ...);

2.2 Operações

Os conceitos e relações dos dados geográficos diferem dos dados alfanuméricos. Esses conceitos são utilizados para a coleta, armazenamento e análise dos dados. Uma prática comum em sistemas de gerenciamento é a implementação de operações úteis aos tipos de dados armazenados. Estas são as operações de conjuntos.

Estas operações são as mais comuns em contextos espaciais, como união, intersecção e pertinência. Elas podem ser topológicas ou não. Topológicas são as propriedades que não perdem suas características com a variação dos corpos envolvidos, por exemplo: uma floresta dentro de uma cidade continua dentro dela mesmo se a cidade seja ampliada ou deformada. Distância e área são características não topológicas, pois variam se o espaço for deformado.

Como alvo de estudo deste trabalho, serão utilizadas operações topológicas (pertinência para validação dos pontos) e não topológicas (distância para qualificar os pontos). No PostGis têm-se as seguintes funções:

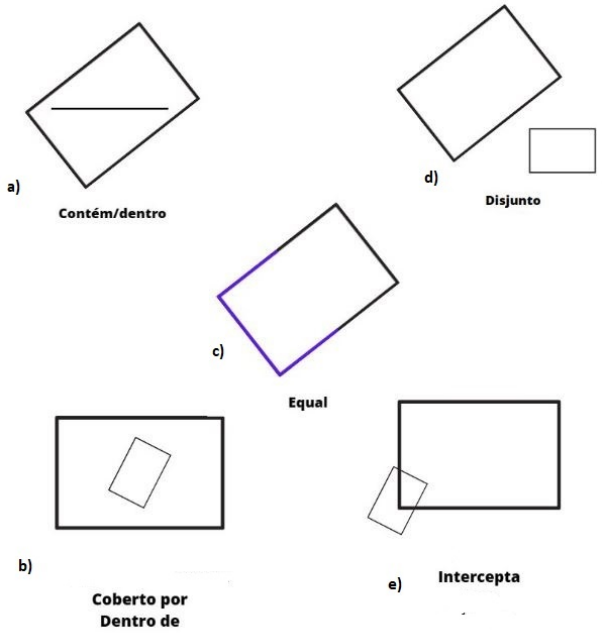

Figura 2.2 – Operações disponíveis no PostGis

- Contém - Um objeto A contem o objeto B (Figura 2.2.a)
  - $ST\_Contains(A, B) \Leftrightarrow (A \cap B = B) \wedge (Int(A) \cap Int(B) \neq \emptyset)$;
  - ST_Contains(geometry geomA, geometry geomB);
- Coberto por - Verifica de um objeto A é coberto por um objeto B (Para qualquer direção de A, encontra-se B) 2.2.b);
  - $ST\_CoveredBy(A, B) \Leftrightarrow A \cap B = A$;
  - ST_CoveredBy(geometry geomA, geometry geomB);
- Disjunto - Verifica se A e B não se cruzam 2.2.d);
  - $ST\_Disjoint(A, B) \Leftrightarrow A \cap B = \emptyset$;
  - ST_Disjoint(geometry A , geometry B);
- Intercepta - Verifica se dois objetos possuem um ponto em comum 2.2.e);
  - $ST\_Intersects(A, B) \Leftrightarrow A \cap B \neq \emptyset$;
  - ST_Intersects(geometry geomA , geometry geomB);
- Dentro de - Verifica se um objeto A está coberto por um objeto B 2.2.b).
  - $ST\_Within(A, B) \Leftrightarrow (A \cap B = A) \wedge (Int(A) \cap Int(B) \neq \emptyset)$;
  - ST_Within(geometry A, geometry B);

Para poder organizar os dados de forma que as operações tenham maior eficiência e sejam aplicadas ao menor número de dados possível e necessário, pode-se fazer o uso de indexações.

2.3 Indexação

Índices são mecanismos úteis para acelerar o processo de busca de dados. Quando se lida com um grande volume de dados, ou quando esses dados são identificados por uma junção de várias informações ou quando a relação entre estes dados não é facilmente inferida (ex. número ordinais), o uso de índices é muito útil(SHEKHAR; CHAWLA, 2003).

Os índices podem ser construídos de diversas formas mas sua utilização segue o mesmo princípio: criar identificadores únicos para cada informação e possibilitar estabelecer relações entre estes identificadores como: maior, menor, se pertencem a um mesmo grupo X ou não, dividir em grupos menores.

Existem duas formas principais de criar índices espaciais: adicionar estruturas espaciais extras dedicadas e mapear os objetos espaciais em um espaço bidimensional.

2.3.1 Grid file

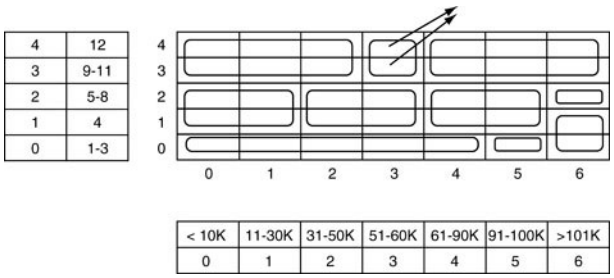

Figura 2.3 – Grid File

O conceito de Grid-file foi apresentado por (NIEVERGELT; HINTERBERGER; SEVCIK, 1984). Seu funcionamento consiste na divisão de um espaço em uma grade regular (grid), onde cada partição pode conter um ou mais pontos de dados. Cada célula pode, mas não necessariamente, ser dividida em partições menores, o que pode eliminar a ocorrência de células vazias e evitar células com grandes quantidades de dados. Cada divisão possui seus intervalos de dados que são mapeados por um vetor onde cada entrada deste vetor configura um intervalo no Grid.

Nesta indexação temos duas etapas para localizar um dado, encontrar seu intervalo no vetor e o valor dentro da divisão. A busca no vetor possui complexidade O(1), pois o acesso é feito diretamente na chave identificadora e a busca nas células O(n), sendo n o número de dados daquela célula.

Suas vantagens estão na possibilidade de subdivisões dinâmicas, podendo evitar espaços vazios e nas buscas por regiões do espaço, como "buscar por todos os dados dentro da região 1".

A distribuição desigual dos dados e o grande número de divisões, podem ser fatores que prejudicam a eficiência desta indexação, já que podem gerar uma estrutura difícil de gerenciar e com alto custo de espaço de armazenamento.

2.3.2 Quad-tree

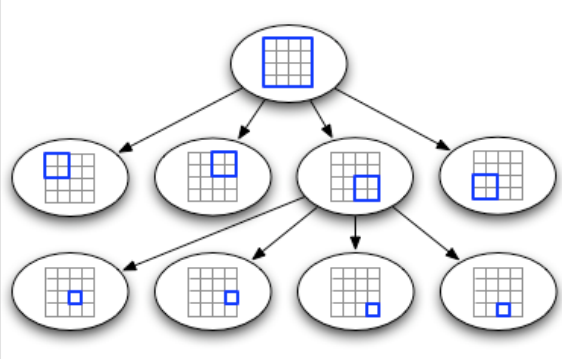

Figura 2.4 – Árvore de quadrante

Proposta por (FINKEL; BENTLEY, 1974), a árvore de quadrante, ou quad-tree, é um modelo de indexação para dados bidimensionais, muito eficiente para lidar com dados hierárquicos. Nesta estrutura, o espaço inicial é dividido em quatro regiões de tamanho igual. Cada partição possui um limite de dados que podem ser armazenados. Quando esse limite é excedido, a partição é subdividida em quatro novas regiões.

Caso a árvore seja balanceada, ou seja, se para qualquer nó, a altura de suas duas subárvores diferem de no máximo uma unidade, a busca dos dados possui complexidade de O($\log_2 n$). Caso contrário, no pior dos casos, a busca teria complexidade O(n).

A definição de um limite de dados faz com que regiões com poucos dados não sejam divididas desnecessariamente e facilita o armazenamento de grande volume de dados. Suas buscas de dados em áreas específicas também são facilitadas pela sua menor quantidade de comparação. Todavia, se os dados estiverem concentrados em uma pequena área, isto leva a uma árvore desbalanceada que prejudica sua manipulação. A estrutura ainda é limitada a dados bidimensionais, não sendo possível lidar com mais de duas dimensões.

2.3.3 R-tree

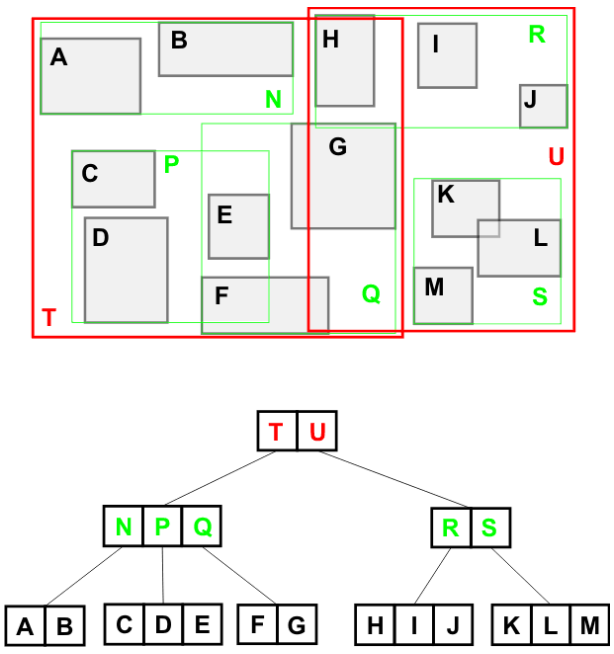

Figura 2.5 – Árvore R

Introduzida por (GUTTMAN, 1984) R-tree é uma estrutura para dados multidimensionais como coordenadas, polígonos, etc. Na árvore R, cada nó interno contém um conjunto de retângulos mínimos delimitadores - MBR (Minimum Bounding Rectangle). O MBR de cada nó é o menor retângulo possível que abrange todos os MBRs de seus filhos. Cada nó externo (ou nó folha) contém os MBRs que englobam os objetos reais, como pontos, polígonos ou segmentos de retas.

Cada nó possui um número mínimo e máximo de MBRs. Quando a quantidade é excedida, o nó é dividido em novos MBRs.

A busca na árvore R parte do nó raiz e percorre os MRBs que cobrem a mesma região que a desejada. Isso limita a busca apenas às regiões de interesse. Em casos com uma boa distribuição de dados, a busca pode ter complexidade de O($\log_2 n$) mas com dados desbalanceados ou muitas regiões sobrepostas este valor pode aumentar.

A R-tree é eficiente para consultas de interseção, proximidade e inclusão de áreas espaciais. Ao dividir o espaço usando MBRs, minimiza-se a sobreposição entre retângulos, reduzindo assim o número de subárvores que precisam ser exploradas durante as buscas. O problema se encontra na sobreposição de MRBs nos níveis mais altos da árvore, exigindo que vários nós sejam visitados. Portanto, faz-se necessário o uso de uma boa estratégia de divisão dos MRBs para evitar tais problemas.

2.4 Linguagem SQL

Para criação do banco de dados e desenvolvimento de funções, será utilizada a linguagem SQL (Structured Query Language)(ROCKOFF, 2021) que é a linguagem padrão para banco de dados relacionais. É uma linguagem que permite criação, leitura, deleção, alteração, transferência e alteração sobre o controle dos dados. Aceita por quase todos os SGBDs, a SQL se torna uma ferramenta flexível e que ainda permite ao usuário desenvolver funções exclusivas para uso em diferentes bancos de dados.

2.5 Visão - Tabelas Temporárias

Dentro dos SGBDs é possível criar novas “tabelas” para um fim específico (uma pesquisa ou operação). Estas representações podem ser temporárias (Tabelas Temporárias) ou persistentes (Visões). A principal diferença está no uso de espaço e custo de processamento. Visões não armazenam dados, armazenam uma requisição que sempre é executada, tendo maior custo computacional e menor ocupação da memória. As tabelas temporárias armazenam resultados de uma requisição em uma tabela enquanto durar uma sessão, seu uso é recomendado quando uma sub-consulta ou um determinado conjunto de dados é utilizado várias vezes nas consultas. As operações são executadas apenas uma vez e armazenadas, tendo baixo impacto computacional e facilitando sua reutilização (RUSSELL; STEPHENS, 2004).

As visões são comumente utilizadas para acesso de dados, enquanto as tabelas temporárias para processamento de dados. Estes recursos podem apresentar um grande ganho de tempo nas operações de um banco de dados, principalmente se o volume de dados for grande e houver operações que se repetem.

2.6 Trabalhos Relacionados

Com o aumento da representação digital dos dados, desenvolver, aprender e avaliar novos recursos para os bancos de dados é imprescindível. Nesta seção serão exploradas outras opiniões e estudos feitos.

O primeiro passo dentro de uma solução envolvendo branco de dados é a elaboração do esquema de dados, seu relacionamento e o tipo dos dados utilizados. Estudos independentes mostram que até 80% dos dados possuem partes espaciais, sendo informações geográficas ou geométricas, podendo estar em bancos, indústrias de chips, logística e várias outras (SCHNEIDER, 2009). Em seu estudo, Schneider evidencia o valor dos tipos de dados nos modelos, a possibilidade de representar os dados de diversas formas e atribuir operações específicas para os dados espaciais são características atraentes para o uso destes recursos.

Para implementar e padronizar as soluções de banco de dados, a *Open Geospatial Consortium Technical Committee* estabeleceu em (OPEN, 1999) uma série de especificações para como fazê-los na linguagem SQL. Com a linguagem é possível realizar tarefas de leituras, escritas e processamento dos dados de uma forma padronizada, possibilitando seu uso por diferentes desenvolvedores e mantendo as mesmas sintaxes. Todo processo de interação com os dados dependem da forma que essas tarefas são escritas e caso elas não sejam bem formuladas, o custo operacional pode ser muito grande. (SHEKHAR et al., 1999) faz um compilado das partes que compõem um processo de gerenciamento de dados espaciais e elucida a importância de construir um modelo e requisições que tenham boa performance, por exemplo: dada a busca "selecione a maior distância entre todos os endereços que estão na cidade X para o ponto Y", não é necessário calcular a distância de todos os endereços da base para o ponto Y, por isso, a operação que verifica a se um ponto Z pertence a cidade X deve preceder a operação de distância entre Z e Y.

Buscas em banco de dados espaciais são mais complexas do que em banco de dados tradicionais, devido ao fato de que as operações ordinais não são aplicáveis aos tipos espaciais. Sendo assim, a construção das requisições devem ser pensadas para evitar processamento excessivo de dados e realizar junções entre tabelas de forma otimizada. Mella (MELLA et al., 2019) apresenta em seu estudo os impactos gerados pela utilização correta de junções e também de um recurso indispensável em bancos espaciais, as indexações. Todavia, como apresentado no estudo de Mella, nem todas as operações se beneficiam do uso de índices, deve-se analisar a solução e mensurar o ganho que pode-se ter com seu uso.

Em seus estudos, Shekkar (SHEKHAR et al., 1999) apresenta como o uso dos índices podem melhorar um processo e nas análises feitas por Mella (MELLA et al., 2019) e Ooi (OOI; SACKS-DAVIS; HAN, 1993), as indexações geraram ganhos significativos nas base de dados, chegando a até 50%, como mostrado nos estudo de Mella.

Outro recurso importante quando lidando com um grande volume de dados é o uso de visões e tabelas temporárias. A bibliografia descreve os casos de uso para estas ferramentas,

como já citado no capítulo anterior, mas não foram encontrados estudos que apresentassem experimentos com estas abordagens.

Com todas essas informações busca-se apresentar um modelo relacional, registrar tempo e custo de armazenamento considerando os diferentes tipos de dados, funções e indexações. Essa metodologia será melhor descrita no capítulo que se segue.

3 Método

Para criar os bancos de dados que serão usados nos experimentos, adotou-se o modelo relacional. Para isso, far-se-á o uso dos conceitos de Relação, Chave Primária, Chave Estrangeira e Atributos. O modelo proposto será derivado do modelo do TerraLab, que é apresentado na Figura 3.1.

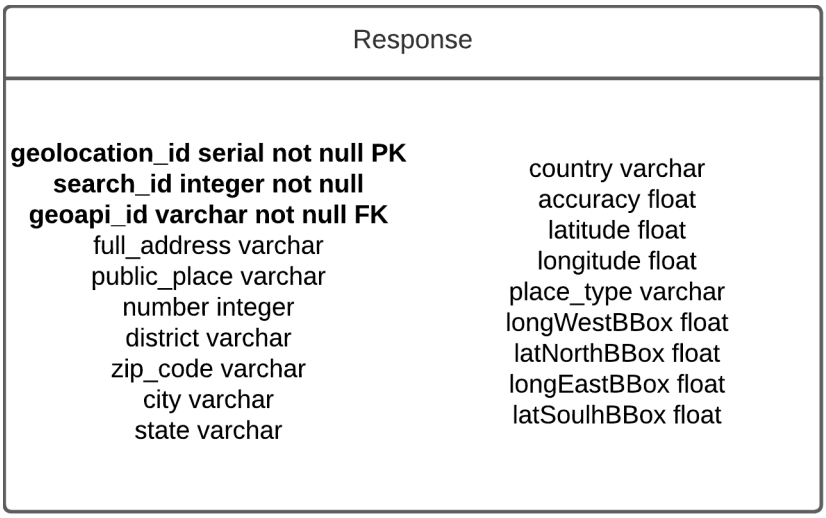

Figura 3.1 – Modelo relacional utilizado pelo Terralab

Podemos observar que todas as informações estão dispostas em apenas uma relação, trazendo repetitividade dos dados. Por exemplo, o atributo 'country' que representa o país de um endereço. Como todos os endereços são coletados apenas no Brasil, não há necessidade de guardar esta informação para cada endereço armazenado. Os atributos longWestBBox, latNorthBBox, longEastBBox e latSouthBBox podem ser removidos, devido ao fato de que eles não são utilizados em nenhum tipo de operação no Terralab, o que melhora o tempo de escrita e poupa espaço.

Os tipos de dados variam entre inteiros (integer), reais(float) e caracteres (varchar), não sendo possível representar tipos mais complexos como linhas, pontos, polígonos, etc, dificultando sua interpretação e utilização em operações. Além disso, o PostgreSQL não conta nativamente com operações espaciais, fazendo com que essas tenham que ser providenciadas pelo usuário e executadas após a leitura dos dados, o mesmo acontece na utilização das malhas das cidades ou estados. Como o banco não contém essas informações, o usuário fica responsável por obtê-las.

O SGBGE PostGIS ainda possui várias formas para representação de dados. Além dos formas mais tradicionais: ponto, linha, polígono, existem outros tipos de dados como multi polígono, superfície, multi pontos que devem ser avaliados para verificar a eficiência de seu uso.

Visando melhorar a eficiência das requisições, foram utilizados índices, que é um recurso essencial para lidar com grande volumes de dados e, principalmente, de dados espaciais. Essas

indexações permitem que as operações e acesso aos dados sejam mais rápidos.

As operações podem ser feitas diretamente no banco de dados, em uma seleção, ou podem ser feitas em uma visão ou tabela temporária. Estes dois últimos recursos possibilitam limitar o universo de busca para apenas os dados relevantes e também a deixarem resultados pré calculados para as requisições.

Com uma representação mais precisa dos dados e um acesso mais rápido, é possível reduzir o tempo gasto nas operações de comparação entre as geoAPIs. Para tentar solucionar esses problemas e apresentar uma solução prática e eficiente, o estudo será dividido em etapas, sendo elas:

- Modelagem - Estabelecer um novo modelo de dados que seja mais prático que o atual;
- Avaliação do tempo de escrita - Verificar o tempo necessário para inserir um grande volume de dados nos bancos desenvolvidos em comparação com o Terralab;
- Avaliação do tempo de leitura - Tempo gasto para leitura de dados no novo modelo em comparação com o Terralab;
- Avaliação do tempo de processamento - Tempo gasto para realizar requisições ao banco de dados aplicando as operações de distância e pertinência em comparação com o Terralab;
- Avaliação do ganho a partir da utilização de novos recursos (índices, visões e tabelas temporárias)

As etapas foram desenvolvidas pelo autor e o ambiente de implementação têm as seguintes especificações técnicas:

- Processador Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz, 2400 Mhz, 4 Núcleo(s), 8 Processador(es) Lógico(s)
- Sistema operacional: Windows 10 - Versão 10.0.19045 Compilação 19045
- 16gb de memória RAM
- 500gb de espaço de armazenamento
- A versão utilizada do PostGreSQL é a versão 14 e do PostGis é a versão 3.3.2.

Serão utilizados 8.393.477 (três milhões) de dados https://drive.google.com/file/d/1foaGSSD74NEEKB4rEzaPA3tZkXnCkcR8/view?usp=sharing) disponibilizados pelo Terralab. As malhas das cidades e estados (objeto 2d que representa os limites destes locais, também chamado de superfície ou polígono) foram retirados do site do Instituto Brasileiro de Geografia e estatística IGBE (https://servicodados.ibge.gov.br/api/docs /malhas?versao=3):

Malha dos estados - https://drive.google.com/file/d/1foaGSSD74NEEKB4rEzaPA3tZkXnCkcR8/view?usp=sharing

Malha das cidades - https://drive.google.com/file/d/1QREj2ziQRRkHHIr6iwOZXs9MzkmnBHHH/view?usp=sharing

Código do IBGE das Cidades e Estados - https://geoftp.ibge.gov.br/organizacao_do_territorio/estrutura_territorial/divisao_territorial/2022/DTB_2022.zip

3.1 Modelagem

O modelo relacional nos é mostrado na Figura 3.2 e a representação obtida através dele no esquema da Figura 3.3. Para o problema, partiu-se da relação Ponto, quais informações um ponto deveria ter? Isso originou os atributos ligados aos Pontos. Informações sobre estado e cidade seriam redundantes dentro desta relação, por haver vários pontos de um mesmo estado e cidade. Daí foram criadas as relações Estados e Cidades. A mesma lógica foi aplicada na criação de API.

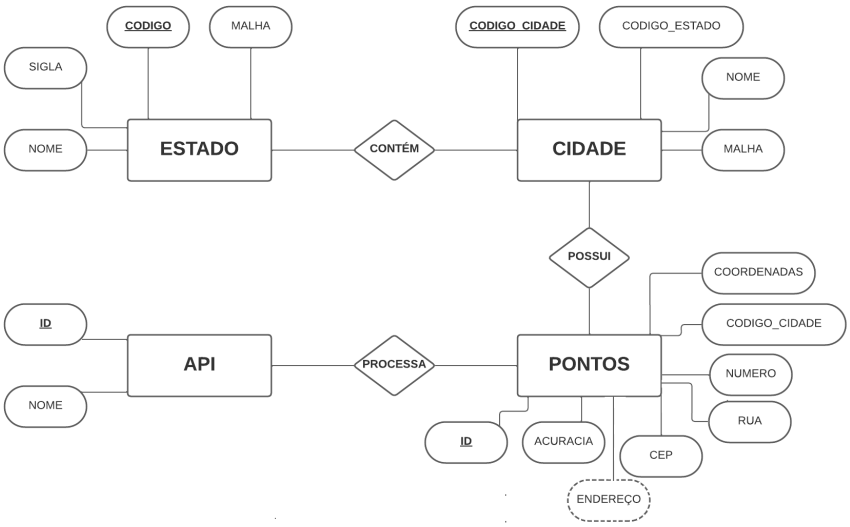

Figura 3.2 – Diagrama Entidade Relacionamento

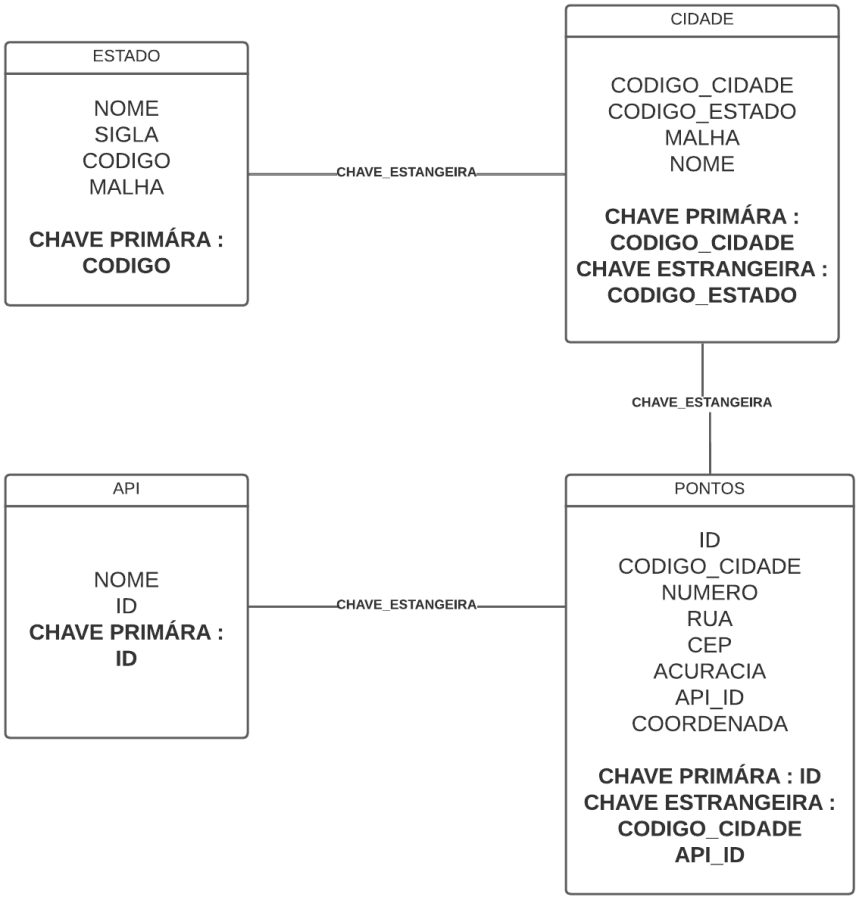

Figura 3.3 – Relações do Modelo

3.1.1 Relação Estado

Esta entidade é responsável por armazenar as principais informações dos estados, como nome, sigla e malha. O código de identificação é provido pelo próprio IBGE. A modelagem é apresentada na tabela 3.1.

Estado		
CODIGO (chave primária)	INT	Atributo de identificação das tuplas
NOME	VARCHAR(20)	Nome do estado
SIGLA	VARCHAR(2)	Abreviação do estado
MALHA	MULTIPOLYGON	Varíavel que guarda as coordenadas que delimitam o espaço do estado

Tabela 3.1 – Estado - PostGis

3.1.2 Relação Cidade

Para abstrair mais informações dos pontos armazenados, a entidade cidade conta com as informações exclusivas do local, contando com código e malha também disponibilizados pelo IBGE. A sua representação está descrita na Tabela 3.2.O atributo “CODIGO_ESTADO” é uma chave estrangeira que referencia a tabela Estado.

Cidade		
CODIGO (chave primária)	INT	Atributo de identificação das tuplas
CODIGO_ESTADO	NUMERIC(2)	Código de referencia do estado
NOME	VARCHAR(50)	Nome da cidade
MALHA	MULTIPOLYGON	Variável que guarda as coordenadas que delimitam o espaço do estado

Tabela 3.2 – Cidade - PostGis

3.1.3 Relação Pontos

A entidade “Pontos” é responsável pela parte mais delicada da modelagem por contar com milhões de dados, em que sua organização precisa ser eficiente e os dados armazenados devem ser somente os necessários, por isso o atributo "endereço completo"é derivado dos dados de endereço. Têm-se para o PostGis o esquema 3.3. Os atributos “CODIGO_CIDADE” e “API_ID” são chaves estrangeiras que referenciam as tabelas Cidade e API, respectivamente.

Pontos		
ID (chave primária)	INT	Atributo de identificação das tuplas
CODIGO_CIDADE	NUMERIC(20)	Código do IBGE da cidade
NUMERO	VARCHAR(40)	Número do endereço do ponto
RUA	VARCHAR(300)	Rua do ponto
CEP	VARCHAR(20)	CEP do ponto
ACURACIA	NUMERIC(4,3)	Valor de precisão que a API de geolocalização retorna
API_ID	NUMERIC(4)	Identificador da API de geolocalização que gerou o ponto
COORDENADAS	POINT	Varíavel que guarda a Latitude e Longitude do ponto

Tabela 3.3 – Pontos - PostGis

3.1.4 Relação API

A representação da API responsável pela geocodificação de um ponto é explicitada nesta entidade. Ela apresenta o nome das APIs e possibilita inserção de novas informações das mesmas. São descritas em 3.4.

API		
ID (chave primária)	NUMERIC(4)	Atributo de identificação das tuplas
NOME	VARCHAR(20)	Nome da API dde geolocalização

Tabela 3.4 – API

3.1.5 Casos de implementação

3.1.5.1 Modelo TerraLab

A implementação do modelo do TerraLab se dá pela requisição 3.1. Nesta requisição cria-se a relação onde todos os dados coletados das APIs são armazenados e não há armazenamento das informações das cidades e estados.

```
1    CREATE TABLE response
2        (   full_address NUMERIC(4),
3            public_place VARCHAR,
4            number VARCHAR,
5            district VARCHAR,
6            zip_code VARCHAR,
7            city VARCHAR,
8            state VARCHAR,
9            country VARCHAR,
10           accuracy NUMERIC(10,2),
11           latitude NUMERIC(20,7),
12           longitude NUMERIC(20,7),
13           place_type VARCHAR,
14           longWestBBox NUMERIC(20,7),
15           latNorthBBox NUMERIC(20,7),
16           longEastBBox NUMERIC(20,7),
17           latSouthBBox NUMERIC(20,7),
18           geolocation_id INTEGER NOT NULL,
19           search_id INTEGER,
20           geoapi_id VARCHAR,
21           key_id INTEGER,
22           request_id INTEGER,
23           CONSTRAINT pk_response PRIMARY KEY (geolocation_id)
24       );
25
```

Algoritimo 3.1 – Criação bando de dados do TerraLab

3.1.5.2 Modelo Proposto

Para implementar o modelo proposto, as seguintes requisições foram utilizadas. Os comandos “CONSTRAINT ___ PRIMARY KEY” são responsáveis por criar a chave primária da relação e “FOREIGN KEY (nome-da-chave) REFERENCES relação (atributo)” cria uma chave estrangeira na tabela utilizada referenciando o atributo da relação informada no comando.

```
1    CREATE DATABASE "banco_espacial"
2
```

Algoritimo 3.2 – Criação do banco de dados

```
1
2    CREATE EXTENSION postgis
3
```

Algoritimo 3.3 – Adição extensão espacial PostGis

```
1
2     CREATE TABLE api
3        (
4            id NUMERIC(4) NOT NULL,
5            nome VARCHAR(20) NOT NULL,
6            CONSTRAINT pk_api PRIMARY KEY (id)
7        )
8
```

Algoritimo 3.4 – Criação relação API

```
1
2     CREATE TABLE estado
3        (
4            codigo NUMERIC(2) NOT NULL,
5            nome VARCHAR(20) NOT NULL,
6            sigla VARCHAR(2) NOT NULL,
7            malha geometry(MultiPolygon),
8                CONSTRAINT pk_estado PRIMARY KEY (codigo)
9        )
10
```

Algoritimo 3.5 – Criação relação Estado

```
1
2    CREATE TABLE cidade
3        (
4            codigo NUMERIC(10) NOT NULL,
5            codigo_estado NUMERIC(2) NOT NULL,
6            nome VARCHAR(50) NOT NULL,
7            malha geometry(MultiPolygon),
8                CONSTRAINT pk_cidade PRIMARY KEY (codigo),
```

```
                FOREIGN KEY (codigo_estado) REFERENCES
                estado (codigo)
        )

```

Algoritimo 3.6 – Criação relação Cidade

```

    CREATE TABLE pontos
        (
            id NUMERIC(28) NOT NULL,
            codigo_cidade NUMERIC(10) NOT NULL,
            numero VARCHAR(40) ,
            rua VARCHAR(300) ,
            cep VARCHAR(20) ,
            acuracia NUMERIC(7,3) ,
            api_id NUMERIC(4) ,
            coordenadas geometry(POINT),
            CONSTRAINT pk_pontos PRIMARY KEY (id),
            FOREIGN KEY (codigo_cidade) REFERENCES
            cidade (codigo)
        )

```

Algoritimo 3.7 – Criação relação Pontos

3.1.6 Inserção

Para inserir os dados nas relações criadas é necessário realizar uma padronização dos dados, sendo: Padronizar os nomes dos estados e realizar um parser (analisador) nos dados referentes às malhas das cidades e estados. A padronização dos estados podem ser feitas da maneira que o desenvolvedor preferir. Os dados das malhas obtidos do IBGE podem estar divididos (como quando uma cidade tem ilhas), sendo assim, devem ser colocadas em uma única string. Para isso, para cada malha obtida do IBGE que será inserida no banco, executar a seguinte lógica. O resultado da execução é a variável “string_malha” que deve ser inserida no banco na devida cidade/estado.

```
def get_limits(code):

    with open('nomedoarquivo.geojson', 'r') as file:
    limites = json.load(file)
    for dado in limites['features']:
        if dado['properties']['codarea'] == str(code):
            return dado
    return None


```

```
malha = get_limits(codigo_do_ibge_da_cidade)
if malha['geometry']['type'] != 'Polygon':
    string_malha = "("
    for poligono in malha['geometry']['coordinates']:
        string_malha += '('
        for par in poligono[0]:
            string_malha += "{} {}, ".format(par[0], par[1])
        string_malha = string_malha[:-2] +'), '
    string_malha = string_malha[:-2] + ')'
else:
    #Verificar se o tipo eh multipolygon mas eh formado por apenas um poligono
    if len(malha['geometry']['coordinates']) > 1:
        string_malha = "("
        for poligono in malha['geometry']['coordinates']:
            string_malha += '('
            for par in poligono:
                string_malha += "{} {}, ".format(par[0], par[1])
            string_malha = string_malha[:-2] +'), '
        string_malha = string_malha[:-2] + ')'
    else:
        string_malha = "("
        for poligono in malha['geometry']['coordinates']:
            string_malha += '('
            for par in poligono:
                string_malha += "{} {}, ".format(par[0], par[1])
            string_malha = string_malha[:-2] +')'
        string_malha += ')'
```

As tabelas API e PONTOS são populadas com os dados disponibilizados pelo TerraLab disponíveis no início desta seção e as tabelas ESTADO e CIDADE, pelas malhas apresentadas também no início do capítulo. Os códigos das cidades e estados para relacionar os pontos e localidades estão presentes no arquivo “Código do IBGE das Cidades e Estados”. Para inserir os dados, utilizar as seguintes requisições:

```
    INSERT INTO pontos (
                    id,
                    codigo_cidade,
                    numero,
                    rua,
                    cep,
                    acuracia,
                    api_id,
                    coordenadas
                )
    VALUES (id1,cod1,'numero1','rua1','cep1',acuracia1,idApi,'POINT(x y)'),
```

```
    (id2,cod2,'numero2','rua2','cep2',acuracia2,idApi,'POINT(x y)'),
    (id3,cod3,'numero3','rua3','cep3',acuracia3,idApi,'POINT(x y)'), ...
    ;

```

Algoritimo 3.8 – Inserção na relação Pontos

```
INSERT INTO api (
                    id,
                    nome
                )
VALUES (id1,'nome1'), (id2,'nome2'), (id3,'nome3'), ... ;

```

Algoritimo 3.9 – Inserção na relação

```
INSERT INTO cidade (
            codigo,
            nome,
            codigo_estado,
            malha
        )
VALUES (cod1,'nome1','cod1','MULTIPOLYGON(string_malha1)'),
(cod2,'nome2','cod2','MULTIPOLYGON(string_malha1)'),
(cod3,'nome3','cod3','MULTIPOLYGON(string_malha1)'), ... ;

```

Algoritimo 3.10 – Inserção na relação Cidades

```
INSERT INTO estado (
                    codigo,
                    nome,
                    sigla,
                    malha
                )
VALUES (cod1,'nome1','sigla1','MULTIPOLYGON(string_malha1)'),
(cod2,'nome2','sigla2','MULTIPOLYGON(string_malha2)'),
(cod3,'nome3','sigla3','MULTIPOLYGON(string_malha3)'), ... ;

```

Algoritimo 3.11 – Inserção na relação

3.2 Operações

3.2.1 Caso 1 - Distância para ponto médio

Esta operação visa encontrar como as APIs se comportam em relação às outras, endereços que possuem um valor baixo de distância tendem a ser uma geolocalização confiável. Seu cálculo

é feito da seguinte forma: dado os resultados X da geolocalização de um endereço Y para as N APIs, calcular o ponto médio H dos resultados e a distância de cada X para H.

A implementação no PostgreSQL se dá da seguinte forma: Os dados são recuperados da base de dados pela seguinte requisição:

```
1         SELECT * FROM Response
2
```

Algoritimo 3.12 – Seleção de dados no PostgreSQL

Os dados então são aplicados a uma função em Python desenvolvida pelo TerraLab. A primeira operação realizada é a ordenação dos dados pelo endereço completo, em seguida, utiliza-se um script em C++ para gerar os cálculos mais rapidamente. As etapas para desenvolvimento em C++ são:

- Carregar informações em lote de 300 endereços
- Salvar em um vetor os dado agrupados por endereços
- Gerar um arquivo de saída, definindo o cabeçalho dos dados
- Calcular a distância usando a fórmula de Haversine 3.13
- Abrir o resultado no código em Python com resultados

```
1     static double haversine(
2             double lat1, double lon1,
3             double lat2, double lon2
4         ){
5             // distance between latitudes
6             // and longitudes
7             double dLat = (lat2 - lat1) * M_PI / 180.0;
8             double dLon = (lon2 - lon1) * M_PI / 180.0;
9
10            // convert to radians
11            lat1 = (lat1) * M_PI / 180.0;
12            lat2 = (lat2) * M_PI / 180.0;
13
14            // apply formula
15            double a = pow(sin(dLat / 2), 2) +
16                       pow(sin(dLon / 2), 2) *
17                       cos(lat1) * cos(lat2);
18            double rad = 6371;
19            double c = 2 * asin(sqrt(a));
20
21            return rad * c;
22        }
```

```
23
```

Algoritimo 3.13 – Cálculo da distância de Haversine

3.2.1.1 Implementação inicial PostGis

A solução proposta simplifica a aplicação da função em apenas comandos SQL da seguinte forma:

```
1   SELECT P.codigo_cidade, P.rua, P.numero, P.api_id,
2       ST_Distance(P.coordenadas, medio.centro) as distancia
3   FROM Pontos as P
4   LEFT OUTER JOIN
5   (
6       SELECT pp.codigo_cidade, pp.rua, pp.numero,
7       ST_Centroid(
8               ST_Multi(
9                   ST_Union(pp.coordenadas)
10                      )::geometry(MultiPoint, 0)
11                  ) as centro
12      FROM pontos AS pp
13      GROUP BY pp.codigo_cidade, pp.rua, pp.numero
14  ) AS medio ON
15  p.codigo_cidade = medio.codigo_cidade AND
16  p.rua = medio.rua AND
17  p.numero = medio.numero
18
```

Todos os processos de agrupamento de endereços, cálculo de distância e transformação nos dados já são feitos dentro da própria requisição. O ponto central é calculado pela função "ST_Centroid" e a distância dos pontos pela "ST_Distance".

3.2.1.2 Uso de indexação

Observa-se que a requisição possui várias validações de endereço, tendo de assertar o código da cidade, a rua e o número. Isso gera um custo excessivo de comparações, que pode ser simplificada pela criação de um índice para estes dados, dado da seguinte forma:

```
1   CREATE INDEX end ON Pontos (codigo_cidade, rua, numero);
2
```

Algoritimo 3.14 – Criação index END

Isso possibilita que exista apenas uma comparação durante a execução da requisição para a validação do endereço.

Também foi criado uma indexação nos pontos a fim de avaliar o impacto nas operações.

```
CREATE INDEX point_index_grid
ON pontos USING brin (coordenadas)
```

Algoritimo 3.15 – Criação index BRIN

```
CREATE INDEX point_index_quadtree
ON pontos USING spgist (coordenadas)
```

Algoritimo 3.16 – Criação index SPGIST

```
CREATE INDEX point_index_rtree
ON pontos USING gist (coordenadas)
```

Algoritimo 3.17 – Criação index GIST

As requisições continuam a mesma com o uso de índices mas pode-se intermediar com visões e tabelas temporárias.

3.2.1.3 Visões

Visões são uma representação da requisição já existente. O seguinte comando cria a visão para a operação deste Caso.

```
CREATE VIEW dist_pt_med_view AS
SELECT P.codigo_cidade, P.rua, P.numero, P.api_id,
    ST_Distance(P.coordenadas, medio.centro) AS distancia
FROM
    Pontos AS P
LEFT OUTER JOIN
(
    SELECT pp.codigo_cidade, pp.rua, pp.numero,
           ST_Centroid(ST_Multi(ST_Union(pp.coordenadas))
           ::geometry(MultiPoint, 0)) AS centro
    FROM
        Pontos AS pp
    GROUP BY
        pp.codigo_cidade, pp.rua, pp.numero
) AS medio
ON
    P.codigo_cidade = medio.codigo_cidade
    AND P.rua = medio.rua
    AND P.numero = medio.numero
```

Para realizar a consulta dos dados, basta recuperar todas as informações da visão:

```
SELECT * FROM dist_pt_med_view
```

3.2.1.4 Tabelas Temporárias

Assim como as visões, tabelas temporárias replicam as requisições e possibilitam que os dados sejam resgatados pela tabelas. Para cria-las, foram usadas os seguintes comandos:

```
            CREATE TEMP TABLE centroide AS
            SELECT
                pp.codigo_cidade,
                pp.rua,
                pp.numero,
                ST_Centroid(
                        T_Multi(
                            ST_Union(pp.coordenadas)
                                )::geometry(MultiPoint, 0)
                            ) AS centro
            FROM
                Pontos AS pp
            GROUP BY
                pp.codigo_cidade, pp.rua, pp.numero

```

```
        CREATE TEMP TABLE dist_medio AS
        SELECT P.codigo_cidade, P.rua, P.numero, P.api_id,
            ST_Distance(P.coordenadas, C.centro) AS distancia
        FROM
            Pontos AS P
        LEFT JOIN
            centroide AS C ON
            P.codigo_cidade = C.codigo_cidade AND
            P.rua = C.rua AND P.numero = C.numero

```

Para resgatar estes dados, realiza-se a requisição:

```
            SELECT * FROM dist_medio

```

3.2.2 Caso 2 - Distância máxima para o ponto médio

O caso 2 é derivado do caso 1, onde o resultado é agrupado por endereço e resgata-se a maior distância obtida. No TerraLab, após a operação descrita anteriormente, resultado é gerado com o agrupamento dos dados e resgate do maior valor, utilizando a biblioteca do Pandas.

```
    distancia_total = dist_ponto_medio(
    df, full_address,)
    resposta = distancia_total.loc[distancia_total.groupby(
        [full_address])['Valor'].idxmax()]
    resposta = resposta.reset_index(drop=True)

```

```

```

A solução proposta se dá da seguinte forma:

3.2.2.1 Implementação inicial PostGis

A requisição baseia-se na requisição anterior onde, com resultado obtido, agrupa-se por endereço e obtêm-se a maior distância. Vale observar que a sub consulta da requisição 3.18 é idêntica ao caso anterior.

```
with dist_medio as
(
    SELECT P.codigo_cidade, P.rua, P.numero, P.api_id,
    ST_Distance(P.coordenadas, medio.centro) as distancia
    FROM Pontos as P
    LEFT OUTER JOIN
    (
        SELECT pp.codigo_cidade, pp.rua, pp.numero,
        ST_Centroid(ST_Multi(ST_Union(pp.coordenadas))::
        geometry(MultiPoint, 0)) as centro
        FROM pontos AS pp
        GROUP BY pp.codigo_cidade, pp.rua, pp.numero
    ) AS medio ON
    p.codigo_cidade = medio.codigo_cidade AND
    p.rua = medio.rua AND
    p.numero = medio.numero
)

SELECT a.codigo_cidade, a.rua, a.numero, a.api_id, a.distancia
FROM dist_medio a
INNER JOIN
(
SELECT dist_medio.codigo_cidade, dist_medio.rua,
dist_medio.numero, MAX(dist_medio.distancia) as dist_max
FROM dist_medio
GROUP BY 1,2,3
)
b ON a.distancia = b.dist_max AND
a.codigo_cidade = b.codigo_cidade AND
a.numero = b.numero AND a.rua = b.rua

```

Algoritimo 3.18 – Requisição das distâncias máximas para os pontos médios

3.2.2.2 Índices

Para tentar otimizar essa consulta, foram utilizadas as indexações criadas anteriormente (3.14, 3.15, 3.16 e 3.17).

3.2.2.3 Visões

A visão é criada pela requisição elaborada, basta acrescentar o seguinte comando antes da requisição:

```
1              CREATE VIEW dist_max_pt_med_view AS
2
```

3.2.2.4 Tabelas Temporárias

As tabelas temporárias particionam a requisição. A primeira parte é o cálculo dos pontos médios 3.19. Logo após, é possível calcular todas as distâncias para os pontos médios 3.20 e obter o valor máximo 3.21 . Após esta etapa, utiliza-se a requisição 3.22 para obter o resultado final da operação.

```
1     CREATE TEMP TABLE centroide AS
2     SELECT
3         pp.codigo_cidade, pp.rua, pp.numero, ST_Centroid(ST_Multi(
    ST_Union(pp.coordenadas))::geometry(MultiPoint, 0)) AS centro
4     FROM
5         Pontos AS pp
6     GROUP BY
7         pp.codigo_cidade, pp.rua, pp.numero
8
```

Algoritimo 3.19 – Criação da tabela temporária dos pontos médios

```
1     CREATE TEMP TABLE dist_medio AS
2         SELECT
3             P.codigo_cidade, P.rua, P.numero, P.api_id,
4             ST_Distance(P.coordenadas, C.centro) AS distancia
5         FROM
6             Pontos AS P
7         LEFT JOIN
8             centroide AS C ON P.codigo_cidade = C.codigo_cidade AND
9             P.rua = C.rua AND P.numero = C.numero
10
```

Algoritimo 3.20 – Criação da tabela temporária das

```
1     CREATE TEMP TABLE max_distancias AS
2         SELECT
3             codigo_cidade, rua, numero,
4             MAX(distancia) AS dist_max
5         FROM
6             dist_medio
7         GROUP BY
8             codigo_cidade, rua, numero
```

```

```

Algoritimo 3.21 – Criação da tabela temporária das distâncias máximas

```
    SELECT
        dm.codigo_cidade, dm.rua, dm.numero, dm.api_id,
        dm.distancia
    FROM
        dist_medio AS dm
    INNER JOIN
        max_distancias AS md ON
        dm.distancia = md.dist_max AND
        dm.codigo_cidade = md.codigo_cidade AND
        dm.rua = md.rua AND dm.numero = md.numero

```

Algoritimo 3.22 – Seleção dos resultados

3.2.3 Caso 3 - Distância média entre APIs

Para a operação do Caso 3, o processo desenvolvido pelo TerraLab é parecido com os casos 1 e 2, mas o cálculo da distância é entre os pontos gerados pelas APIs.

3.2.3.1 Implementação inicial PostGis

A implementação proposta visa eliminar todo processamento realizado fora do banco de dados feito pela solução utilizada atualmente no laboratório. No algoritmo 3.23 a seguir podemos observar a implementação no banco:

```
SELECT c.codigo_cidade, c.rua, c.numero, AVg(c.dist_api)
FROM
(
   SELECT a.codigo_cidade, a.rua, a.numero, a.api_id, b.api_id,
   ST_Distance(a.coordenadas,b.coordenadas) as dist_api
   FROM Pontos a, Pontos b
   WHERE
   a.codigo_cidade = b.codigo_cidade AND a.rua = b.rua AND
   a.numero = b.numero
) as c
GROUP BY 1,2,3

```

Algoritimo 3.23 – Requisição da distância média entre APIs

3.2.3.2 Índices

Para tentar otimizar essa consulta, serão utilizadas as indexações já apresentadas (3.14, 3.15, 3.16 e 3.17).

3.2.3.3 Visões

Para criar a visão do caso 3, adicionar o código 3.24 à requisição 3.23 e resgatar as informações com o comando 3.25

```
CREATE VIEW dist_med_api_view AS

```

Algoritimo 3.24 – Prefixo para criação da visão

```
SELECT * FROM dist_med_api_view

```

Algoritimo 3.25 – Seleção das distâncias médias entre as APIs a partir da visão

3.2.3.4 Tabelas Temporárias

No caso 3, utiliza-se apenas uma tabela temporária para registrar a requisição, como não há sub consultas ou várias operações, não é necessário a sua divisão. No código 3.26 consta a criação e no 3.27 sua consulta.

```
CREATE TEMPORARY TABLE dist_api_temp AS
SELECT
    a.codigo_cidade, a.rua, a.numero,
    a.api_id AS api_id_a, b.api_id AS api_id_b,
    ST_Distance(a.coordenadas, b.coordenadas) AS dist_api
FROM
    Pontos a
JOIN
    Pontos b
ON
    a.codigo_cidade = b.codigo_cidade
    AND a.rua = b.rua
    AND a.numero = b.numero

```

Algoritimo 3.26 – Criação da tabela temporária das distâncias das APIs

```
SELECT
    codigo_cidade, rua, numero,
    AVG(dist_api) AS avg_dist_api
FROM
    dist_api_temp
GROUP BY
    codigo_cidade, rua, numero

```

Algoritimo 3.27 – Requisição da distância média entre as APIs

3.2.4 Caso 4 - Distância máxima entre APIs

O caso 4 visa encontrar quais APIs mais divergem na resposta de um determinado endereço. No TerraLab, o processo segue a lógica dos casos anteriores, porém registrando as distâncias entre cada uma das APIs. A solução é apresentada nos tópicos subsequentes.

3.2.4.1 Implementação inicial PostGis

A operação do caso 4 possui uma implementação parecida com o caso 3, variando a operação utilizada de média para máximo. No comando 3.28 pode-se observar a solução proposta:

```
SELECT
    c.codigo_cidade, c.rua, c.numero, MAX(c.dist_api)
FROM
    (
        SELECT a.codigo_cidade, a.rua, a.numero,
        a.api_id, b.api_id,
        ST_Distance(a.coordenadas,b.coordenadas) as dist_api
    FROM Pontos a, Pontos b
    WHERE
        a.codigo_cidade = b.codigo_cidade AND
        a.rua = b.rua AND a.numero = b.numero) as c
GROUP BY 1,2,3

```

Algoritimo 3.28 – Requisição da distância máxima entre APIs

3.2.4.2 Índices

Para tentar otimizar essa consulta, será utilizado as mesmas indexações utilizadas anteriormente (3.14, 3.15, 3.16 e 3.17).

3.2.4.3 Visões

A visão do caso 4 é criada adicionando o comando 3.29 e o resgate das informações com o comando 3.30

```
      CREATE VIEW dist_max_api_view AS


```

Algoritimo 3.29 – Prefixo para criação da visão

```
       SELECT * FROM dist_max_api_view

```

Algoritimo 3.30 – Requisição na visão das maiores distâncias entre APIs

3.2.4.4 Tabelas temporárias

O caso 4 é um caso interessante de aplicação das tabelas temporárias devido a existência de sub consultas. Sua criação é dada pelo comando 3.26 e o comando 3.31 resgata os resultados desejados.

```
1    SELECT
2        codigo_cidade, rua, numero,
3        MAX(dist_api) AS max_dist_api
4    FROM
5        dist_api_temp
6    GROUP BY
7        codigo_cidade, rua, numero
8
```

Algoritimo 3.31 – Requisição das distâncias máximas entre as APIs

3.2.5 Caso 5 - Pertinência dos pontos

O caso 5 é o caso mais problemático do TerraLab, exigindo maior processamento e mais recursos do analista. Para realizar a operação, é preciso:

- Carregar dados em um Pandas.DataFrame com a requisição 3.12
- Filtrar os dados e remover colunas desnecessárias
- Abrir um arquivo contendo o nome das cidades do Brasil e o código do IBGE para cada uma delas
- Carregar as malhas das cidades obtidas no IBGE
- Padronizar a string dos estados
- Ordenar os dados pelos endereços
- Transformar o par latitude/longitude em um shapely.geometry.Point
- Transformar os dados agrupados em um GeoPandas.GeoDataFrame
- Identificar o nome da cidade do Endereço
- Resgatar o código do IBGE pelo nome da cidade
- Resgatar a malha da cidade pelo código do IBGE
- Transformar a malha em um objeto GeoPandas.GeoDataFrame
- Calcular a pertinência dos pontos naquela malha

Todos os processos anteriores têm de ser desenvolvidos pelo analista e as transformações e filtros executados todas as vezes. Para evitar este processo, este trabalho propõe a seguinte solução:

3.2.5.1 Implementação inicial PostGis

Como na solução proposta todos os dados estão dentro do banco de dados, podemos gerar os cálculos apenas com a requisição SQL. Ela se dá da seguinte forma:

```
SELECT  COUNT(*), ST_Contains(c.malha, p.coordenadas), c.nome
FROM pontos p, cidade c
WHERE p.codigo_cidade = c.codigo
GROUP BY 3,2

```

Algoritimo 3.32 – Requisição de pertinência

A função “ST_Containis” é responsável pela verificação da pertinência dos pontos, ela pode ser substituída por outras funções equivalentes: ST_within, ST_intersects, not ST_disjoint e ST_coveredby.

3.2.5.2 Índices

Para tentar otimizar essa consulta, será utilizado as mesmas indexações utilizadas anteriormente 3.15, 3.16 e 3.17).

3.2.5.3 Visões

Para cada função utilizada neste caso, deve-se criar uma visão para requisição de seus dados. Basta adicionar o comando 3.33 e substituir o nome da visão e a operação utilizada. A recuperação de dados dar-se-a por 3.34 informando o nome utilizado na visão.

```
    CREATE VIEW pontos_cidade_ST_Intersects_view AS
    SELECT
        COUNT(*) AS total_pontos,
        ST_Intersects(c.malha, p.coordenadas) AS malha_contida,
        c.nome AS nome_cidade
    FROM
        pontos p
    JOIN
        cidade c
    ON
        p.codigo_cidade = c.codigo
    GROUP BY
        c.nome, malha_contida

```

Algoritimo 3.33 – Criação visão de pertinência

```
        SELECT * FROM pontos_cidade_ST_Within_view

```

Algoritimo 3.34 – Requisição de pertinência a partir da visão

3.2.5.4 Tabelas Temporárias

No caso 5, será utilizado 2 tabelas temporárias, tabela 3.35 e tabela 3.36. O objetivo é simplificar a junção com as duas primeiras. A consulta pode ser feita pelo comando 3.37. Para as outras funções de pertinência, basta criar uma tabela com nome diferente utilizando a função desejada.

```
    CREATE TEMPORARY TABLE temp_pontos AS
    SELECT
        codigo_cidade,
        coordenadas
    FROM
        pontos

```

Algoritimo 3.35 – Criação de tabela temporária para os pontos

```
    CREATE TEMPORARY TABLE temp_cidade AS
    SELECT
        codigo, malha, nome
    FROM
        cidade

```

Algoritimo 3.36 – Criação de tabela temporária para as cidades

```
    SELECT
        COUNT(*) AS total_pontos,
        ST_Intersects(p.coordenadas, c.malha) AS malha_contida,
        c.nome AS nome_cidade
    FROM
        temp_pontos p
    JOIN
        temp_cidade c
    ON
        p.codigo_cidade = c.codigo
    GROUP BY
        c.nome, malha_contida

```

Algoritimo 3.37 – Seleção de pertinência pela tabelas temporária

4 Resultados

Para analisar os efeitos das modificações propostas, foram executados testes com todas as funções no novo modelo.

4.1 Análise do tempo de escrita

As requisições desenvolvidas para inserção dos dados seguem a linguagem SQL, o que facilita sua utilização e apresentaram os resultados das Tabelas 4.1 e Tabela 4.2. O PostgreSQL não armazena as malhas das cidades e, ainda sim, na implementação teve um custo de espaço muito maior que o PostGis, de quase 12x seu tamanho. Seu tempo de criação foi mais alto, devido ao fato de seus dados serem menos complexos, o que indica que uma modelagem simples para o problema tende a ser mais rápida mas tem um custo de armazenamento maior que uma solução mais complexa.

Resultados Criação PostgreSQL			
Relação	Custo de espaço	Tempo para escrita	Desvio padrão
Response	5484MB	61.16 seg	16.50 seg

Tabela 4.1 – Resultados da criação do banco em PostgreSQL

Resultados Criação PostGIS			
Relação	Custo de espaço	Tempo para escrita	Desvio padrão
API	24KB	<0.001 seg	<0.001 seg
Estado	704KB	0.05 seg	<0.001 seg
Cidade	36MB	2.74 seg	0.1 seg
Pontos	406MB	93.46 seg	7.33 seg

Tabela 4.2 – Resultados da criação do banco em PostGIS

4.2 Custo de perações no PostgreSQL

Primeiramente, é necessário avaliar o custo das operações iniciais no PostgreSQL, para ter-se a referência de ganho ou perda no tempo de execução.

Todos os pontos são armazenados em uma única tabela e sem tipagem. As malhas de cidades e estados são externas ao banco. Sendo assim, todo processamento é feito pelo consultor do banco. As funções foram desenvolvidas em Python, utilizando funções definidas em uma biblioteca exclusiva do TerraLab. Os resultados das operações estão descritas na tabela A.10.

4.3 Custo de operações no PostGis

Nos estudos do PostGis foram utilizados diferentes mecanismos de banco de dados, como visões, índices e tabelas temporárias. Estes recursos só precisam ser executadas uma vez para poder ser utilizados em várias requisições. Para cada operação será analisado o tempo total das operações (processamento da requisição + tempo de criação dos mecanismos) mas vale ressaltar que em casos reais, onde há várias requisições, o tempo de criação deveria ser considerado apenas uma vez.

4.3.1 Caso 1 - Distância para o ponto médio

O primeiro caso de estudo é a operação que verifica a distância dos diferentes pontos coletados para um mesmo endereço para o ponto médio destes pontos. Na tabela A.2 e figura 4.1 podemos observar o custo desta operação em cada caso de otimização proposto. Tendo como referência o caso 25, o valor atual utilizado pelo TerraLab (181 segundos em média), podemos ver que todos os resultados obtidos possuem um valor menor que o PostgreSQL.

Podemos observar que o uso da indexação END gerou resultados mais baixos na requisição padrão, no uso de visão e no uso de tabelas temporárias. Isso se deve ao fato de que a operação mais custosa realizada é o pareamento dos endereços, que pode ser feito de forma otimizada. Como é um caso em que possuímos sub-consultas, o uso de tabelas temporárias se mostrou eficiente para diminuir ainda mais o tempo da requisição.

O resultado base é o 23, onde apenas com a mudança de arquitetura e tipagem o tempo de execução médio é de 120 segundos, 33% a menos que a referência. Os menores valores obtidos foram com uso de tabelas temporárias. Vale ressaltar que existe um custo para a criação das tabelas temporárias, apresentadas na tabela A.14 e dos índices, da tabela A.11. Então, têm-se:

- Melhor tempo médio para criação da tabela 40.23 segundos (None, DM_TT)
- Tempo para requisição da tabela temporária - 4.71 segundos

Totalizando 44.23 segundos de tempo total de requisição. 75% a menos que o caso referência 25. Como a tabela temporária só precisa ser executada uma vez para sua utilização, um processo que utiliza a mesma requisição várias vezes tende a se beneficiar deste recurso.

4.3.2 Caso 2 - Distância máxima para o ponto médio

A segunda operação analisada é uma extensão da primeira, onde é calculado o máximo a partir dos resultados obtidos anteriormente. O index END segue sendo um recurso útil nas operações e as tabelás temporárias ainda mais, por possuir mais sub-consultas. Por esse motivo, as mesma condições permaneceram, onde o método mais rápido é o da criação das tabelas temporárias sem o uso de indexação, observado na figura 4.2 e tabelas A.1, A.12, A.14 e A.11.

Considerando o tempo total (criação de index, visões e tabelas temporárias) temos o menor tempo no caso 15 com 49.36 segundo de tempo de requisição, 76% a menos que o caso base 25 (213 segundos).

4.3.3 Caso 3 - Distância média entre APIs

O objetivo da operação do caso 4 é encontrar quais endereços possuem uma grande discrepância nos dados coletados. O caso referência é o 25 e o base 23. A forma mais rápida encontrada foi a indexação por END, pelo custo gerado pelo cálculos da verificação de endereços.

O uso de visão gerou os menores resultados devido a sua seleção mais externa não possuir operações entre tabelas, sendo um caso viável para visões, por isso as tabelas temporárias geraram bons resultados mas não os melhores possíveis. Os índices Brin, Gist e SpGist não representam ganhos significativos devido a operação identificar os pontos pelo endereço e não pela sua localização no espaço.

O melhor tempo registrado foi na solução 2, com tempo total aproximado de 43.17 segundos, um ganho de 84% com relação à operação original 25, podendo ser observado na figura 4.4 e tabelas A.4, A.12, A.14 e A.11. Vale ressaltar que o tempo de criação do índice só ocorre uma vez, logo, em uma solução com repetição de requisições, o tempo de criação não interfere.

4.3.4 Caso 4 - Distância máxima entre as APIs

Nesta operação, calcula-se a distância entre os diferentes pontos relativos a um mesmo endereço para poder mensurar quais APIs possuem grandes variações nos endereços. Neste caso, pode-se observar na figura 4.3 e nas tabelas A.3, A.12, A.14 e A.11 os resultados obtidos. O caso do TerraLab é o 25 e o base, 23.

O uso de recursos de otimização segue igual ao caso anterior, como a operação apenas foi modificada de média para máximo, as condições para as otimizações são as mesmas. O caso mais rápido foi com a utilização de visão e a indexação de END (índices criados para representar um endereço). Isso resultou num tempo total de processamento de aproximadamente 43.26 segundos, representando um ganho de mais de 82% para os 556 segundos do caso base 25.

4.3.5 Caso 5 - Pertinência

Os casos de pertinência são os que possuem maior variedade de funções disponíveis para execução, todas gerando o mesmo resultado. Esta é a operação que o processo atual do TerraLab mais tem custos, levando aproximadamente 40 minutos para processar os dados. Vale ressaltar que os cálculos são feitos durante a execução do programa e as informações das cidades e estados, além das funções que realizam os cálculos precisam ser obtidas pelo desenvolvedor

para realização dos cálculos. No modelo proposto, todas as informações e funções necessárias estão contidas no banco de dados.

O caso base é a proposta 23 nas variações.A primeira delas a ser descartada é o “ST_disjoint”, devido a seu custo de realização. As demais possuem resultados parecidos em todas as suas variações. A indexação melhorou o tempo geral das requisições devido ao fato de que os dados são classificados pela sua geolocalização, facilitando as operações de pertinência.

Como se trata de um processo principalmente de seleção dos dados, as visões foram recursos que trouxeram um ganho nas requisições. Por outro lado, as tabelas temporárias geraram um custo mais alto que a operação base, devido a inexistência de sub-consultas ou junções complexas. Isso fez com que a intermediação fosse mais custosa que o processo base.

A obtenção mais rápida dos resultados foi o caso 1 com uso de visão e da indexação BRIN, totalizando aproximadamente 23,55 segundos, quase 99% mais rápido que a operação do TerraLab.

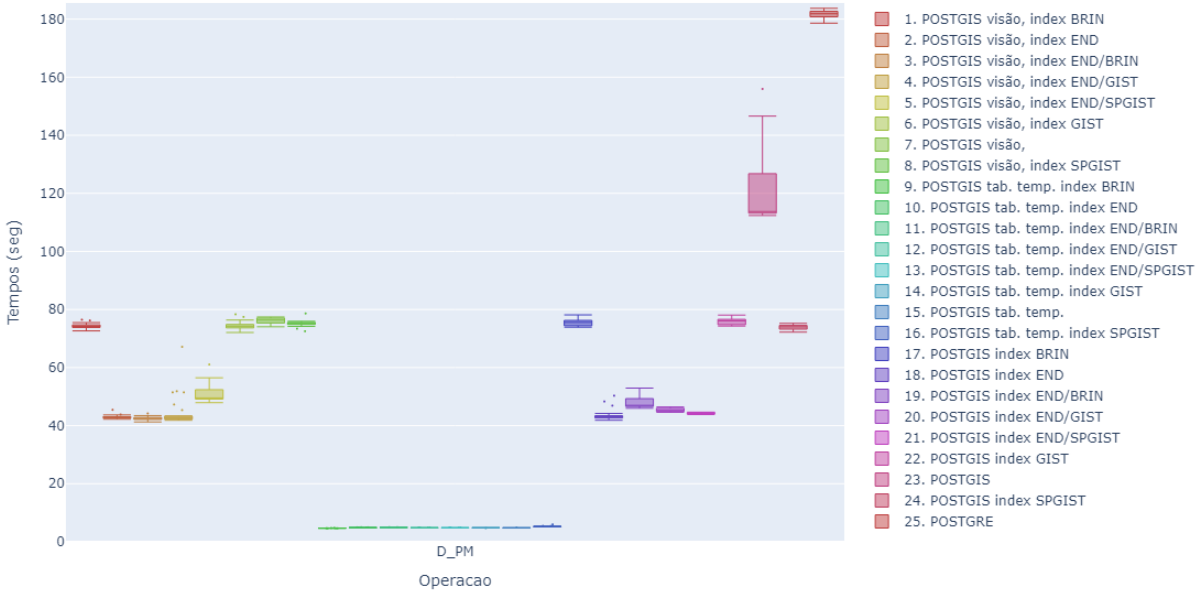

Figura 4.1 – Distância para o ponto médio

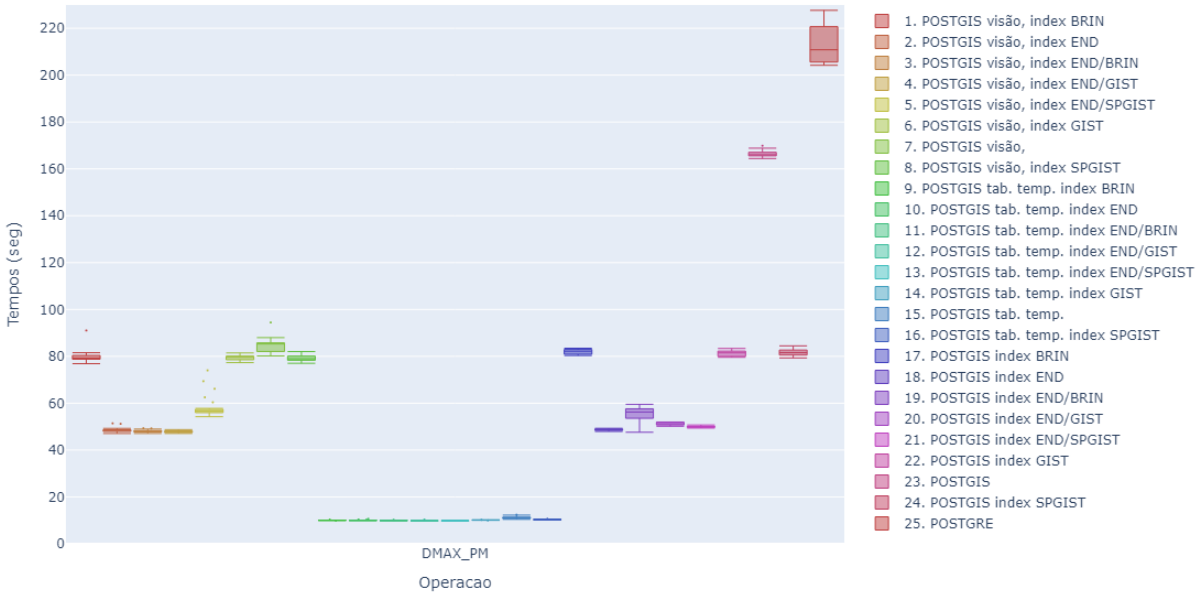

Figura 4.2 – Distância máxima do ponto médio

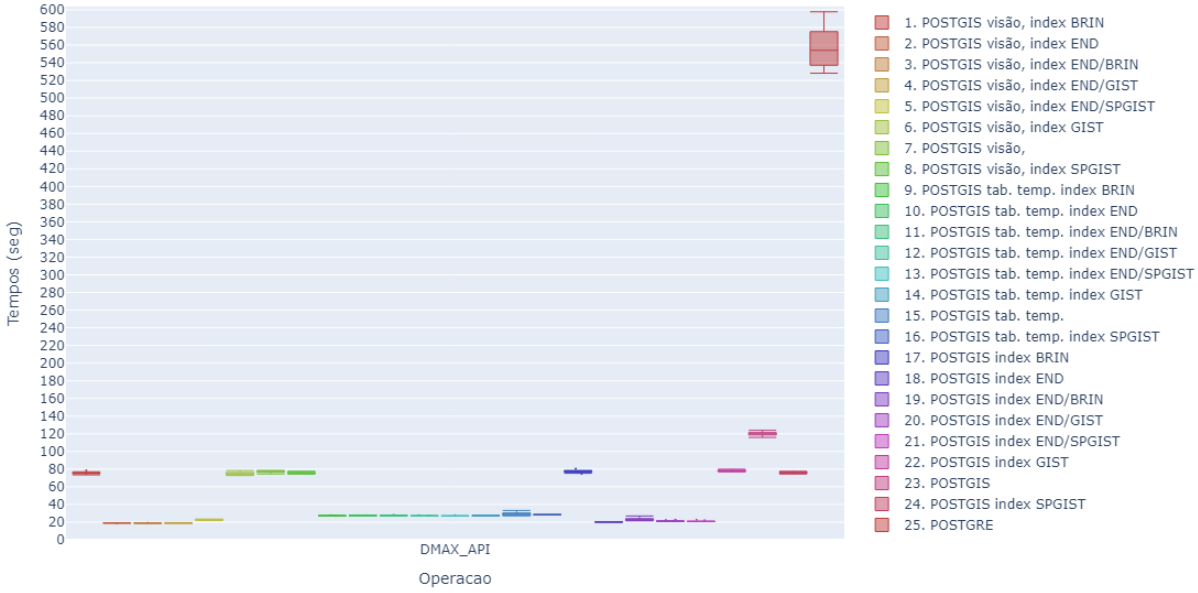

Figura 4.3 – Distância máxima entre as APIs

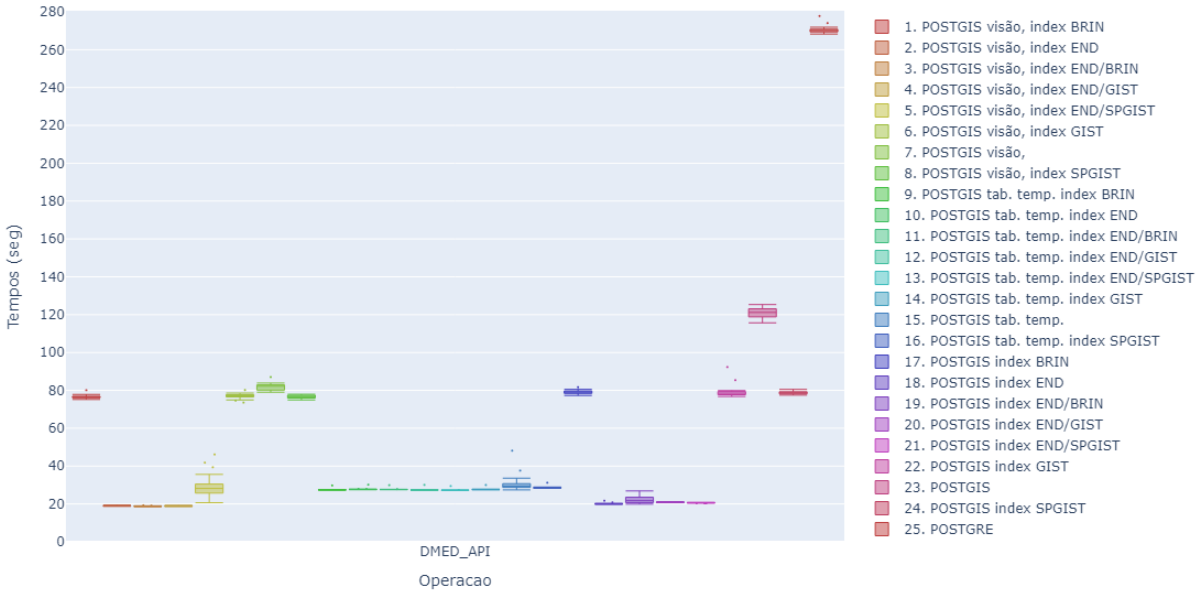

Figura 4.4 – Distância média entre as APIs

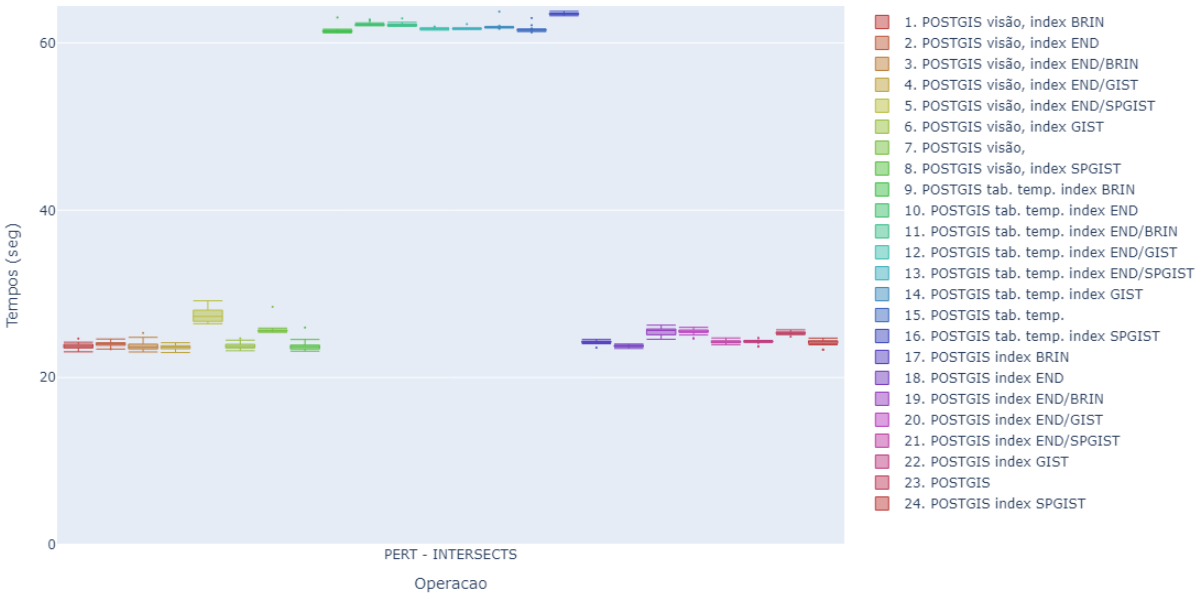

Figura 4.5 – Pertinência - Intersects

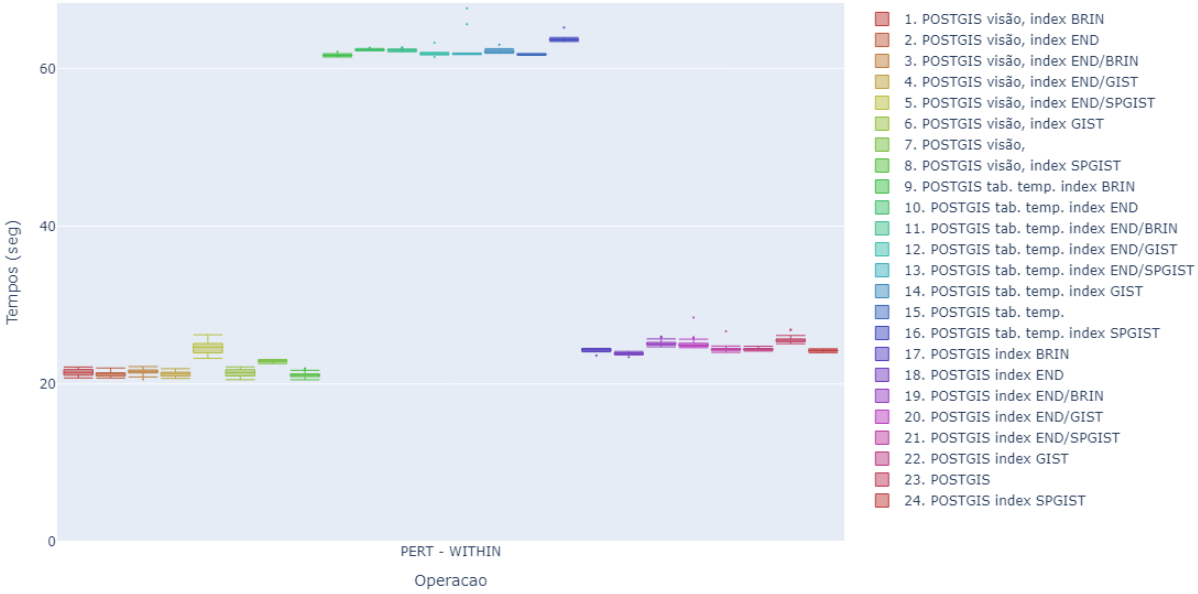

Figura 4.6 – Pertinência - Within

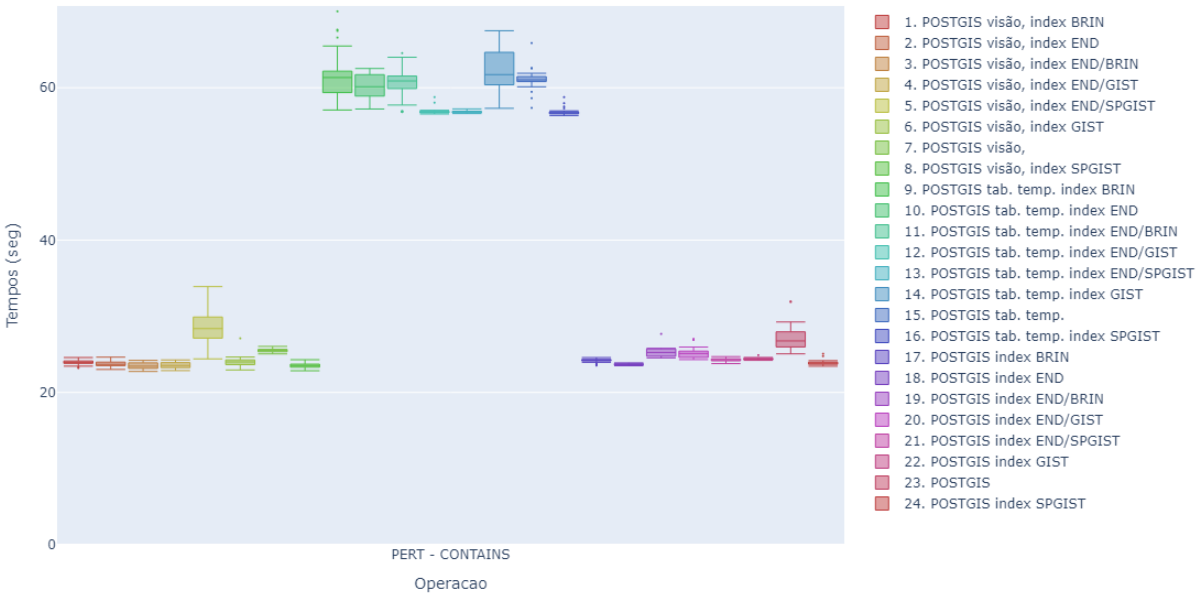

Figura 4.7 – Pertinência - Contains

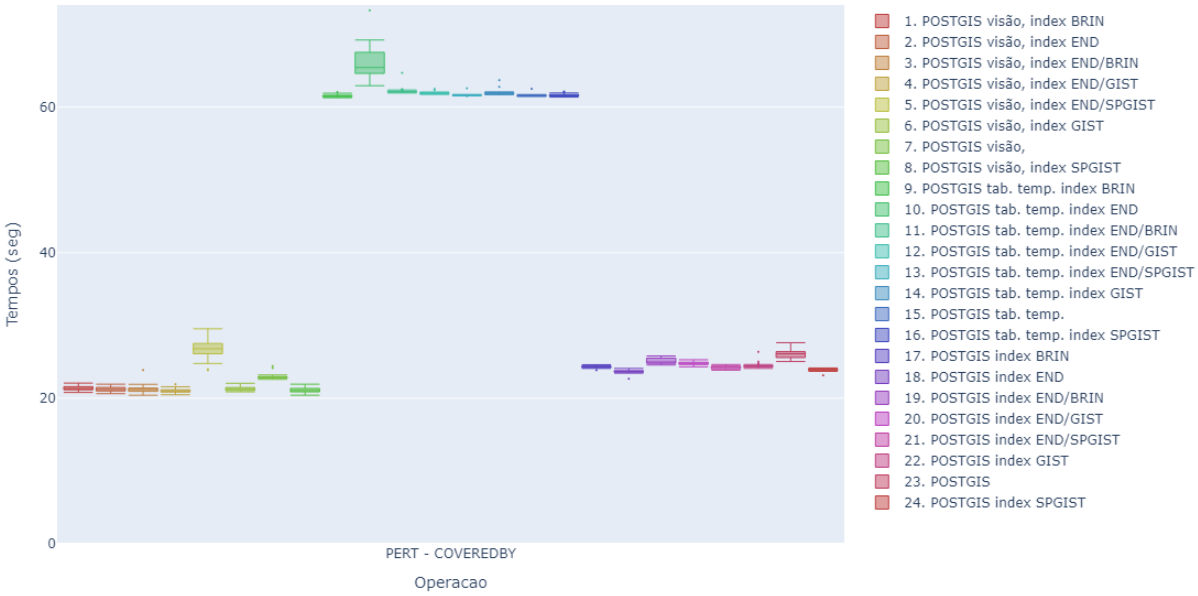

Figura 4.8 – Pertinência - CoveredBy

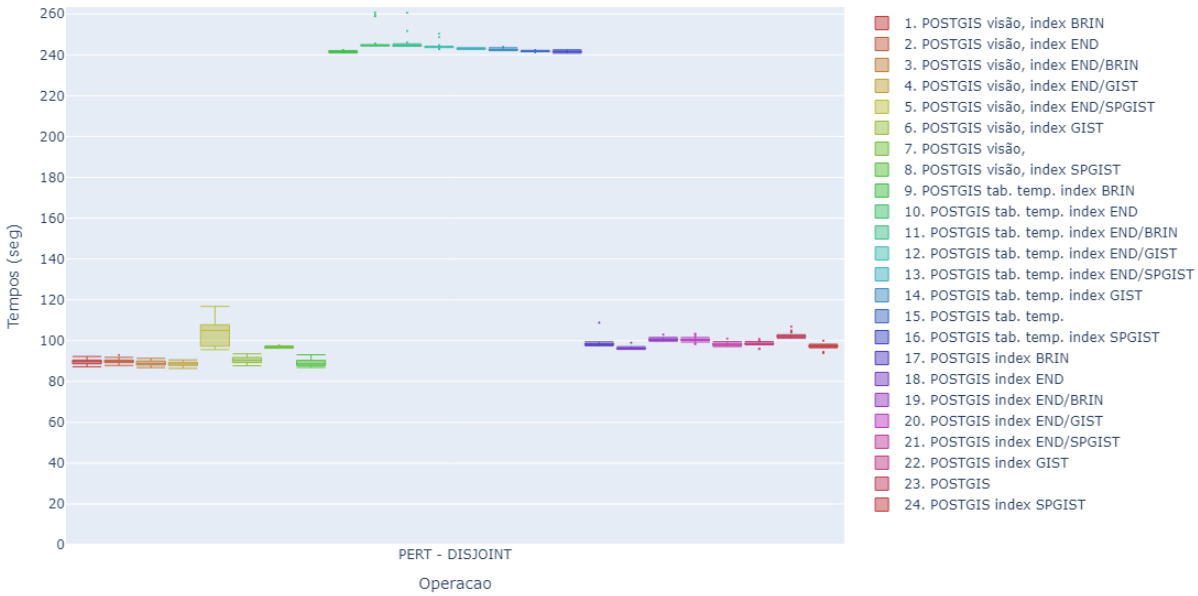

Figura 4.9 – Pertinência - Disjunto

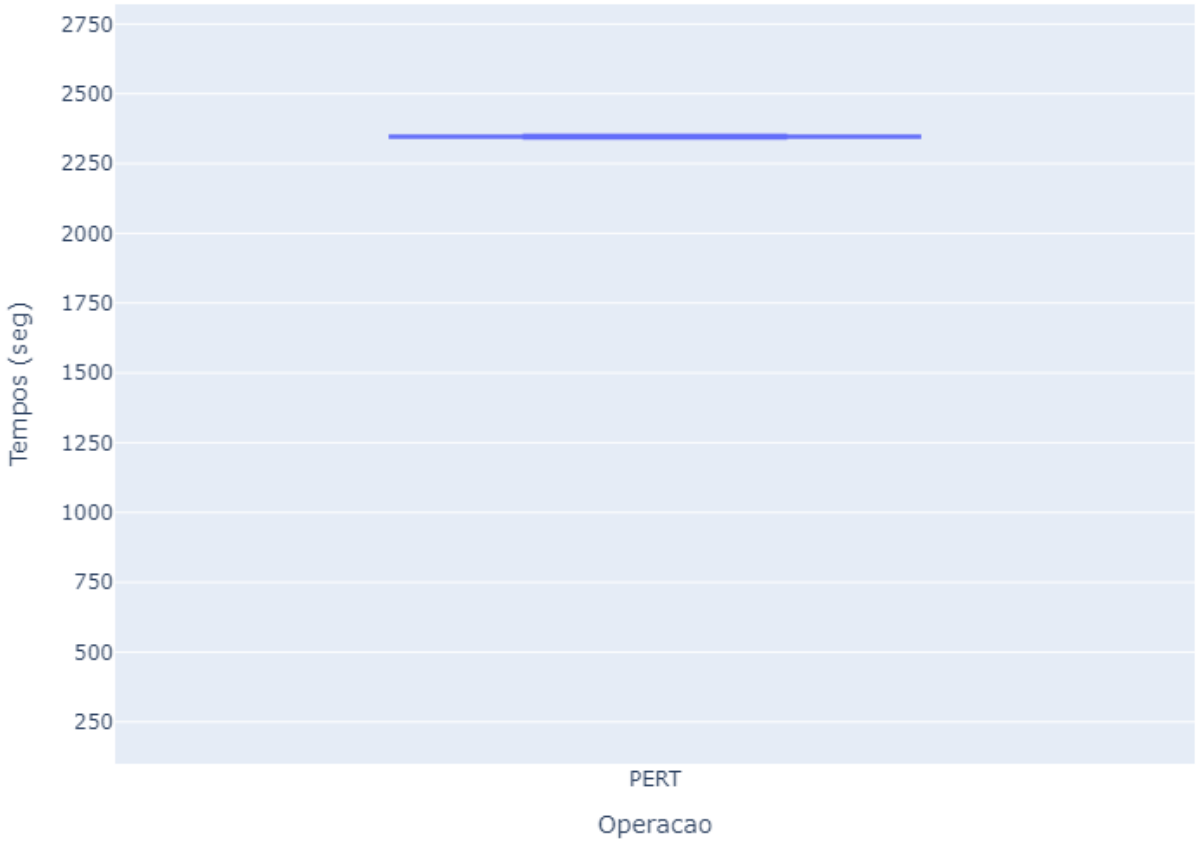

Figura 4.10 – Pertinência PostgreSQL

5 Considerações Finais

5.1 Conclusão

Um caso de implementação espacial tende a ser inviável de ser construído em um SGBD sem utilizar representações e operações geoespaciais. Sendo assim, este trabalho se propôs a avaliar as técnicas envolvidas na construção de um banco de dados espaciais e os recursos disponíveis para sua otimização, para poder ser uma ferramenta estatística na literatura para evidenciação dos ganhos obtidos pela devida implementação de um modelo de dados. Os ganhos foram significativos em quase todas as operações, além do custo de memória utilizado ter diminuído drasticamente.

O PostgreSQL não armazena as malhas das cidades e, ainda sim, na implementação teve um custo de espaço muito maior que o PostGIS, mais que 12x seu tamanho. Seu tempo de criação foi menor, possivelmente devido ao fato de seus dados serem menos complexos, o que leva a concluir que: uma solução simples tende a ser mais rápida mas tem um custo de armazenamento muito maior que uma solução menos abstrata.

Como pode-se observar, cada operação possui uma indexação, visão ou tabela temporária que pode melhorar seus resultados. Indexações tendem a otimizar quase todas as operações espaciais, as visões são úteis quando colocadas em funções que não exigem junções e subconsultas, como o Caso de estudo 3,4 e 5, e as tabelas temporárias são úteis quando a operação possui muitas junções ou sub-consultas, observado no Caso 1 e 2 do capítulo anterior.

5.2 Trabalhos Futuros

A partir deste trabalho, novos trabalhos podem ser desenvolvidos avaliando o custo de novas funções com os recursos aqui propostos. Seria interessante desenvolver uma indexação personalizada, de acordo com as malhas dos estados e cidades e avaliar seus ganhos.

Referências

CÂMARA, G.; MEDEIROS, J. S. d. Modelagem de dados em geoprocessamento. Sistemas de Informação Geográfica: aplicações na agricultura (ED Assad & EE Sano, eds). EMBRAPA, Brasília, p. 47–66, 2005.

CASANOVA, M. A.; CÂMARA, G.; DAVIS, C.; VINHAS, L.; QUEIROZ, G. d. Banco de dados geográficos. MundoGEO Curitiba, 2005.

CHOPRA, R. Database Management System (DBMS) A Practical Approach. [S.l.]: S. Chand Publishing, 2010.

CODD, E. F. Data models in database management. In: Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling. [S.l.: s.n.], 1980. p. 112–114.

CODD, E. F. A relational model of data for large shared data banks. Communications of the ACM, ACM New York, NY, USA, v. 26, n. 1, p. 64–69, 1983.

FINKEL, R.; BENTLEY, J. Quad trees: A data structure for retrieval on composite keys. Acta Inf., v. 4, p. 1–9, 03 1974.

GÜTING, R. H. An introduction to spatial database systems. the VLDB Journal, Springer, v. 3, p. 357–399, 1994.

GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. SIGMOD Rec., Association for Computing Machinery, New York, NY, USA, v. 14, n. 2, p. 47–57, jun. 1984. ISSN 0163-5808. Disponível em: <https://doi.org/10.1145/971697.602266>.

HARRINGTON, J. L. Relational database design and implementation. [S.l.]: Morgan Kaufmann, 2016.

MELLA, E.; RODRÍGUEZ, M. A.; BRAVO, L.; GATICA, D. Query rewriting for semantic query optimization in spatial databases. GeoInformatica, Springer, v. 23, p. 79–104, 2019.

NIEVERGELT, J.; HINTERBERGER, H.; SEVCIK, K. C. The grid file: An adaptable, symmetric multikey file structure. ACM Trans. Database Syst., Association for Computing Machinery, New York, NY, USA, v. 9, n. 1, p. 38–71, mar. 1984. ISSN 0362-5915. Disponível em: <https://doi.org/10.1145/348.318586>.

OOI, B. C.; SACKS-DAVIS, R.; HAN, J. Indexing in spatial databases. Unpublished/Technical Papers, Citeseer, 1993.

OPEN, G. Consortium Inc. Opengis simple features specification for sql. 1999.

ROCKOFF, L. The language of SQL. [S.l.]: Addison-Wesley Professional, 2021.

RUSSELL, C.; STEPHENS, J. Beginning MySQL database design and optimization: From novice to professional. [S.l.]: Apress, 2004.

SCHNEIDER, M. Spatial Data Types. [S.l.]: Citeseer, 2009.

SHEKHAR, S.; CHAWLA, S. Spatial databases - a tour. [S.l.: s.n.], 2003. ISBN 978-0-13-017480-2.

SHEKHAR, S.; CHAWLA, S.; RAVADA, S.; FETTERER, A.; LIU, X.; LU, C.-t. Spatial databases-accomplishments and research needs. IEEE transactions on knowledge and data engineering, IEEE, v. 11, n. 1, p. 45–55, 1999.

Apêndices

APÊNDICE A – Tabelas dos resultados

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	79.816722	2.389625
END	1.000000	NONE	48.418696	0.963168
END/BRIN	1.000000	NONE	48.084464	0.529707
END/GIST	1.000000	NONE	47.834226	0.452287
END/SPGIST	1.000000	NONE	57.960938	4.461303
GIST	1.000000	NONE	79.473941	0.971501
NONE	1.000000	NONE	84.365680	3.003944
SPGIST	1.000000	NONE	79.268274	1.191071
BRIN	NONE	1.000000	10.049679	0.061585
END	NONE	1.000000	10.052615	0.152662
END/BRIN	NONE	1.000000	9.949674	0.070767
END/GIST	NONE	1.000000	9.869149	0.095677
END/SPGIST	NONE	1.000000	9.851019	0.040504
GIST	NONE	1.000000	10.125438	0.053949
NONE	NONE	1.000000	11.158815	0.452381
SPGIST	NONE	1.000000	10.344344	0.105989
BRIN	NONE	NONE	82.004220	1.004942
END	NONE	NONE	48.677216	0.375827
END/BRIN	NONE	NONE	55.394353	2.980303
END/GIST	NONE	NONE	51.166108	0.555082
END/SPGIST	NONE	NONE	49.944483	0.300088
GIST	NONE	NONE	81.278240	1.236505
NONE	NONE	NONE	166.282792	1.216868
SPGIST	NONE	NONE	81.542673	1.229274

Tabela A.1 – DMAX_PM

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	74.288951	0.903507
END	1.000000	NONE	42.963209	0.615177
END/BRIN	1.000000	NONE	42.518647	0.611004
END/GIST	1.000000	NONE	44.464381	5.172927
END/SPGIST	1.000000	NONE	50.806441	2.984126
GIST	1.000000	NONE	74.377925	1.386647
NONE	1.000000	NONE	76.148117	1.020046
SPGIST	1.000000	NONE	75.154670	0.980794
BRIN	NONE	1.000000	4.707993	0.027413
END	NONE	1.000000	4.886966	0.049834
END/BRIN	NONE	1.000000	4.929144	0.045198
END/GIST	NONE	1.000000	4.829675	0.034268
END/SPGIST	NONE	1.000000	4.820913	0.041009
GIST	NONE	1.000000	4.811549	0.036990
NONE	NONE	1.000000	4.793671	0.034658
SPGIST	NONE	1.000000	5.272409	0.166388
BRIN	NONE	NONE	75.422169	1.075984
END	NONE	NONE	43.498929	1.830191
END/BRIN	NONE	NONE	47.965901	1.962262
END/GIST	NONE	NONE	45.409695	0.617415
END/SPGIST	NONE	NONE	44.284389	0.213904
GIST	NONE	NONE	75.878448	1.080023
NONE	NONE	NONE	120.384008	11.389707
SPGIST	NONE	NONE	73.888164	0.896858

Tabela A.2 – D_PM

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	75.378273	1.023113
END	1.000000	NONE	19.093311	0.128200
END/BRIN	1.000000	NONE	18.939523	0.136710
END/GIST	1.000000	NONE	19.038368	0.158007
END/SPGIST	1.000000	NONE	22.690596	0.432062
GIST	1.000000	NONE	75.166599	1.491060
NONE	1.000000	NONE	76.967075	0.930646
SPGIST	1.000000	NONE	75.851378	0.905636
BRIN	NONE	1.000000	27.264800	0.105580
END	NONE	1.000000	27.476551	0.079388
END/BRIN	NONE	1.000000	27.513189	0.196417
END/GIST	NONE	1.000000	27.162829	0.138045
END/SPGIST	NONE	1.000000	27.048592	0.170156
GIST	NONE	1.000000	27.369992	0.076601
NONE	NONE	1.000000	29.100907	1.797406
SPGIST	NONE	1.000000	28.360257	0.125572
BRIN	NONE	NONE	77.289539	1.123728
END	NONE	NONE	20.246066	0.216154
END/BRIN	NONE	NONE	23.170826	1.398031
END/GIST	NONE	NONE	21.165837	0.407884
END/SPGIST	NONE	NONE	20.884215	0.425124
GIST	NONE	NONE	78.259763	0.809210
NONE	NONE	NONE	119.847693	2.154817
SPGIST	NONE	NONE	76.076885	0.841827

Tabela A.3 – DMAX_API

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	76.506460	0.948081
END	1.000000	NONE	19.007038	0.166882
END/BRIN	1.000000	NONE	18.804402	0.164861
END/GIST	1.000000	NONE	18.891354	0.160170
END/SPGIST	1.000000	NONE	28.897232	5.884694
GIST	1.000000	NONE	77.016131	1.217094
NONE	1.000000	NONE	81.766941	1.764076
SPGIST	1.000000	NONE	76.611863	0.889105
BRIN	NONE	1.000000	27.520130	0.429082
END	NONE	1.000000	27.757502	0.469307
END/BRIN	NONE	1.000000	27.724698	0.423061
END/GIST	NONE	1.000000	27.463012	0.495139
END/SPGIST	NONE	1.000000	27.320436	0.407664
GIST	NONE	1.000000	27.696730	0.443559
NONE	NONE	1.000000	30.596270	3.952378
SPGIST	NONE	1.000000	28.650828	0.505990
BRIN	NONE	NONE	79.136959	0.992516
END	NONE	NONE	20.130665	0.354171
END/BRIN	NONE	NONE	22.288423	1.916730
END/GIST	NONE	NONE	20.889911	0.158070
END/SPGIST	NONE	NONE	20.626308	0.137542
GIST	NONE	NONE	79.141857	2.926215
NONE	NONE	NONE	120.999713	2.625274
SPGIST	NONE	NONE	78.767342	0.778829

Tabela A.4 – DMED_API

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	23.951673	0.335749
END	1.000000	NONE	23.755408	0.391347
END/BRIN	1.000000	NONE	23.506138	0.424174
END/GIST	1.000000	NONE	23.558660	0.369800
END/SPGIST	1.000000	NONE	28.704471	2.015247
GIST	1.000000	NONE	24.056757	0.704235
NONE	1.000000	NONE	25.509017	0.239469
SPGIST	1.000000	NONE	23.540325	0.333924
BRIN	NONE	1.000000	61.426002	3.215997
END	NONE	1.000000	60.155927	1.560557
END/BRIN	NONE	1.000000	60.481112	2.069382
END/GIST	NONE	1.000000	56.891576	0.444237
END/SPGIST	NONE	1.000000	56.819576	0.145290
GIST	NONE	1.000000	62.194646	2.979255
NONE	NONE	1.000000	61.107804	1.362591
SPGIST	NONE	1.000000	56.826277	0.491968
BRIN	NONE	NONE	24.196749	0.225385
END	NONE	NONE	23.698002	0.101614
END/BRIN	NONE	NONE	25.286228	0.645434
END/GIST	NONE	NONE	25.151271	0.655719
END/SPGIST	NONE	NONE	24.281082	0.199346
GIST	NONE	NONE	24.412844	0.136375
NONE	NONE	NONE	27.088551	1.725247
SPGIST	NONE	NONE	23.893850	0.337975

Tabela A.5 – CONTAINS

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	21.398077	0.323293
END	1.000000	NONE	21.242519	0.345204
END/BRIN	1.000000	NONE	21.256412	0.628024
END/GIST	1.000000	NONE	21.051142	0.319595
END/SPGIST	1.000000	NONE	26.723219	1.291217
GIST	1.000000	NONE	21.331964	0.337022
NONE	1.000000	NONE	22.964305	0.386109
SPGIST	1.000000	NONE	21.112324	0.325993
BRIN	NONE	1.000000	61.555813	0.170601
END	NONE	1.000000	65.965010	2.185015
END/BRIN	NONE	1.000000	62.239489	0.484174
END/GIST	NONE	1.000000	61.936067	0.160543
END/SPGIST	NONE	1.000000	61.675424	0.181221
GIST	NONE	1.000000	61.988691	0.381009
NONE	NONE	1.000000	61.618215	0.182484
SPGIST	NONE	1.000000	61.656952	0.171279
BRIN	NONE	NONE	24.326564	0.154614
END	NONE	NONE	23.665457	0.235888
END/BRIN	NONE	NONE	25.114415	0.372770
END/GIST	NONE	NONE	24.780165	0.231711
END/SPGIST	NONE	NONE	24.252145	0.215532
GIST	NONE	NONE	24.477777	0.412098
NONE	NONE	NONE	26.074126	0.695882
SPGIST	NONE	NONE	23.917492	0.198325

Tabela A.6 – COVEREDBY

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	89.702920	1.316673
END	1.000000	NONE	89.848457	1.129624
END/BRIN	1.000000	NONE	88.945681	1.201925
END/GIST	1.000000	NONE	88.695120	1.086701
END/SPGIST	1.000000	NONE	104.070354	6.204928
GIST	1.000000	NONE	90.538480	1.461689
NONE	1.000000	NONE	96.746347	0.316482
SPGIST	1.000000	NONE	89.007908	1.487328
BRIN	NONE	1.000000	241.526097	0.305382
END	NONE	1.000000	246.207273	4.610545
END/BRIN	NONE	1.000000	245.391563	3.174471
END/GIST	NONE	1.000000	244.099384	1.569657
END/SPGIST	NONE	1.000000	243.024527	0.167105
GIST	NONE	1.000000	242.640021	0.427468
NONE	NONE	1.000000	241.788835	0.215746
SPGIST	NONE	1.000000	241.620544	0.367217
BRIN	NONE	NONE	98.593268	1.977279
END	NONE	NONE	96.326087	0.606147
END/BRIN	NONE	NONE	100.612064	0.634674
END/GIST	NONE	NONE	100.487103	1.044915
END/SPGIST	NONE	NONE	98.184251	0.847001
GIST	NONE	NONE	98.547140	0.930985
NONE	NONE	NONE	102.461461	1.170851
SPGIST	NONE	NONE	97.155452	1.328154

Tabela A.7 – DISJOINT

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	21.480298	0.399647
END	1.000000	NONE	21.282047	0.341400
END/BRIN	1.000000	NONE	21.553040	0.336783
END/GIST	1.000000	NONE	21.292458	0.311147
END/SPGIST	1.000000	NONE	24.654195	0.851919
GIST	1.000000	NONE	21.426933	0.434205
NONE	1.000000	NONE	22.871479	0.122104
SPGIST	1.000000	NONE	21.190295	0.340592
BRIN	NONE	1.000000	61.653614	0.136968
END	NONE	1.000000	62.357872	0.082537
END/BRIN	NONE	1.000000	62.307836	0.127730
END/GIST	NONE	1.000000	61.886310	0.277545
END/SPGIST	NONE	1.000000	62.161582	1.236454
GIST	NONE	1.000000	62.181969	0.221675
NONE	NONE	1.000000	61.763174	0.065592
SPGIST	NONE	1.000000	63.681989	0.313834
BRIN	NONE	NONE	24.276919	0.184537
END	NONE	NONE	23.845671	0.141300
END/BRIN	NONE	NONE	25.166165	0.372377
END/GIST	NONE	NONE	25.137070	0.729321
END/SPGIST	NONE	NONE	24.454246	0.469541
GIST	NONE	NONE	24.415359	0.156817
NONE	NONE	NONE	25.604339	0.402109
SPGIST	NONE	NONE	24.225377	0.126686

Tabela A.8 – WITHIN

index	views	tabelas_temporarias	mean	std
BRIN	1.000000	NONE	23.715328	0.346289
END	1.000000	NONE	23.984875	0.298012
END/BRIN	1.000000	NONE	23.726438	0.501478
END/GIST	1.000000	NONE	23.624202	0.276508
END/SPGIST	1.000000	NONE	27.437390	0.781870
GIST	1.000000	NONE	23.750879	0.354031
NONE	1.000000	NONE	25.695392	0.534587
SPGIST	1.000000	NONE	23.747000	0.529224
BRIN	NONE	1.000000	61.468621	0.319252
END	NONE	1.000000	62.199025	0.160388
END/BRIN	NONE	1.000000	62.173395	0.195734
END/GIST	NONE	1.000000	61.673344	0.088311
END/SPGIST	NONE	1.000000	61.716281	0.109334
GIST	NONE	1.000000	61.916957	0.350319
NONE	NONE	1.000000	61.593636	0.296181
SPGIST	NONE	1.000000	63.461776	0.133450
BRIN	NONE	NONE	24.246782	0.182777
END	NONE	NONE	23.760699	0.134926
END/BRIN	NONE	NONE	25.507394	0.422014
END/GIST	NONE	NONE	25.488313	0.294414
END/SPGIST	NONE	NONE	24.273550	0.203245
GIST	NONE	NONE	24.302288	0.199617
NONE	NONE	NONE	25.335904	0.160773
SPGIST	NONE	NONE	24.192233	0.315254

Tabela A.9 – INTERSECTS

operacao	mean	std
DMAX_API	556.172845	20.282813
DMAX_PM	213.504581	8.016883
DMED_API	270.382645	1.787354
D_PM	181.702729	1.323348
PERT	2345.909667	2.421577

Tabela A.10 – POSGRESQL

index	mean	std
BRIN	2.169425	0.069180
END	24.176439	0.704765
GIST	16.448952	0.164345
SPGIST	30.788618	0.321272

Tabela A.11 – Criação de index

view	mean	std
CONTAINS	0.000798	0.000921
COVEREDBY	0.000798	0.000922
DISJOINT	0.000964	0.001603
DMAPI_VIEW	0.001232	0.002248
DMAXAPI_VIEW	0.001130	0.001000
DMPM_VIEW	0.002625	0.007984
DPM_VIEW	0.001231	0.001896
INTERSECTS	0.000863	0.000894
WITHIN	0.000499	0.000971

Tabela A.12 – Criação de visões

index	temp_table	mean	std
BRIN	centroide	37.636128	3.799028
END	centroide	25.088751	1.081542
END/BRIN	centroide	23.984936	0.076977
END/GIST	centroide	23.918189	0.053907
END/SPGIST	centroide	23.775284	0.044281
GIST	centroide	32.869122	0.273795
None	centroide	32.450320	0.718112
SPGIST	centroide	32.341169	0.170419
BRIN	temp_cidade	0.257249	0.027886
END	temp_cidade	0.253723	0.031367
END/BRIN	temp_cidade	0.216339	0.013034
END/GIST	temp_cidade	0.216284	0.012881
END/SPGIST	temp_cidade	0.214353	0.011993
GIST	temp_cidade	0.227498	0.015142
None	temp_cidade	0.288767	0.074156
SPGIST	temp_cidade	0.222826	0.016014
BRIN	dist_api_temp	69.834235	1.545675
END	dist_api_temp	22.716870	1.806207
END/BRIN	dist_api_temp	19.950978	0.485270
END/GIST	dist_api_temp	19.887766	0.420087
END/SPGIST	dist_api_temp	19.306561	0.167870
GIST	dist_api_temp	72.161883	1.689700
None	dist_api_temp	69.937762	1.604318
SPGIST	dist_api_temp	69.053612	1.460129
BRIN	DM_API_TT	77.389473	11.127772
END	DM_API_TT	20.667618	1.143410
END/BRIN	DM_API_TT	19.718727	0.165788
END/GIST	DM_API_TT	19.631710	0.262546
END/SPGIST	DM_API_TT	19.185025	0.286339
GIST	DM_API_TT	68.828272	1.229682
None	DM_API_TT	79.773993	11.417979
SPGIST	DM_API_TT	66.059547	1.370455

Tabela A.13 – Criação de tabelas temporárias

index	temp_table	mean	std
BRIN	dist_medio	40.170946	0.125785
END	dist_medio	17.544236	2.204720
END/BRIN	dist_medio	15.624671	0.424300
END/GIST	dist_medio	15.575622	0.046014
END/SPGIST	dist_medio	15.469088	0.059627
GIST	dist_medio	42.753963	3.532333
None	dist_medio	40.230009	0.113034
SPGIST	dist_medio	40.296331	0.130641
BRIN	max_distancias	11.215136	0.096291
END	max_distancias	11.127246	0.350103
END/BRIN	max_distancias	10.673554	0.099692
END/GIST	max_distancias	10.608336	0.100651
END/SPGIST	max_distancias	10.570399	0.170034
GIST	max_distancias	11.228148	0.092017
None	max_distancias	11.220107	0.136571
SPGIST	max_distancias	11.255274	0.106460
BRIN	temp_pontos	4.013073	0.212491
END	temp_pontos	3.417050	0.292288
END/BRIN	temp_pontos	3.232325	0.067935
END/GIST	temp_pontos	3.204088	0.031723
END/SPGIST	temp_pontos	3.174487	0.028386
GIST	temp_pontos	3.540532	0.079532
None	temp_pontos	3.470156	0.045844
SPGIST	temp_pontos	3.540011	0.209266

Tabela A.14 – Criação de tabelas temporárias