Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Banco de Dados Geográfico utilizando NoSQL Cassandra: Estudo de Caso em um Sistema de Informação Geográfico com Participação Popular

Matheus da Silva Nascimento

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia de Computação

Orientadora
Prof.ª Dr.ª Maristela Terto de Holanda

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia de Computação

Coordenador: Prof. Dr. Ricardo Pezzuol Jacobi

Banca examinadora composta por:

Prof.ª Dr.ª Maristela Terto de Holanda (Orientadora) — CIC/UnB
Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo — CIC/UnB
Prof. Dr. Marcio de Carvalho Victorino — CIC/UnB

CIP — Catalogação Internacional na Publicação

Nascimento, Matheus da Silva.

Banco de Dados Geográfico utilizando NoSQL Cassandra: Estudo de Caso em um Sistema de Informação Geográfico com Participação Popular / Matheus da Silva Nascimento. Brasília : UnB, 2016.
147 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. SIG, 2. Banco de Dados Geográfico, 3. NoSQL, 4. Cassandra.

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Banco de Dados Geográfico utilizando NoSQL Cassandra: Estudo de Caso em um Sistema de Informação Geográfico com Participação Popular

Matheus da Silva Nascimento

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia de Computação

Prof.ª Dr.ª Maristela Terto de Holanda (Orientadora)
CIC/UnB

Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo
CIC/UnB

Prof. Dr. Marcio de Carvalho Victorino
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia de Computação

Brasília, 4 de agosto de 2016

Dedicatória

Dedico este trabalho a minha família.

Agradecimentos

Agradeço a Deus pela força concedida para a realização deste trabalho. Agradeço a toda minha família, amigos pelo apoio e incentivo a continuar nesta trajetória. Também agradeço à professora Dr.ª Maristela Terto de Holanda por toda orientação, apoio, aprendizado e tolerância.

Resumo

O Sistema de Informação Geográfica é uma ferramenta muito importante em diversas áreas de pesquisa. Entretanto, junto com a atual plataforma que a *Internet* oferece, o volume de dados gerados por usuários comuns está cada vez maior e, com isso, vem a necessidade de novas soluções para o armazenamento de dados geográficos, estruturados e não estruturados de uma maneira mais eficiente, de tal forma que o usuário possa ter condições de participar voluntariamente das solicitações de uma determinada aplicação. Este trabalho propõe a utilização do banco de dados NoSQL Cassandra para armazenamento de dados geográficos e utilização em Sistema de Informação Geográfico com Participação Popular.

Palavras-chave: SIG, Banco de Dados Geográfico, NoSQL, Cassandra.

Abstract

The Geographic Information System is a tool very important in several research areas. However, together with the curent platform that the Internet offers, the amount of data that commun users generate are becoming bigger, then comes the need of new solutions to store geographical, unstructured and structured data in a more efficient way that allow users to voluntarily participate of a request for a given application. This work purpose the Cassandra database to store geographical data and usage in Public Participation Geographic Information System.

Keywords: GIS, geodatabase, NoSQL, Cassandra

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um banco de dados geográfico (ou espacial) possui a característica de armazenar e manipular informações espaciais a partir de operações matemáticas e geométricas e é comumente utilizado em aplicações com a abordagem de Sistema de Informação Geográfica (SIG) em que esses dados são representações espaciais da terra e podem ser analisados em diversos aspectos [32].

Dados geográficos já são abordados há muito tempo e o termo Sistema de Informação Geográfica como dados computacionais é utilizado desde a década de 1960 [28, 41]. Atualmente, os dispositivos móveis, como *smartphones* e *tablets*, chegam a ter uma capacidade de processamento que os tornam capazes de acessar a *Internet*, localização via *Global Positioning System* (GPS) e isso favorece o geoprocessamento como uma aplicação de fácil acesso ao usuário comum.

Vários fatores contribuíram para o desenvolvimento do SIG em dispositivos móveis, dentre eles estão a WEB 2.0 que traz a concepção da *Internet* como uma plataforma, o qual o usuário interage e armazena suas informações, e a expansão de uma infraestrutura de comunicação com serviço GPS e o acesso à *Internet* por meio de serviços de comunicação de redes 3G, 4G e *wi-fi*, com velocidades cada vez maiores.Outro fator é a crescente facilidade de aquisição desses dispositivos. Esses fatores combinados tornam o usuário um forte participante na geração de informações georreferenciadas [21, 29, 36].

Um desafio na área de aquisição e disponibilização de dados para o usuário final é que esses dados podem ser rapidamente requeridos e armazenados em quantidades cada vez maiores, tornando-os difíceis de serem processados. Além disso, eles podem conter complexos tipos de estruturas como por exemplo áudio, vídeo e foto. Esse grande volume de dados, que pode ter formato heterogêneo, sendo produzindo rapidamente pelo ambiente é conhecido como "*Big Data*" [17, 19]. Para o gerenciamento do *Big Data*, novos modelos de dados, chamados NoSQL (*Not only SQL*), foram definidos. Esses modelos de dados mostram-se bastante eficientes para armazenar dados não estruturados quando comparados com alguns bancos de dados relacionais, assim como propõem estratégias para poder sanar o problema de grande volume de dados e picos de requisições de leitura/escrita. Neste sentido, os dados geográficos também possuem tendências a serem cada vez maiores devido a quantidade de informação e nível de detalhamento que são introduzidos [28, 30].

A proposta desta monografia é desenvolver uma arquitetura para armazenamento de dados geográficos com o NoSQL Cassandra. O banco de dados Cassandra é um banco de

dados NoSQL orientado à coluna, sendo analisado nesta monografia como proposta para armazenamento de dados geográficos obtidos por participação popular da comunidade, a qual utiliza o serviço de localização geográfica a partir de dispositivos móveis.

1.1 Objetivo

O principal objetivo deste trabalho é apresentar uma solução para armazenamento de dados geoespaciais utilizando o banco de dados NoSQL Cassandra, tendo como cenário uma análise do seu desempenho em relação ao volume de requisições e dados armazenados por parte do usuário em uma aplicação de Sistema de Informação Geográfica com Participação Popular.

1.1.1 Objetivos Específicos

Os objetivos específicos a serem alcançados neste trabalho estão descritos a seguir:

- Propor uma solução para armazenamento de dados geográficos com o banco de dados NoSQL Cassandra;
- Validar a arquitetura proposta com um sistema de informação com participação popular;
- Mensurar o desempenho da arquitetura proposta.

1.2 Estrutura do Trabalho

Os próximos capítulos desta monografia estão dispostos da seguinte forma:

- Capítulo 2: Fundamentação teórica acerca do Sistema de Informação Geográfica, Sistema de Informação Geográfica com Participação Popular (SIGPP), bancos de dados geográficos, bancos de dados NoSQL, tipos de banco de dados NoSQL e arquitetura e características do banco de dado Cassandra.
- Capítulo 3: Apresentação do desenvolvimento da solução de armazenamento de dados espaciais no banco de dados Cassandra com teste de leitura e escrita.
- Capítulo 4: Estudo de caso em uma aplicação móvel com abordagem de SIGPP.
- Capítulo 5: Conclusões e trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo são descritos os conceitos teóricos relacionados aos bancos de dados geográficos. Na Seção 2.1 é presentado o bancos de dados geográficos, na Seção 2.2 o Sistema de Informação Geográfico, e na Seção 2.3 os bancos de dados NoSQL. Em especial, é explicado a arquitetura do banco de dados NoSQL Cassandra.

2.1 Banco de Dados Geográfico

2.1.1 Fenômeno Geográfico

Os fenômenos geográficos são ou "objetos" ou "campos" que podem ser representados em duas ou três dimensões no espaço euclidiano e possuem as características de serem definidos ou nomeados, georreferenciados e pode-se designar um valor temporal o qual o fenômeno está presente [2, 13]. Objetos e campos são definidos da seguinte forma [2]:

- Campo: o campo é um fenômeno geográfico que para cada ponto na área de estudo pode ser determinado um valor, como por exemplo altura e temperatura.
- Objeto: o objeto é um fenômeno geográfico que povoa a área de estudo e é bem-definido, discreto e possui limites. O espaço entre objetos são potencialmente desconhecidos ou vazios.

Os objetos geográficos podem conter uma combinação de lugar, forma, tamanho, e orientação. O formato representa a dimensão do objeto. Um ponto por exemplo pode representar uma árvore em um bosque, uma linha representa uma parede, uma área pode ser um campo de futebol e volume podem ser montanhas. Formato, tamanho e local trazem a noção de borda ou limite. Se uma borda é reconhecida, então um objeto tem seu formato, tamanho e local determinado. Os fenômenos de campo normalmente representam fenômenos naturais no mundo real, e os objetos produtos feitos pelo homem. Desta forma, são empregados objetos como regiões de interesse no mapa virtual a fim de modelar o mundo real [2].

2.1.2 Representação Computacional

A representação espacial é feita por modelos que são limitados pela finitude do banco de dados, sendo assim necessário fazer simplificações, abstrações, generalizações e até

comprimir os dados. O conjunto de objetos que representam uma mudança em apenas uma variável são denominados "camadas", e a associação desses modelos é chamado de "modelos de camada". Em um modelo de camada, por definição, cada localização possui um único valor da variável em questão. Um desafio que aparece com a modelagem desses objetos é quando um objeto bem definido não está em um local bem definido, e uma solução comum é a "múltipla representações" que é a variação da escala para a extensão espacial [20].

Modelos de Camada

Um modelo de camadas é uma coleção de objetos que tem por objetivo a representação espacial com a mudança de apenas uma variável. Cada modelo é capaz de retornar um valor em qualquer localização (x, y) e um banco de dados geográfico pode conter vários modelos. Alguns modelos de camadas estão retratados na Figura 2.1 e, normalmente, são dos seguintes tipos [20]:

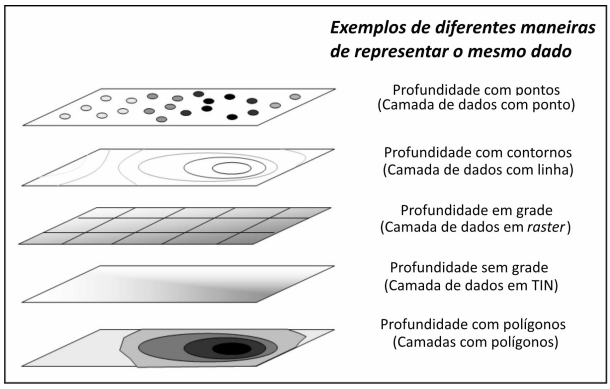


Figura 2.1: Diferentes Modelos de Camadas Representando a Mesmo Informação [1].

1. Amostra irregular de pontos: O banco contém um conjunto de registros $< x, y, z >$ representando amostras do valor da variável em um conjunto finito. As amostras estão irregularmente espaçadas. Exemplo: dados de estação meteorológica.

2. Contornos: É um conjunto de dados composto por linhas, cada uma consistindo de um conjunto ordenado $< x, y >$ de pares de pontos. Cada linha possui um valor z associado. Os pontos de cada conjunto são conectados por uma linha reta. Exemplo: contorno de dados digitalizados.

3. Matriz de célula: A área é quebrada em células regulares em uma rede quadrilateral. O valor da variável é o mesmo para todo o espaço ocupado por uma célula. Exemplo: imagem por sensor remoto.

4. Rede Triangular: Também chamada de TIN (*Triangulated Irregular Network*), se aproveita da partição da área em triângulos irregulares. O valor da variável é determinado em cada vértice e varia linearmente sobre o triângulo. Exemplo: rede triangular irregular de elevação.

[1] Adaptado de (http://www.gisinecology.com/faqs__frequently_asked_question.htm)

5. Polígonos: A área é quebrada em conjunto de polígonos, de modo que toda localização se encaixa em exatamente um polígono. As bordas do polígono são descritas por um conjunto ordenado $< x, y >$ de pares de pontos. Exemplo: mapa do solo.

O modelo (3) é comumente chamados de *"raster"*, e os modelos (1), (2), (4) e (5) são chamados de "vetor". Intuitivamente, os dois grupos representam o modo como o dado é representado, seja por criação de imagem (*raster*) ou por formas geométricas (vetor). Esses cinco modelos de camadas podem ser visualizados com a exibição de um conjunto de pontos, linhas ou/e áreas [20].

Nesta monografia, toda a análise deu-se com dados vetoriais, desta forma esta representação é detalhada na próxima subseção.

Representação por Vetor

Esta representação costuma ser utilizada com objetos que são bem definidos. A representação por *raster* não define o local exato do fenômeno georreferenciado e sim uma área inteira, que é definida pela resolução da célula. Na representação por vetor, existe o esforço de detalhar a georreferencia dos fenômenos. Georreferência é um par coordenado que representa um espaço geográfico, e esse par coordenado é definido como vetor. A Figura 2.2 mostra como os objetos são representados digitalmente. Essas representações seguem os seguintes critérios [2]:

Figura 2.2: Representação por Vetor [2].

- Representação de ponto: Ponto é um par coordenado simples $< x, y >$ no plano 2D e uma tripla coordenada $< x, y, z >$ no plano 3D. Pontos representam objetos

[2]Adaptado de (http://gisedu.colostate.edu/WebContent/nr505/2012_Projects/Team5/GISConcepts.html)

os quais a forma e o tamanho não são relevantes em um contexto específico. Cada ponto pode carregar dados extras refentes ao contexto. Exemplo: Carros em um estacionamento;

- Representação de linha: A linha é composta por dois nós distintos que se ligam e podem ter zero ou mais nós internos (vértices). Nós são semelhante aos pontos, entretanto não carregam valor semântico e não precisam ser visualizados em conjunto com a linha, apenas ajudam a obter uma melhor aproximação para a característica do objeto real. Para representar linhas curvas, é utilizado uma lista finita de vértices. Exemplo: rios e rodovias;

- Representação de área: A melhor representação de área com abordagem vetorial é por meio de polígonos para delimitar os limites de uma área.

  Em outras palavras, polígono é uma sequência circular de linhas. O espaço é determinado por partições. Cada partição é um polígono e com isso os polígonos não são sobrepostos, apenas delimitados por outros polígonos em uma mesma camada. Polígonos, assim como curvas, continuam sendo aproximações imperfeitas da realidade. Exemplo: prédios, estados, países e lagoas.

2.1.3 Modelo Geométrico

Na Seção 2.1.1, a informação geográfica foi delimitada em objetos e campos, e a Seção 2.1.2 definiu maneiras de representar essas entidades geometricamente. Neste sentido, os bancos de dados geográficos possuem a função de armazenar e manusear dados espaciais. Dados espaciais são importantes para aplicações em SIG, pois possuem métodos de indexação espacial e possibilitam a análise georreferenciada [18].

Guimarães et al. [13] ressaltam que na década de 80, as estações de trabalho gráficas em conjunto com evolução dos computadores pessoais e surgimento de banco de dados relacionais corroborou para uma grande difusão do uso de SIG com a introdução de muitas funções de análise espacial. Entretanto, algumas complicações surgiram devido a essa difusão:

- Diferentes métodos e padrões utilizados para a coleta e a manutenção dos dados inviabilizam o uso em conjunto;

- Políticas de disponibilização dos dados; e

- Diferença de unidades de medidas e conceitos para os dados.

Assim, foi criado em 1994 o *Open GIS Consortium* (OGC[3]) com objetivo de solucionar estes problemas. OGC é uma organização internacional que, entre outras coisas, propõe padrões de modelagem da terra e os fenômenos a ela relacionados, matematicamente e conceitualmente [13].

A documentação da OCG descreve a modelagem e a arquitetura comum para geometrias com características simples, definida como *Simple Feature Specification* (SFS) em notação UML conforme a Figura 2.3. Na figura é possível ver que todas as outras entidades são herdadas de *Geometry* e que esta entidade possui um sistema de referencia espacial

[3]http://www.opengeospatial.org/

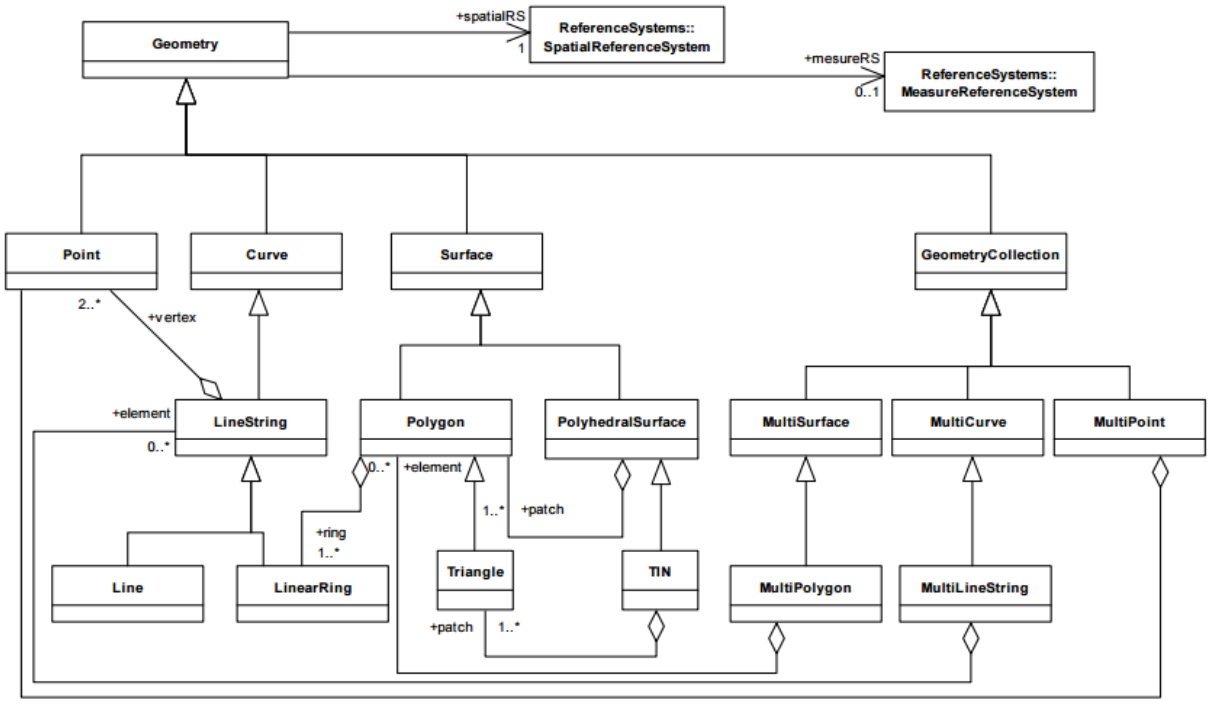


Figura 2.3: Hierarquia de Classes de Geometria[4].

e de medida. Ponto, curva, superfície e coleção de geometrias são herdadas diretamente da classe raiz e podem dar origem a diversos tipos de geometrias que se relacionam entre si. A documentação descreve todas as propriedades da classe raiz e o padrão para os atributos de cada tipo de geometria [39].

Este padrão também apresenta a definição de *Well-known Text Representation of Geometry* (WKTGeometry) e *Well-known Binary Representation for Geometry* (WKBGeometry). Estas duas estruturas são utilizadas para representar a informação geométrica. A primeira estrutura representa a informação com um modelo textual e o segundo com uma concatenação de *bytes* [39].

Uma segunda documentação da OCG especifica a modelagem para banco de dados relacionais. A Figura 2.4 mostra as três tabelas deste modelo. A tabela "SPATIAL_REF_SYS" possui um atributo (SRID) para identificar um sistema de referência espacial e outros três atributos que caracterizam essa referência. Um sistema de referência espacial dá significado para as coordenadas de uma geometria como por exemplo "Latitude Longitude" e "UTM Zone 10". A tabela "Features Table" é a tabela que armazena os objetos que representam a abstração do mundo real, com atributos e com a coluna que representa a geometria: "Goemetry_column(GID)". A última tabela, a "GEOMETRY_COLUMNS", mantém um registro de todas as colunas geométricas existentes. Ela possui um atributo para referenciar um sistema de referência espacial, um para a dimensão da coordenada da coluna geométrica, três para referenciar a *Feature Table* que possui essa coluna e um para referenciar o atributo que representa a geometria [38].

[4]fonte: (http://portal.opengeospatial.org/files/?artifact_id=25355)
[5]fonte: (http://portal.opengeospatial.org/files/?artifact_id=25354)

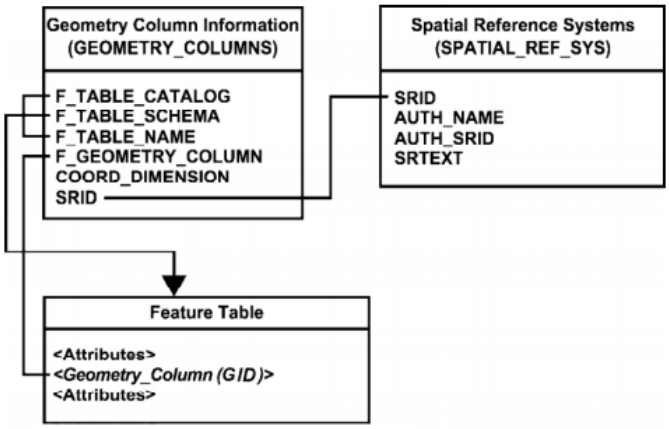


Figura 2.4: Modelagem SQL Geoespacial[5].

2.2 Sistema de Informação Geográfico (SIG)

Xiaomin et al. [19] definem o SIG como um sistema de *hardware* e *software* combinado com geografia, geomática, cartografia, ciência da computação, operações de pesquisa e outras disciplinas. Os objetivos do SIG são capturar, guardar, gerenciar, mostrar e analisar dados geoespaciais.

As aplicações de SIG são vastas: planejamento urbano, gerenciamento de tráfego, suporte em desastres naturais, SIG com participação popular, instalações elétricas, áreas de proliferação de doenças, planejamento regional e análise de decisão. Esta abordagem também pode ser aplicada na área da saúde para localização de hospitais em situações de emergência, rotas para companhias de entrega, localização de veículo particular ou público na área de transporte, avaliação de estabelecimentos comerciais ou instituições públicas, outras aplicações como terras de plantio e de corte, dentre outros [16, 30].

O objetivo primário do SIG é criar uma representação do mundo real por meio de modelos que podem ser visualizados digitalmente, recriando assim aspectos e fenômenos chaves do mundo real, conforme mostra a Figura 2.5 onde tem-se fenômenos do mundo real sendo transformados em dados digitais e manipulados computacionalmente. As representações nunca serão perfeitas, tendo em vista vários fatores limitantes como por exemplo a quantidade de dados que se pode armazenar, a quantidade de detalhes que se pode capturar e fatos ou relações que não são abrangidos pelo modelo escolhido. Modelar é o processo de criar uma abstração de algo (no caso a superfície terrestre) para ser mais facilmente manipulado; discretização que converte a complexa geografia real em números [2, 20].

2.2.1 Sistema de Informação Geográfico Voluntário

Goodchild [22]define Sistema de informação Geográfica Voluntário (SIGV) como sendo a soma da plataforma que a Web 2.0 oferece com inteligência coletiva e neogeografia [3].

Neogeografia é o conjunto de técnicas e ferramentas que permitem usuários criarem e usarem seus próprios mapas. Neogeografia está relacionado com compartilhamento de

[6]Adaptado de [2]

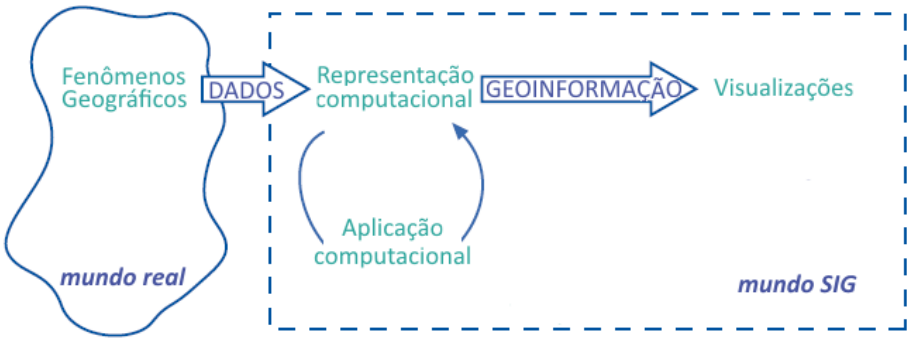


Figura 2.5: Representação do Mundo Real em um SIG [6].

informações de localização com amigos e visitantes, ajudando a modelar o contexto, e transferindo conhecimento por meio da cultura do lugar [3].

VGI está associado com os dados geográficos gerados e distribuídos por pessoas que não sejam necessariamente profissionais, mas que possuem conhecimentos capazes de aprimorar as informações espaciais existentes[22]. Bons exemplos de trabalhos com VGI são o OpenStreetMaps e Wikimapia [21].

2.2.2 Sistema de Informação Geográfico com Participação Popular

A participação popular em sistemas de informação geográfico envolve a pesquisa em aplicações regionais e ambientais em que o público em geral é o principal grupo participante em contraste com grupos de interesse. O objetivo é melhorar a qualidade de tomada de decisões e o crescimento do nível de participação da sociedade. O termo *Public Participation Geographic Information System* (PPGIS) foi concebido em 1996 para descrever como o sistema de informação geográfico poderia suportar a participação pública em uma variedade de aplicações com objetivo de conceder mais importância a parte da sociedade mais desfavorecida. PPGIS também pode ser utilizado pelo governo de países democráticos como forma mais efetiva de potencializar a participação pública e a consulta à comunidade no planejamento urbano e tomada de decisões [4].

O Sistema de Informação Geográfico com Participação Popular oferece a habilidade de investigação de um problema a partir de dados regionais para retornar propostas positivas de solução. Os profissionais dessa área tem visto que a participação popular é mais efetiva com orientação visual. Este motivo torna SIG uma ferramenta importante por promover a leitura de mapas e facilitar o entendimento mútuo e aceitação pública sobre fatos básicos [37].

Um desafio desta abordagem é definir qual o "público" alvo a se coletar as informações. Claramente, diferentes públicos afetam os resultados de uma pesquisa. Geralmente o público tende a ser indivíduos familiarizados com a área de pesquisa, o que torna os resultados mais acurados, entretanto indivíduos menos familiarizados podem se expressar de forma a demonstrar controvérsia no resultado do público familiarizado [4].

2.3 Banco de Dados NoSQL

Os bancos de dados NoSQL (*Not only SQL*) começaram a aparecer no início do Século XXI, quando a Web 2.0 trouxe a visão da *Internet* como uma plataforma para milhões de usuários interagirem e participarem, gerando um volume imenso de dados [28, 29, 36].

Em 2000 começou o projeto do neo4j, em 2004 começou o projeto BigTable do Google, em 2005 foi aberto o projeto CouchDB, em 2006 foi publicado o artigo "Big Table" pelo Google, em 2007 o artigo "Dynamo" pela Amazon, 10gen começou a codificar o MongoDB, foi lançado o neo4j e Powerset abriu o projeto HBase e em 2008 o Facebook abriu o projeto do Cassandra. Vários outros projetos ocorreram nesse período. Em 2009 o Termo NoSQL foi reintroduzido como banco de dados não-relacional com abordagens para soluções de armazenamentos de dados [12, 23, 27].

Antes de tratar especificamente do banco de dados NoSQL, na próximo Subseção são apresentados características do banco de dados relacional. Isto é importante para uma comparação entre os dois modelos.

2.3.1 Banco de Dados Relacional

Banco de dados relacional vem sendo utilizados desde 1970 quando foi desenvolvido por Edgar Codd [5]. Desde então, muitos *softwares* foram desenvolvidos tendo como base o modelo de banco de dados relacional como solução pronta e definitiva para qualquer tipo de sistema, entretanto utilizar esse modelo para todos os casos e problemas pode ser ineficiente, já que esse modelo é baseado nas propriedades ACID (atomicidade, consistência, isolamento e durabilidade) de transações em que as propriedades são bem restritivas. As propriedades ACID estão presente nas transações de banco de dados relacional, são elas [7, 31]:

- Atomicidade: a transação é uma unidade atômica, ou ela é executada completamente ou nada é executado;
- Consistência: uma transação deve garantir que o banco de dados permaneça consistente após ser efetuada;
- Isolamento: cada transação é efetivada independente de transações paralelas; e
- Durabilidade: se uma transação for concluída, então o banco deve garantir que a mudança persistirá mesmo em cenário de falha.

Para muitos sistemas, o banco de dados relacional é inteiramente suficiente para atender as necessidades de persistência seguindo essas propriedades, no entanto Vaish [24] lista algumas características que o modelo relacional não provê de forma eficiente e que podem ser eventualmente requeridos se ocorrer uma expansão do sistema em termos da capacidade e forma como os dados são armazenados:

- Esquema Flexível: Otimizações futuras podem necessitar integrar o banco com aplicações externas, o que requer uma flexibilização do modelo de dados. Em um SGBDR (Sistema Gerenciador de Banco de Dados Relacional) que já contenha algum dado, adicionar colunas que não tenham valor padrão necessitará de atualização

para cada novo valor desta coluna. Esta não é uma boa opção principalmente se existe muitas linhas e o número de colunas a serem adicionadas é grande. Novas tabelas aumentam a complexidade por introduzir mais relacionamentos.

- Atualização de dados: Em um banco de dados distribuídos, a propagação de atualização para múltiplos nós sobre o sistema. Se o sistema não suportar escrita para múltiplos nós simultaneamente, então existe o risco de um nó falhar, o que pode comprometer todo o processo de escrita. Se alguma transação fica aberta por muito tempo, o atualização dos dados pode perder em termos de desempenho.
- Escalabilidade: Na maioria das vezes, a única escalabilidade que é necessária é a operação de leitura, entretanto alguns fatores impactam o crescimento da velocidade dessas operações. Algumas perguntas chaves para se fazer em relação à escalabilidade: Qual é o tempo gasto para sincronizar os dados sobre as instâncias do banco de dados físico? Qual é o tempo gasto para sincronizar os dados sobre o *datacenter*? Qual é a largura de banda requerida para sincronizar os dados? A troca de dados está otimizada? Qual a latência quando alguma atualização é sincronizada no servidor? Tipicamente as gravações serão bloqueadas durante uma atualização.

Outras características do modelo relacional são [7]:

- Consultas complexas: Tradicionalmente as tabelas planejadas são normalizadas, o que significa que os desenvolvedores precisam escrever consultas complexas com operações de *"JOIN"*, que não são apenas difíceis de implementar e manter como também utilizam recurso substancial do banco de dados para executar.
- Limitação do Hardware: Apesar de SGBDRs terem capacidade para aumentar a escala verticalmente, isto é, aumentar o número de transações por período de tempo em um mesmo nó, isto pode demandar uma capacidade de processamento e memória proporcionalmente maiores.

Xiaomin et al. [19] ainda menciona a inabilidade dos bancos relacionais de tratarem dados semi-estruturados e não-estruturados, ou seja, arquivos de imagens, vídeos, áudio e etc.

2.3.2 Características do Banco de Dados NoSQL

NoSQL é definido como o conjunto de banco de dados que não seguem o princípio de modelo relacional e que são arquitetados principalmente para serem rápidos e escaláveis.

Os bancos de dados não-relacionais são guiados pelas propriedades denominada BASE e surgiram com a proposta de implementar bancos que atendam melhor a determinados requisitos que surgiram ao longo dos anos como por exemplo o grande volume de dados gerados e a heterogeneidade dos dados.

Esses novos bancos de dados são uma proposta para resolver os problemas de *Big Data*. *Big Data* é caracterizado pelos "3 Vs": volume, velocidade, variedade [28]. O volume representa a escala de dados que é gerado e armazenado em quantidades cada vez maiores. A IBM estima que são criados 2.5 quintilhões de bytes (2.3 trilhões de gigabytes) todos os dias. A velocidade diz respeito à análise dos dados que precisam seguir o ritmo da

velocidade em que os dados são adquiridos. A variedade simboliza as várias formas e fontes tais como sensores, e-mails, *stream*, áudio, imagem. Alguns pesquisadores ainda definem outras características, chegando até "7 Vs": volume, velocidade, variedade, veracidade, validade, volatilidade e valor. A veracidade dos dados refere-se a qualidade de confiança e integridade das informações. Validade é a exatidão e a precisão que os dados possuem. A volatilidade é definida como o tempo de vida dos dados. O valor descreve a parte de aplicação destes dados aos negócios e à economia [15].

Em geral, os bancos NoSQL enfrentam o problema do grande volume de dados aumentado escalabilidade horizontal, isto é, aumentando a quantidade de máquinas interconectadas. O conjunto de máquinas conectadas entre si é denominado sistema distribuído ou *cluster*.

O banco relacional (amplamente utilizado), pode ser o suficiente para o desenvolvimento de uma nova aplicação, visto que dependendo do número de prováveis usuários que submeterão operações de escrita e leitura. Os SGBDRs podem até apresentar desempenho maior que banco de dados NoSQL caso o número de operações não seja tão grande. Por outro lado, se o objetivo é suportar uma quantidade de operações cada vez maior, provavelmente com cada vez mais usuários cadastrados nesta aplicação, então deve-se analisar a prioridade da performance e da disponibilidade de receber novas requisições de operações, sendo assim o NoSQL uma melhor opção. Para avaliar os bancos NoSQL, Chandra [7] aponta algumas características dos bancos NoSQL:

- Esquema Livre: Bancos relacionais seguem estritamente o esquema definido pelas tabelas em cada tupla, ou seja, cada atributo é atômico e já tem prescrito seu domínio. Bancos NoSQL são menos restritos ao esquema e são uma melhor solução para sistemas que tendem a mudar frequentemente o esquema de dados.

- Arquitetura não compartilhada : Ao invés de utilizar um ambiente compartilhado como base de dados, como a rede de área de armazenamento (*storage area network*), a arquitetura é feita para cada nó utilizar somente o próprio disco local. Isto permite que a velocidade de acesso aos dados seja a velocidade de acesso ao disco local, assim evitando que a velocidade da conexão seja um gargalo para requisições de dados que estão em outro nó. A redução de custo também é observada já que o hardware tem a comodidade de não ser compartilhado.

- Elasticidade: Capacidade de expandir dinamicamente ao adicionar um nó à malha.

- *Sharding*: Ao invés de guardar os dados em forma de pilha de localização de memória, gravações podem ser particionadas em fragmentos que são pequenos o suficiente para serem manejados por um único nó. Os fragmentos são replicados conforme o requisito e divididos quando ficam grandes.

- Replicação assíncrona : Adicionando um nó à malha, um conjunto de dados é replicado para esse nó. NoSQL é assíncrono durante a replicação de dados, o que torna as escritas rápidas e mais leves, já que são independentes do tráfego da rede. Entretanto os dados não são replicados imediatamente e podem não estar consistentes em certas janelas e, geralmente, bloqueios não estão disponíveis para a proteção de cópias.

- BASE ao invés de ACID: O acrônimo BASE (*Basic Availability, Soft State e Eventually consistecy*) é propositalmente um contraste com o paradigma ACID. Os bancos de dados NoSQL dão ênfase em "Disponibilidade" e "Performance". É difícil desenvolver um banco de dados que garanta as propriedades ACID, deste modo a consistência e o isolamento são muitas vezes perdidos, resultando em grande parte na abordagem BASE. Um dos conceitos por trás dos princípios BASE é que a consistência deve ser tratada pelo desenvolvedor e não pelo banco de dados.

Os princípios fundamentais dos bancos de dados NoSQL são o Teorema CAP, os princípios BASE e a consistência eventual que serão apresentados nas subseções seguintes.

2.3.3 Teorema CAP

Este teorema, apresentado por Eric Brewer [1], é composto por três propriedades: consistência forte (*Consistency*), disponibilidade(*Availability*) e tolerância a partição(*Partition tolerance*) das quais conjecturam que um sistema distribuído pode prover apenas duas propriedades, simultaneamente, como ilustrado esquematicamente na Figura 2.6.

As três propriedades do teorema CAP são descritas como segue [14, 26, 29]:

- Consistência forte: diz que todos os clientes veem a mesma versão de um dado armazenado independente de atualizações;
- Disponibilidade: é a capacidade de uma requisição ser atendida em qualquer instante de tempo, independente de falhas, ou seja, existir pelo menos uma cópia dos dados disponível todo o tempo; e
- Tolerância a partição: o sistema é capaz de manter-se disponível mesmo com falhas que leve alguns nós a ficarem inalcançáveis.

De acordo com o Teorema CAP, é possível escolher qual configuração é mais adequada para cada projeto de banco de dados. As três configurações são CA, AP e CP como mostra a Figura 2.6. A configuração CA utiliza primordialmente a abordagem de replicação para garantir a consistência e a disponibilidade, como é o caso dos bancos de dados relacionais. Os sistemas com configuração AP garante a disponibilidade e tolerância a falhas (CouchDB e SimpleDB são exemplos desta configuração). A Última configuração (CP) possui nós distribuídos e consistência, abrindo mão da disponibilidade, como exemplos, o MongoDB, Hbase e BigTable possuem essa configuração. Com a Web 2.0, é importante se optar pela disponibilidade, já que o número de requisições ao banco possui a tendência de ser cada vez maior, restando escolher entre CA e AP 14, [29].

Estas regras não são aplicadas de modo integral, isto quer dizer que a consistência, a disponibilidade e à tolerância a partição podem ser escolhidas em níveis variados dependendo da aplicação desejada. Se por exemplo o número de nós em um *cluster* diminuir, a disponibilidade também irá diminuir e, consequentemente, a tolerância à partição aumentará. O BigTable, usado pelo Google, e o HBase, que roda sobre o apache Hadoop, afirmam serem sempre consistente e altamente disponíveis. O Dynamo da Amazon, que é utilizado por serviços S3, e o banco de dados Cassandra abrem mão da consistência, estando a favor da disponibilidade e tolerância à partição.

[7] Adaptado de (https://changeaas.com/2014/08/20/nosql-and-its-use-in-ceilometer/)

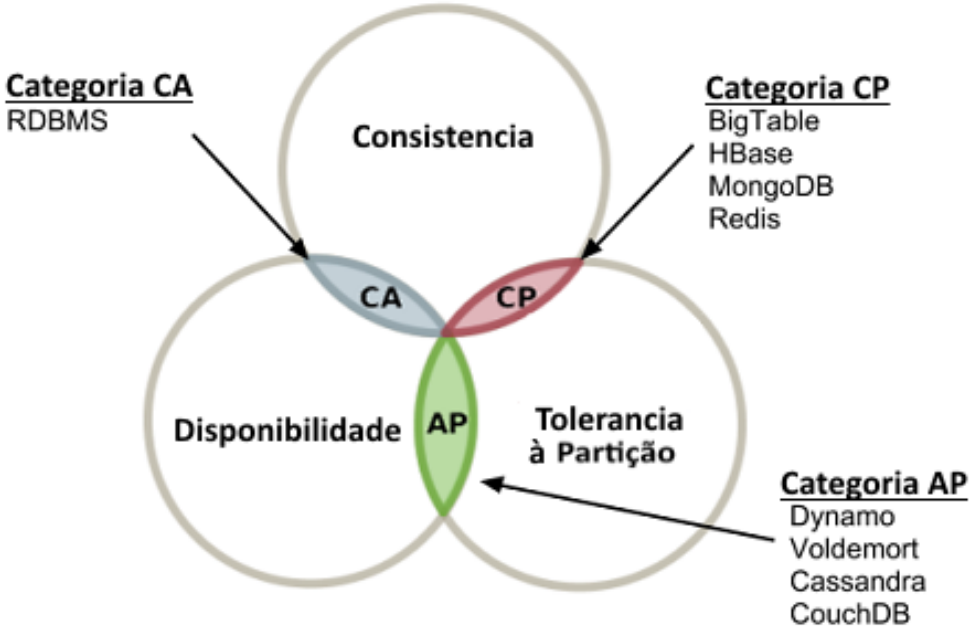


Figura 2.6: Diagrama CAP[7].

2.3.4 Princípios BASE

Os três princípios que o paradigma BASE possui e que serão explicados abaixo são: Basicamente disponível (*basically available*), Estado leve (*Soft state*) e Eventualmente Consistente (*Eventually consistent*). Estes três princípios são definidos [7, 31]:

- Basicamente Disponível: Este princípio diz que o banco de dados sempre está disponível para uma requisição. Basicamente Disponível significa que o sistema garante a propriedade disponibilidade do Teorema CAP, significa que o dado está disponível na maior parte do tempo.
- Estado Leve: Estado leve significa que as cópias dos dados podem estar inconsistentes e que o estado do sistema pode mudar com o passar do tempo. Neste estado a visão da consistência é relaxada. As informações no estado leve vão expirar se elas não forem renovadas.
- Eventualmente Consistente: Este último princípio diz que as atualizações se propagam para outros nós no *cluster* quando não há atualizações por um certo período de tempo e que eventualmente um dado pode estar atualizado. Dados são replicados quando um novo nó for adicionado. Estes princípios provém pelo menos 80% de consistência em um determinado instante de tempo sobre o fluxo das operações.

Estes princípios vêm da ênfase que os bancos de dados NoSQL dão à performance e à disponibilidade em detrimento da consistência e de que é difícil manter as propriedades ACID entre servidores distribuídos. BASE vem do paradigma de que os dados são distribuídos e a sincronização não é praticável da forma como é abordado nas propriedades ACID, sendo que o exato valor de um dado não é absolutamente necessário em todo o período logo após uma atualização. Os princípios ACID seguem o conceito de que ao

final de cada operação deve ser garantida a consistência dentro da transação, dito assim como abordagem pessimista. Por outro lado, BASE é dito otimista porque a consistência somente após a finalização da transação. Assim, NoSQL pode ser considerada como banco de dados "Não ACID". A Tabela 2.1 mostra a diferença entre o paradigma utilizado em banco de dados relacionais (ACID) em comparação com os bancos de dados NoSQL que utilizam o paradigma BASE [7, 31].

Tabela 2.1: Diferenças entre ACID e BASE.

ACID [C+A]	BASE[A+P]
Isolamento	Disponibilidade primeiro
Foco no "commit"	Melhor esforço
Transações aninhadas	Resposta aproximada
Pessimista	Otimista
Força a consistência no fim da transação	Aceita inconsistências temporárias
Evolução difícil	Mais simples, rápido, fácil evolução
Adequando para portais financeiros	Indicado para aplicações Web não financeiras
Seguro	Rápido
Compartilham algo (disco, memória)	Não compartilham
Escala vertical (limitada)	Escala horizontal (ilimitada)
Código simples, banco robusto	Código complexo, banco de dados simples
Máquina única	Um *cluster*
SQL	APIs personalizadas
Índices completos	Indexação realizada principalmente com chaves

2.3.5 Consistência Eventual

Consistência eventual é um modelo de consistência utilizado em programação paralela. Isto quer dizer que, em um período suficientemente longo e sem atualizações, é esperado que todas as atualizações sejam propagadas, eventualmente, pelo sistema e todos os nós fiquem consistentes. Essa consistência pode ser vista pelo cliente (*client-side consistency*) ou pelo servidor (*server-side consistency*). A consistência vista pelo cliente se refere a como e quando o observador vê a mudança feita nos dados do objeto. Existem três tipos de consistência vista pelo cliente [26]:

- Consistência forte: Após completar uma atualização recente, qualquer acesso subsequente vai retornar o valor atualizado;
- Consistência fraca: O sistema não garante que sempre será lido o valor atualizado como ocorre na consistência forte. O período entre a atualização e o momento em que é garantido que qualquer observador vai sempre ver o valor atualizado é chamado de janela de inconsistência;
- Consistência eventual: é uma forma específica da consistência fraca. Se não houver novas atualizações, o sistema garante que eventualmente todos os acessos irão

retornar o último valor atualizado. Se não ocorrer falhas, o tamanho máximo da janela de inconsistência pode ser determinado baseado nos fatos, tais como *delay* de comunicação e réplicas envolvidas no esquema de replicação.

A consistência eventual pode também ser vista em bancos de dados relacionais que realizam *backup* primário de segurança durante as sincronizações com réplicas, não sendo propriamente de sistemas distribuídos com muitas máquinas. A replicação síncrona é parte da transação e a replicação assíncrona envia ao *backup*, em formato de *logs*, atualizações de maneira atrasada. Caso o nó principal falhe, é necessário ler as informações a partir do *backup*, que é um caso de consistência eventual.

2.3.6 Modelos de Banco de Dados NoSQL

Existem diferentes modelos de banco de dados NoSQL, segundo [14, 24, 29] os principais tipos são: chave/valor, orientado a documento, orientado a grafo e orientado a coluna. Esses grupos representam os modelos que serão explanados adiante.

Banco de Dados com Chave-Valor

Essa abordagem relaciona uma chave com um valor. É possível armazenar um valor que vai de encontro com uma chave, que geralmente é alfanumérico. Uma única tabela armazena as chaves, funcionando como uma tabela *hash*. Os valores podem ser simplesmente textos ou estruturas mais complexas, como listas e conjuntos. Os registros são flexíveis, não sendo necessário definir o tipo dos dados, como é feito em bancos de dados relacionais. Este modelo de banco de dados suporta armazenamento em massa, apresenta alta concorrência e funciona muito bem utilizando memória cache [14, 29].

O sistema é simples e as chaves são feitas para serem únicas e identificarem rapidamente os dados dentro da tabela. Isto torna a consulta extremamente rápida, entretanto é necessária a chave para a consulta, limitado-a a uma correspondência exata da chave. A Figura 2.7 apresenta o esquema utilizado pela abordagem de armazenamento Chave-Valor.

Alguns exemplos de banco de dados NoSQL que possuem a arquitetura Chave-Valor são: Redis, MemcacheDB, Berkley DB, Voldemort, Dynamo e Riak [14, 24, 29].

Banco de Dados Orientado a Documento

Bancos de dados orientados a documento são estruturalmente similares aos bancos com chave-valor, entretanto o banco orientado a documento possui a semântica dos dados agregado à própria estrutura. Os dados são armazenados em estruturas como JSON, BSON, XML e YAML para padronizar o armazenamento da informação. O banco orientado a documento é levemente mais complexo ao esquema chave-valor e com um significado mais elevado para os dados que, no mínimo, podem encapsular os pares chave-valor em documentos. Ainda em comparação com o esquema chave/valor, os bancos orientados a documento podem indexar secundariamente o valor dos dados (diferentemente do esquema chave-valor que apenas pesquisa valores exclusivamente com a chave), facilitando a execução de uma busca.

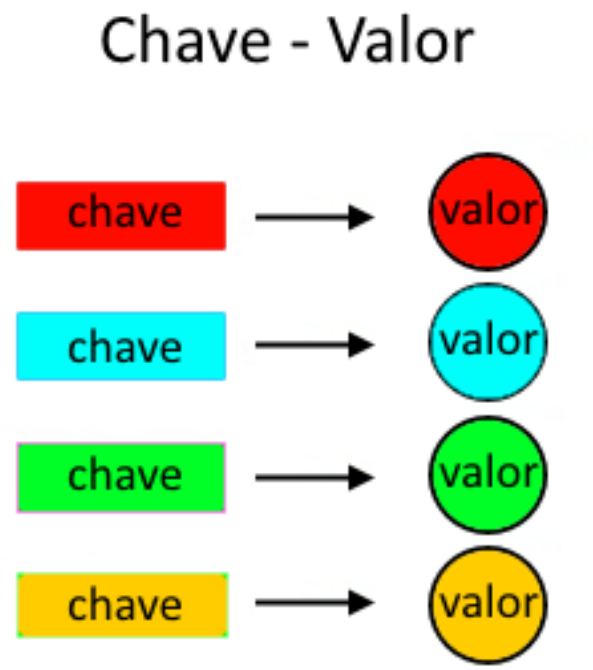


Figura 2.7: Modelo Chave-Valor.

Comparado com os bancos relacionais, os bancos orientados a documento possuem dados semi-estruturados, pois podem armazenar diferentes conjuntos de dados em cada registro (linha), em outras palavras, os registros não precisam aderir a um esquema definido, como é o caso das tabelas no banco relacional. Por esse motivo, o banco pode não suporta um esquema ou validar um esquema do documento.

Este tipo de abordagem funciona bem com grandes volumes de dados literais tais como texto, e-mail e dados espaçados (presença de *nulls*). As pesquisas sobre múltiplas entidades são mais simples em comparação ao banco relacional, pois a flexibilidade de não utilizar tabelas torna as consultas atuantes diretamente em todo o banco de dados.

É possível perceber a diferença da estrutura de um documento com uma tabela relacional na Figura 2.8 em que o documento não possui estrutura fixa (como é o caso da tabela do modelo relacional). Alguns exemplos de bancos orientados a documento são: MongoDB, CouchDB, Lotus Notes e BaseX [14, 24, 29].

Banco de Dados Orientado a Grafo

Bancos de dados orientados a grafo são bancos que possuem relacionamento entre seus registros e é o único tipo entre os quatros tipos de bancos NoSQL que possuem relacionamentos. Os nós representam os objetos conceituais, as bordas representam os relacionamentos e os atributos, descritos como chave-valor, são as propriedades de cada objeto. Este tipo de banco é direcionado para a representação visual, já que a informação fica mais clara de ser entendida em comparação aos outros gerenciadores de bancos NoSQL.

[8]Adaptado de (http://www.pwc.com/us/en/technology-forecast/2015/remapping-database-landscape/document-stores-business-model–transformation.html)

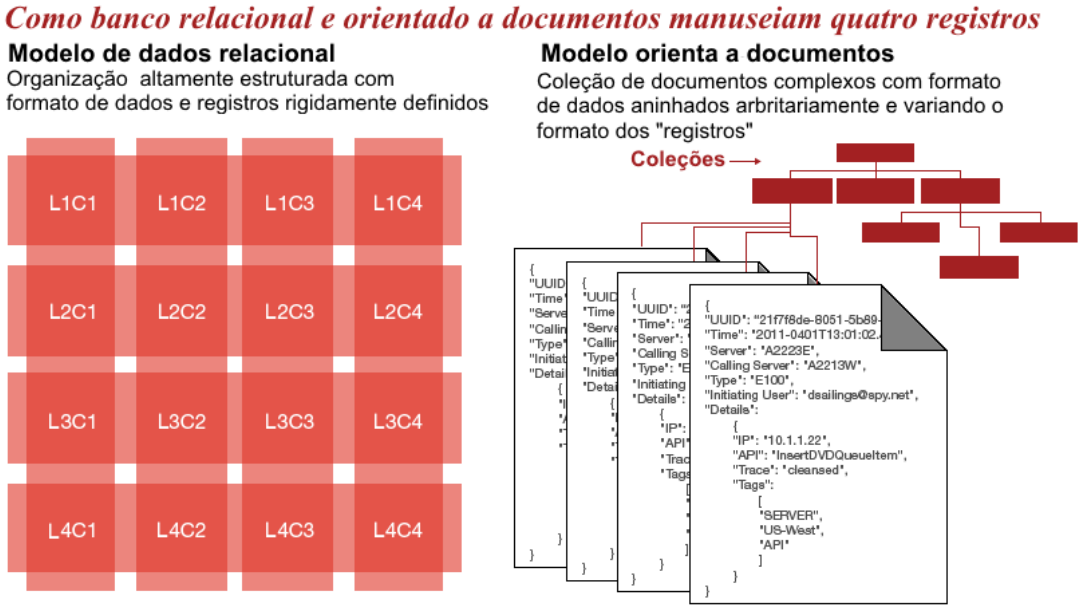


Figura 2.8: Modelo Relacional e Orientado a Documento [8].

Esta abordagem é nitidamente a melhor opção e indicação para aplicação que envolvam relacionamento entre dados. Estes bancos podem representar e cruzar dados em redes sociais, padrões de detecção em investigações forenses e gerar recomendações de produtos similares em lojas eletrônicas. Dois objetos podem ter múltiplos relacionamento distinto, além disto, esta abordagem é otimizada para percorrer relacionamentos e não executar consultas. Desta maneira, se for necessário fazer pesquisas utilizando linguagem de requisições, como o SQL, para analisar os relacionamentos e analisar seus valores, então é sugerido utilizar outro modelo de banco de dados.

Bancos orientados a grafos podem ser utilizados para resolver muitos problemas genéricos sobre vários domínios, como cálculo do menor caminho e caminho geodésico. Este banco é otimizado para processar relacionamentos entre dados e possuir fácil representação, manipulação e recuperação dos dados.

O exemplo da Figura 2.9 mostra claramente que cada nó pode conter atributos distintos, ou seja, possui flexibilidade quanto ao esquema, e cada nó pode estar relacionado com qualquer outro nó sem limite de relacionamentos entre ambos. Exemplos de bancos orientados a grafos são: Neo4j, FlockDB, SonesGraphDB, InfoGrid, AllegroGraph e InfiniteGraph [14, 24, 29].

Banco de Dados Orientado a Coluna

Assim como os bancos orientados a documentos, os bancos orientados a coluna também possuem múltiplos atributos por chave e são empregados distribuidamente em um *cluster*. Esse modelo não altera o armazenamento tradicional por linha, entretanto sua arquitetura é voltada para compressão de dados, sem compartilhamento de banco e hardware, e com um massivo processamento paralelo. Esse modelo pode ter alta performance com a análise de dados e processamento de inteligência de negócio.

[9]fonte: (http://blog.neo4j.org/2010/02/top-10-ways-to-get-to-know-neo4j.html)

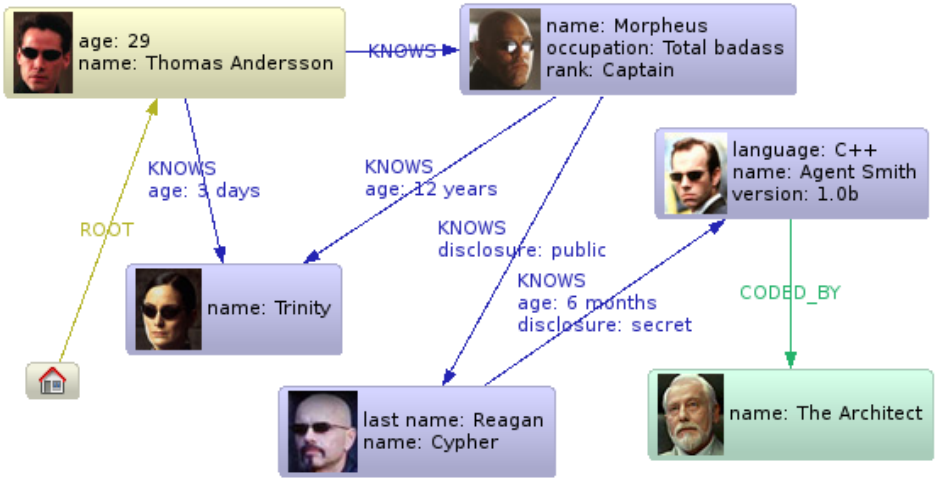


Figura 2.9: Modelo Orientado a Grafo [9].

Os bancos orientados a coluna possuem a vantagem de não precisarem preencher os campo das colunas adicionadas ao longo do tempo, o que dá mais flexibilidade para o banco. É possível ver na Figura 2.10 um exemplo de comparação entre bancos orientado a coluna e o banco relacional em relação a flexibilidade das linhas. Outra vantagem é a execução de requisições como soma e valor mínimo/máximo em um grande subconjuntos tendo em vista que trabalham apenas com um subconjunto de colunas (acesso parcial), não acessando as colunas próximas, o que torna a execução mais rápida. Além disso, quando se trata de dados com tipo uniforme e, na maioria, com o mesmo comprimento de *string*, esses dados podem ser comprimidos eficientemente.

As principais utilizações de banco de dados orientados a coluna são dados distribuídos, particularmente com versões de dados proporcionados por funções de *time-stamping*; grande escala de dados com processamento de dados orientado a *batch*: ordenação, conversão, *parsing* e etc.; e análise investigatória e de previsão executada por estatísticos e programadores experiente.

Alguns bancos orientados a colunas são: Cassandra, HBase, PNUTS, SimpleDB e Bigtable [14, 24, 29].

2.4 Banco de Dados NoSQL Cassandra

O banco orientado a coluna Cassandra foi originalmente desenvolvido pelo Facebook e atualmente é mantido pelo Apache como uma plataforma *open source*. Este é um banco distribuído, composto de vários nós em um *cluster*. O principal objetivo deste banco de dados é lidar com a alta taxa de requisições de gravação sem perder a eficiência da leitura, sendo projetado para funcionar em *hardwares* de baixo custo [33].

As principais características que a arquitetura do banco de dados Cassandra apresentam são [34]:

- arquitetura ponto-a-ponto;

[10] Adaptado de (31)

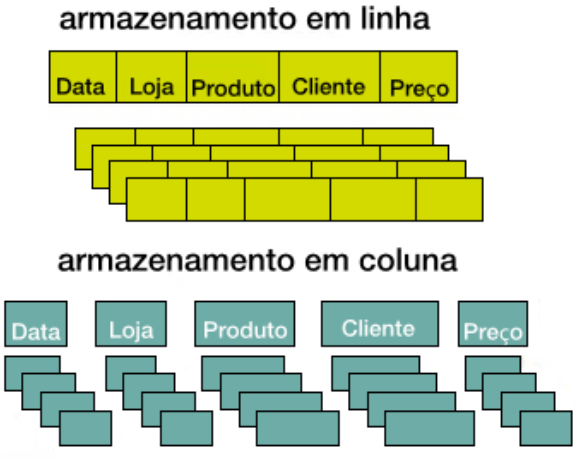


Figura 2.10: Modelo Orientado a Coluna[10].

- escalável e solução robusta para balanceamento de dados;
- detecção de nós e de falhas;
- recuperação de falha;
- sincronização de réplicas;
- manipulação de sobrecarga;
- transferência de estado;
- concorrência e planejamento de serviço;
- requisição de organização de dados;
- requisição de rota;
- sistema de monitoramento e de alarme; e
- gerenciamento de configuração.

As próximas subseções explicarão a maioria dessas características. O artigo "Cassandra - A Decentralized Structured Storage System" [34], publicado em 2009, mostra em detalhes como funciona a arquitetura, o particionamento, a persistência local, a replicação e o *membership*.

2.4.1 Particionamento

O sistema de particionamento do Cassandra visa implementar a escalabilidade, o balanceamento de dados e a requisição de rotas. O Cassandra utiliza partição de dados a partir de um "*consistent hash*", em outras palavras, uma função define randomicamente em qual nó do *cluster* os dados serão armazenados ao mesmo tempo em que a arquitetura garante a consistência destes dados. Para cada dado armazenado é atribuído uma chave que define o nó em que ele será armazenado. O alcance da função *hash* é definido circularmente (anel), ou seja, o último nó se liga ao primeiro, conforme mostra Figura 2.11. O nó

de cada dado é definido pela primeira posição disponível para o dado recuando pelo *cluster* no sentido horário a partir da posição definida pelo *hash*. Esta distribuição de dados por meio do *consistent hash* permite o processamento do nós no anel a fim de balancear a quantidade de dados sobre o *cluster*, mesmo se nós forem removidos ou adicionados [9, 40].

As chaves são utilizadas para as requisição de rotas, para isso se leva em conta que cada nó sabe da existência dos demais nós pertencentes ao *cluster*, então cada nó somente é responsável pela região entre ele e seu predecessor. Por este motivo, um nó com falha apenas afeta ao seu vizinho, deste modo esse sistema é dito que não possui um ponto crítico de falha (*Single Point Of Failure*), isto é, a falha de qualquer nó do *cluster* não afeta diretamente todo o sistema. Este modelo proporciona uma escalabilidade linear em relação as requisições de escrita linear, então dobrando-se o número de nós no *cluster*, o sistema será capaz de lidar com o dobro de operações por tempo suportadas, tornando essa arquitetura altamente escalável [6, 14]. O Cassandra foi planejado como *masterless*, o que quer dizer que a arquitetura não é do tipo mestre-escravo, mas sim ponto-a-ponto (peer-to-peer). Como os nós são iguais, a manutenção, a escalabilidade, adicionar e retirar nós se tornam mais fáceis em comparação com sistemas mestre-escravo.

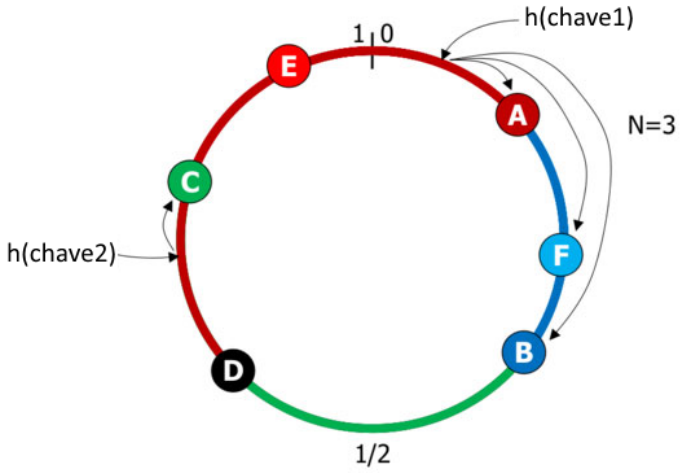


Figura 2.11: Representação do *cluster* no Cassandra [11].

2.4.2 Replicação

Esta parte da arquitetura do Cassandra implementa as funções que permitem o banco sincronizar suas réplicas e a recuperação de falhas. O Cassandra usa um mecanismo de replicação para alcançar alta disponibilidade. A partir do Teorema CAP, o Cassandra é categorizado com a configuração AP (disponibilidade e tolerância à partição), e assim como outros banco NoSQL, este banco se enquadra dentro dos princípios BASE que tem por objetivo aumentar o *throughput* de escrita do sistema, então este banco é basicamente disponível, sendo otimizado para ter um alto grau de operações de escrita por segundo [6].

Ainda em relação aos princípios BASE, o modelo de consistência eventual utilizado pelo Cassandra é fraco para manter as réplicas de um objeto. A configuração TTL (*time*

[11] Adaptado de (http://gotocon.com/dl/goto-aar-2013/slides/HayatoShimizu_ApacheCassandra20.pdf)

to live) é completamente um estado leve referente ao princípio BASE, já que se pode definir o tempo de vida de determinado dado armazenado.

O mecanismo de replicação utiliza N nós no *cluster* e N representa em quantos nós será copiado um conjunto de dados, então o fator de replicação é N. O nó ao qual foi designado uma determinada chave para um dado armazenado é definido como o nó coordenador de replicação, ou seja, ele define quais os outros $N-1$ nós a replicação abrange dentro do anel [7]. A Figura 2.11 mostra o *cluster* com o fator de replicação $N = 3$ e após o *consistent hash*, o coordenador da chave (chave1) é definido (A) e o dado é replicado para os outros $N-1$ nós (F e B).

As estratégias de replicação podem ser definidas no desenvolvimento e podem ser tanto do tipo *SimpleStrategy* quanto *NetworkTopologyStrategy*. A estratégia *SimpleStrategy* é indicada quando é utilizado apenas um *data center*. A estratégia adotada define a política de escolha das réplicas. Por exemplo, com um *data center*, as réplicas são armazenadas nos $N-1$ nós sucessores do nó coordenador. Com a política *NetworkTopologyStrategy*, as réplicas são distribuídas em diferentes *racks* em cada *data center*. Esta última configuração tolera a falha em um único nó para $N = 3$ [6].

A escrita de um dado no Cassandra é feita em todas as réplicas, o sistema roteia a requisição para cada réplica e espera um quorum de respostas desta réplicas para reconhecer a conclusão da escrita. A operação de leitura é semelhante à escrita. Dependendo da garantia de consistência adotada, o sistema recupera o dado a partir da réplica mais próxima ou roteia a requisição para todos as réplicas e espera por um quorum de respostas [7]. Caso ocorra alguma falha de partição na rede, o Cassandra torna o requerimento do quorum mais relaxado.

2.4.3 *Membership*

Dentro do *cluster*, os nós se comunicam através do protocolo Gossip [34]. Este é um protocolo ponto-a-ponto que é executado a cada segundo e envia as informações de estado de um nó para outros três nós no mesmo *cluster*. Cada nó envia as suas informações e as informações recebidas de outros nós, então rapidamente todos os nós passam a conhecer o estado de todos os nós. As mensagens antigas sobre um nó é sobrescrito pelas mensagens com uma versão mais nova.

Detecção de Falhas

A detecção de falhas consiste em determinar pelos outros nós se um nó está funcionando ou não. O Cassandra utiliza uma versão do algoritmo "Φ *Accrual Failure Detector*" [34]. A base deste algoritmo é que cada nó emite um valor que representa um nível Φ de dúvida. Este nível é um *threshold* que é ajustado dinamicamente de acordo com a rede, com a carga de trabalho e o histórico de cada nó. Para ajustar o valor de Φ, cada nó mantém uma janela de espera de mensagens *gossips* para cara outro nó do *cluster*. Para valores decrescentes, a chance de o nó estar com falha é maior, e para valores cada vez maior, a chance de estar com falha é menor.

Os nós estão sempre sujeito a falhas, seja por interrupções na rede ou falhas no hardware. Se um nó não for explicitamente removido pelo administrador, o Cassandra assume que este nó está transientemente inalcançável e por este motivo os outros nós irão periodicamente tentar contactar o nó com falha para verificar se este está novamente

ativo. Caso esse cenário ocorra, provavelmente ele pode ter perdido escritas e estar inconsistente. Um mecanismo de reparo é acionado. Os três mecanismos utilizados são [34]: *Hinted Handoff* para escritas mal-sucedidas, *Read Repair* para leituras desatualizadas e *Anti-entropy repair* para reparos e manutenções manuais de inconsistência em qualquer nó. A gravidade da interrupção irá definir o mecanismo de reparo. Caso um nó esteja definido como inalcançável, as mensagens são evitadas em momentos com várias operações.

Desta forma, a arquitetura proporciona a detecção de nós do *cluster*, de falhas e a transferência de estado entre os nós por parte de todos os computadores envolvidos. O fato de não se armazenar as informações sobre os nós em um determinador "nó mestre" por exemplo, evita que esse ponto seja um ponto crítico do sistema, não impedindo o funcionamento contínuo do banco mesmo se qualquer um dos nós falhar.

2.4.4 Persistência Local

A persistência local garante a recuperação de falhas como queda de energia por exemplo. Esta parte envolve o processo de escrita, leitura, atualização, deleção e compressão no nó local. Nesta subseção será explicado como o Cassandra escreve e lê os dados na persistência local.

Escrita de Dados

A escrita local, como mostra o esquema da Figura 2.12, possui quatro estágios [6, 9]:

- Registro de dado no *commit log*: O armazenamento no *commit log* é feito no disco apenas para evitar a perda dos dados por qualquer motivo de falha, mas não é o armazenamento final do dado. Isto garante a recuperação de falhas.
- Escrevendo os dados na *memtable*: Quando ocorre uma escrita, o Cassandra armazena os registros em uma estrutura de memoria chamada *memtable*. O Cassandra consulta por chave a *memtable* como *cache write-back* de partição de dados.
- Descarregando os dados da *memtable*: Quando os dados na *memtable* atigem uma certa quantidade, a *memtable* é colocada em uma fila para despejar os registros para o disco. Essa quantidade de dados é um *threeshold* configurável. O Cassandra ordena a *memtable* por *token* e escreve sequencialmente no disco. Também é criado um índice de partição no disco para mapear os *tokens* para o local no disco. É chamado *commit log replay* o processo de leitura do *commit log* para recuperar escritas perdidas. Dados no *commit log* são limpos após os dados correspondentes serem descarregados da *memtable* para uma *SSTable* no disco.
- Armazenando os dados no disco(*SSTables*): No Cassandra *SSTable*(*sorted string table*) é definido como um arquivo imutável para armazenar os dados da *memtable* e são armazenadas sequencialmente no disco e assim como as *memtable*, cada *SSTable* é mantida para cada tabela do Cassandra e não são reescritas após o despejo da *memtable*. Desta forma uma partição pode utilizar várias *SSTables*.

[12] Adaptado de (9)

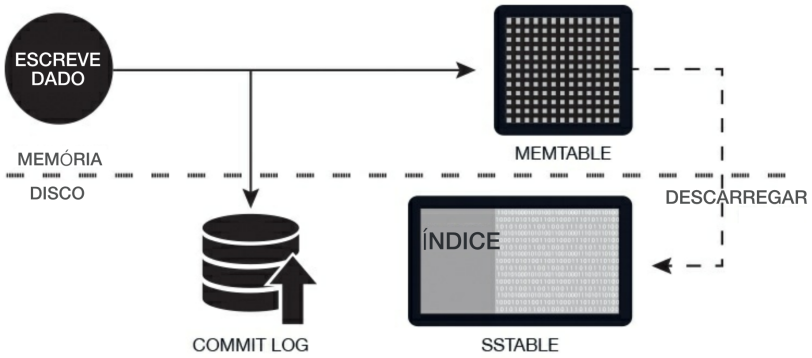


Figura 2.12: Esquematização da Escrita Local[12].

Leitura de Dados

Cada *SSTables* possui um índice de partição, um sumário de partição e um filtro *bloom* [6, 9]. O índice de partição contém lista das chaves de partição e a posição inicial das linhas no arquivo de dado escrito no disco. O sumário de partição é um índice de partição armazenado em memória. O filtro *bloom* é uma estrutura armazenada em memória para verificar se uma linha de dado existe na *memtable* antes de acessar as *SSTables*. Na Figura 2.13 é possível acompanhar o processo de leitura de um dado em um nó do *cluster*.

A leitura de um dado em um nó local começa na *memtable* e passa por alguns estágios até terminar em uma *SSTable*. Estas buscas envolvem várias fases que podem conter acessos a *caches* em memória para maximar a velocidade de leitura. O processo de leitura envolve os seguintes estágios:

- Verificar a *memtable*: O primeiro acesso é à tabela em memória para verificar se alguma partição de dados requeridas é encontrada na *memtable* e fundir com os dados faltantes das *SSTables*.
- Verificar a cache de linha, se disponível: A *cache* de linha armazena na memória subconjuntos da partição de dados que estão no disco. A quantidade de linhas armazenadas e o tamanho da *cache* pode ser configurada. Essas linhas são frequentemente acessadas por serem os dados mais recentes que foram acessados nas *SSTables* e ao acessar o disco, as linhas lidas são fundidas e gravadas na *cache* de linhas. Este estágio pode evitar dois *seeks* do disco.
- Verificar o filtro *bloom*: Este filtro sumariza as chaves, ainda na memória, e previne procurar os dados em arquivos que não os contenham. O filtro escolhe as partições mais prováveis de terem a partição de dados buscada. Cada *SSTable* é associada a um filtro *bloom* e que pode estabelecer que certa partição não está nessa *SSTable* ou a probabilidade de estar. Se o filtro não rejeitar, o Casandra verificar a *cache* de chaves de partição.

- Verificar a *cache* de chave de partição, se disponível: esta *cache* armazena índices de partição em memória. Encontrar a partição do dado pela *cache* de chave previne um *seek* no disco. Se a chave estiver na *cache*, então o Cassandra obtém o *offset* de compressão e faz a leitura do dado no disco. Caso a chave de partição não seja encontrada, o próximo passo é procurar no sumário de partição.
- Sumário de partição: Esta é uma estrutura armazenada em memória que contém amostras dos índices de partição o qual possui todas as chaves de partição. O sumário salva uma amostra a cada X índices. Desta forma, mesmo não sabendo exatamente qual a chave do dado a ser lido, uma passada pelo índice de partição diminui a busca para achar a partição dos dados. Após achar a localização dos X índices aonde possivelmente se encontra o índice desejado, a busca continua no índice de partição.
- Índice de partição: O índice de partição é armazenado no disco e possui todas as chaves de partição mapeado para o seu *offset*. Caso o sumário de partição encontre a região do índice, um único *seek* é suficiente para fazer uma busca sequencial e encontra a chave. A chave encontrada é utilizada pelo mapa de *offset* para encontrar o bloco comprimido no disco que contém o dado de leitura.
- Mapa de *Offset* da Compressão: este mapa de *offset* possui ponteiros para o local do disco em que será encontrado o dado. Pode ser acessado tanto pela *cache* de chave de partição quanto pelo índice de partição. O mapa é mantido em memória e pode chegar a 1-3 GB para cada terabyte comprimido.
- Obter os dados: Finaliza trazendo os dados da *SSTable* correspondente do disco.

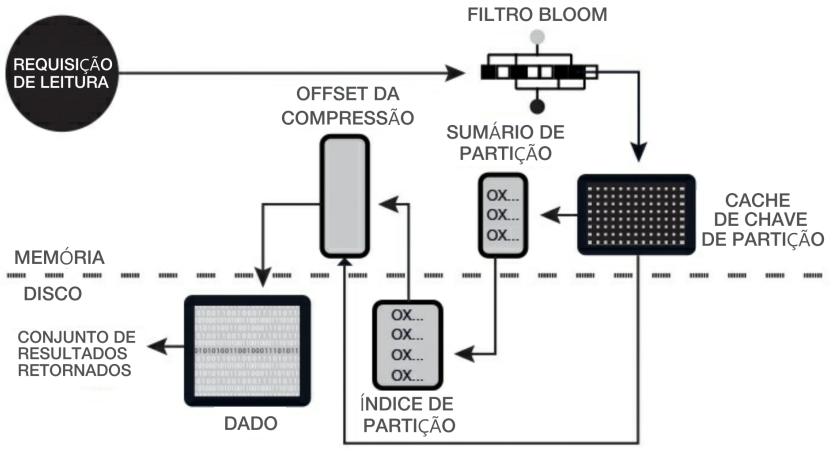


Figura 2.13: Esquematização da Leitura Local[12].

Capítulo 3

Arquitetura de Armazenamento e Processamento Espacial

Neste capítulo é definida a arquitetura de armazenamento para dados geográficos com o Cassandra. A Seção 3.1 propõe uma estrutura de armazenamento para os dados geográficos no Cassandra. A Seção 3.2 apresenta a construção de um banco Cassandra com suporte para armazenamento de dados espaciais. A Seção 3.4 deste capítulo apresenta os testes e as avaliações feitas com dados geométricos.

3.1 Camadas de Persistência e Processamento Espacial

O banco NoSQL Cassandra ainda não possui um suporte nativo para dados geográficos, como é o caso do PostGIS[1] para o sistema gerenciador de banco de dados relacional SQL PostgreSQL[2]. Então, a solução adotada para armazenar dados espaciais foi a implementação de uma camada de persistência em Java para o processamento de dados geográficos. Desta forma, ao invés do banco ficar com a responsabilidade de realizar as consultas geográficas, a aplicação terá que assumir a tarefa do geoprocessamento.

Em conjunto com o sistema gerenciador de banco de dados Cassandra, foi utilizada a API ESRI/geometry-api-java [10]. Esta API é disponibilizada como um projeto aberto para desenvolvedores e possui tanto a definição de variáveis geométricas quanto operações sobre estas geometrias. Desta forma, é possível, por exemplo, realizar uma consulta que retorne um valor booleano se duas geometrias de entrada se tocam ou se sobrepõe, fazer o cálculo de área e perímetro. Outra função desta API é ler dados no formato GeoJSON [11, 25]. GeoJSON é um padrão aberto que utiliza a estrutura JSON com regras específicas para manipular dados geométricos, contendo o tipo de geometria espacial e os pontos pertencentes a essa geometria. Como algumas plataformas de processamento geoespacial possuem suporte para o formato GeoJSON (QGis[3], Geoserver[4], MapBox[5]), foi utilizada essa estrutura para importar dados para o banco, e exportar o resultado de consultas espaciais [25].

[1] http://postgis.net/
[2] https://www.postgresql.org/
[3] http://www.qgis.org/
[4] http://geoserver.org/
[5] https://www.mapbox.com/

A Figura 3.1 apresenta a arquitetura de processamento de dados geográficos proposto neste trabalho para o banco de dados NoSQL Cassandra. A arquitetura de armazenamento de dados geográficos apresentada nesta monografia foi desenvolvida em duas camadas: a primeira é de processamento espacial; e a segunda é de persistência.

A camada de processamento espacial que consiste em receber dados no formato GeoJSON e transformá-los em geometrias através da API Esri/geometry-api-java. As geometrias são os principais elementos desta camada e todas as funções que envolvem processamento espacial ficam nesta camada. Após ler a entrada e criar uma lista de geometrias com essas informações, esta camada envia esta lista para a camada de persistência.

A camada de persistência é formada pelo armazenamento de dados convencionais e por dados espaciais. Acompanhando a Figura 3.1 é possível ver que os dados de entrada/saída são compostos por dados convencionais e por dados geométricos. Os dados convencionais fazem parte de um conjunto de informações que já são bem conhecidos nos bancos de dados relacionais, por exemplo variáveis do tipo *string*, *integer*, *float*. Os dados geométricos são definidos pela API Esri/geometry-api-java, cada um com seus pontos correspondentes, que podem ser extraído para serem gravados no SGBD Cassandra.

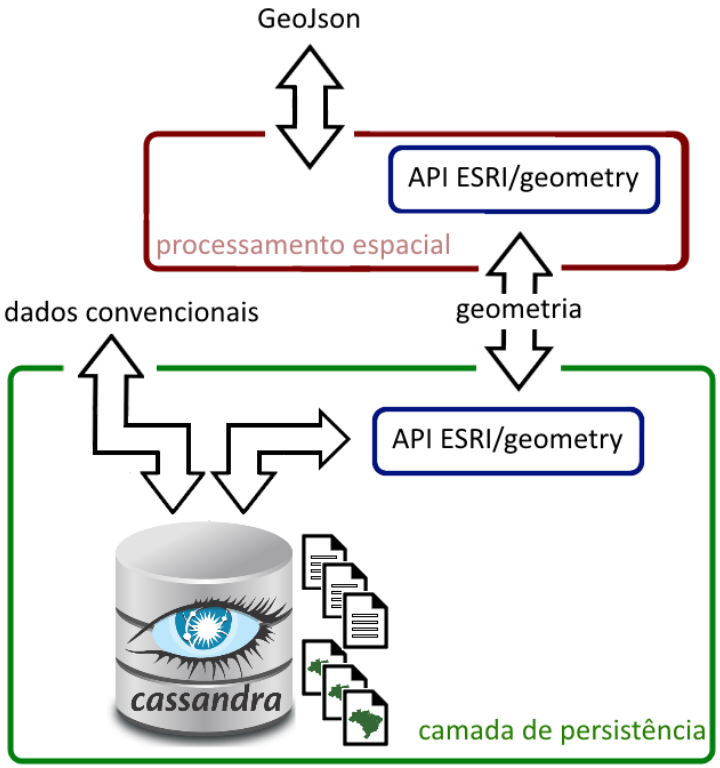


Figura 3.1: Arquitetura em Camadas.

Na arquitetura apresentada na Figura fig:camadas, toda a troca de dados geográficos é realizada com o formato GeoJSON. Para ilustrar o formato de dado geográficos GeoJSON é apresentado o código 3.1.

Código 3.1: Dados no Formato GeoJSON.

```
{"type":"FeatureCollection","features":
   [
   {"type":"Feature","id":"0", "properties":{"name":"Pentágono"},
      "geometry":{"type":"MultiPolygon","coordinates":
   [[[[1.8,3],[3,6],[0,8],[-3,6],[-1.8,3],[1.8,3]]],
   [[[4,6],[5,7],[6,6],[5,5],[4,6]]]]},

   {"type":"Feature","id":"1","properties":{"name":"Quadrado"},
   "geometry":{"type":"Polygon","coordinates":
   [[[3,-3],[3,3],[-3,3],[-3,-3],[3,-3]],[[1,1],[1,0],[-1,0],[-1,1],[1,1]]]},

   {"type":"Feature","id":"2","properties":{"name":"Triangulo"},
   "geometry":{"type":"Polygon","coordinates":
   [[[-2,-1],[-6,5],[-6,-1],[-2,-1]]]},

   {"type":"Feature","id":"3","properties":{"name":"Hexágono"},
   "geometry":{"type":"Polygon","coordinates":
   [[[0,4.5],[1,5],[1,6],[0,6.5],[-1,6],[-1,5],[0,4.5]]]},

   {"type":"Feature","id":"4","properties":{"name":"Linha dupla"},
   "geometry":{"type":"MultiLineString","coordinates":
   [[[-8,-0.5],[-5,-0.5]],[[-8,-1.5],[-5,-1.5]]]},

   {"type":"Feature","id":"5","properties":{"name":"Linha curva"},
   "geometry":{"type":"LineString","coordinates":[[4.5,4],[4.5,-2],[4,-3]]},

   {"type":"Feature","id":"6","properties":{"name":"Linha"},
   "geometry":{"type":"LineString","coordinates":[[0,3],[6,3]]},
   ]
}
```

Todos os objetos da estrutura GeoJSON possuem um campo obrigatório denominado *"type"*. Este campo funciona como metadados para identificar que se trata de uma coleção (*"FeatureCollection"* ou *"GeometryCollection"*) ou apenas um objeto (*"Feature"*). Caso seja algum tipo de coleção, um campo é acrescentado para descrever todos os objetos da coleção (*"features"* ou *"geometries"*). Cada um dos objetos da coleção são delimitados por chaves e separados por vírgulas. Todos os objetos do tipo *"Feature"* possuem obrigatoriamente o campo *"properties"* e o campo *"geomerty"*. O campo *"properties"* possui qualquer quantidade de propriedades no formato de chave-valor para esse objeto. O campo *"geomerty"* contém o campo *"type"* que descreve o tipo da geometria e o campo *"coordinates"* que descreve a estrutura da geometria nas especificações GeoJSON. O campo *"coordinates"* é no formato (x, y, z). Quaisquer outros campos podem ser criados para um objeto e, geralmente, é criado o campo "id" para identificar cada *"Feature"* [25].

3.2 Modelagem dos Dados Geográficos

Esta monografia analisa a utilização do banco de dados NoSQL Cassandra em dados espaciais vetoriais. Desa forma, a seguir é apresentado como manipular ponto, linha e polígono nessa arquitetura.

3.2.1 Estrutura do Armazenamento Geográfico

O ponto é a estrutura básica necessária para criar uma geometria e para este projeto foi definido o tipo "*POINT*" no banco Cassandra como um *datatype* definido pelo usuário conforme o Código 3.2:

Código 3.2: Definição de POINT

```
CREATE TYPE IF NOT EXISTS POINT ( x: DOUBLE, y: DOUBLE)
```

Este novo tipo de dado é definido como tendo duas variáveis do tipo *double*: "x" e "y". Assim, a representação da latitude e da longitude é feita em um plano bidimensional.

Uma lista pode representar o conjunto de pontos que fazem parte de uma determinada geometria, assim, a definição de geometria é ampliada no banco ao se criar uma lista do tipo PINTOPINTO. A declaração é apresentada no Código 3.3:

Código 3.3: Lista de POINT

```
LIST< POINT < x: DOUBLE, y: DOUBLE> >
```

No caso de uma geometria do tipo ponto, a lista terá apenas um ponto, uma linha terá dois pontos, e os polígonos terão, pelo menos, três pontos.

Outra lista mais externa deste campo representa a lista de geometrias deste registro, assim podendo formar multi-pontos, multi-linhas e multi-polígonos. A modelagem final do campo geometria é apresentada no Código 3.4:

Código 3.4: Modelagem Final do Campo Geometria.

```
LIST< LIST< POINT< x: DOUBLE, y: DOUBLE> > >
```

Desta forma, para criar uma tabela com o tipos geométricos no Cassandra é necessário que uma coluna da tabela tenha um campo geométrico. Sendo assim, o Código 3.5 em CQL apresenta uma tabela com um campo geométrico.

Código 3.5: Tabela Geométrica.

```
CREATE TABLE tabela_geometrica
(nome text,
geometria_tipo text,
geometria list< frozen< list< frozen<POINT> >>>,
PRIMARY KEY(nome));
```

O campo geometria suporta tanto pontos quanto linhas e polígonos. O campo nome é a chave primária das geometrias e o campo geometria_tipo identifica qual é o tipo da geometria. Os exemplos a seguir mostram possíveis inserções de dados geométricos no banco. Para adicionar um novo registro de um ponto, o comando CQL, apresentado no Código 3.6, é executado:

Código 3.6: Inserção de um Ponto.

```
INSERT into tabela_geometrica(nome, geometria) VALUES('Ponto de ônibus Via
    N1', 'Point',[[(-15.7834562,-47.912797)]]);
```

Para adicionar um novo registro de linha, o comando CQL, apresentado no Código 3.7, é executado:

Código 3.7: Inserção de uma Linha.

```
INSERT into tabela_geometrica (nome, geometria) VALUES('Via L2 S',
    'Polyline',[[(-15.8402554,-47.9218061),(-15.839592,-47.9209301)]]);
```

Além disso, para adicionar um novo registro de polígono, o comando CQL, apresentado no Código 3.8, é executado:

Código 3.8: Inserção de um Polígono.

```
INSERT into tabela_geometrica (nome, geometria) VALUES('Torre de TV',
    'Polygon', [[(-15.7908382,-47.8928913),(-15.7905284,-47.8931998),
      (-15.790451,-47.8927358)]]);
```

3.2.2 Modelagem das Tabelas Geográficas

Para armazenar dados geográficos no banco de dados, foi criado um banco NoSQL Cassandra com duas tabelas, como é possível ver no *keyspace geometry* da Figura 3.2. As tabelas possuem os campos nome, longitude, latitude, geometria, geometria_tipo e

propriedades. Também foram criados dois índices secundários para cada tabela: um para as chaves e outro para os valores do campo propriedades.

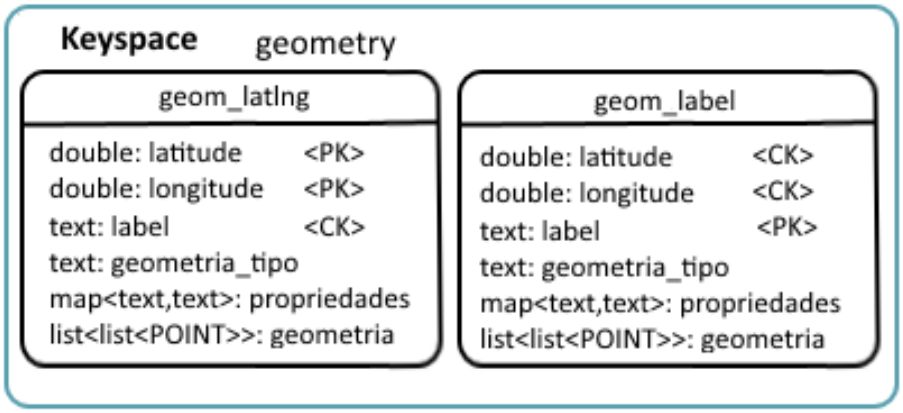


Figura 3.2: Modelagem de Banco de Dados no Cassandra.

Os campos latitude e longitude armazenam o valor numérico do primeiro ponto da geometria a ser armazenada e o campo nome é uma *string* para caracterizar a geometria. Na tabela geom_label, a chave primária consiste em uma chave de partição (*partition key*): nome; e pelas chaves de *cluster* (*clustering keys*): latitude e longitude. A tabela geom_latlng foi criada com a seguinte chave primária: a chave de partição é uma composição dos campos latitude e longitude; e a chave de *cluster* é o campo nome.

Como o Cassandra trabalha com chaves compostas, existe a possibilidade de armazenar dois lugares com o mesmo nome que estejam em diferentes localizações. Desta forma, é possível armazenar geometrias similares, como é o caso de estradas e caminhos por exemplo.

O campo geometria_tipo armazena uma *string* do tipo de geometria desse registro. Os tipos podem ser: *Point* (ponto), *MultiPoint* (multi-ponto), *Polyline* (multi-linha), *Polygon* (polígono) e *MultiPolygon* (multi-polígono).

O campo propriedades armazena as informações de uma geometria. Ele é um mapa de valores de *string* para *string*, isto é, o primeiro campo define a propriedade e o segundo define o valor atribuído para essa propriedade. Então, por exemplo, pode-se criar uma propriedade "estado" que define o estado que uma cidade pertence com seus respectivo valor ("Paraná", "Sergipe", etc.).

O campo geometria possui a informação geométrica e é composta por uma lista que contém os pontos da geometria, envolvido por uma outra lista mais externa, como foi definido na Seção 3.2.1.

A razão de criar duas tabelas com as mesmas informações se deve ao fato de não existirem chaves estrangeiras no banco Cassandra, então para realizar diferentes tipos de consultas são necessárias diferentes chaves primárias. Desta forma, para um outro tipo de consulta, outra tabela com redundância de dados seria criada para suprir essa consulta. Com as duas tabelas criadas, é possível localizar geometrias pelo nome (tabela geom_label) ou pela localização (geom_latlng). Os índices criados no campo propriedade permitem consultar geometrias que contenham algum tipo de propriedade, como por exemplo ser uma construção ou uma área geológica.

3.3 Teste de Consultas Vetoriais Espaciais

Como forma de validar o armazenamento no banco Casandra, o geoprocessamento por parte da API Esri/geometry-api-java e a estrutura GeoJSON, foram feitos alguns teste para analisar as operações da API, simultaneamente, com informações em formato GeoJSON e inserção e recuperação de dados no banco.

Para analisar melhor o comportamento das consultas vetoriais espacias, decidiu-se criar diferentes geometrias no formato GeoJSON conforme o Apêndice B.1, e o Apêndice B.2. A representação destas geometrias é mostrada na Figura 3.4. As geometrias *id*: 0, 1, 2 são polígonos, *id*: 3 é um multi-polígono, *id*: 4 é uma multi-linha, *id*: 5, 6 são linhas, *id*: 7, 9, 10, 11, 12 são pontos e *id*: 8 é um multi-ponto. Esta numeração foi utilizada para identificar as geometrias após as operações.

Assim, foi criada uma aplicação escrita na linguagem Java para desenvlver a conexão com o Cassandra e também para carregar a API Esri/geometry-api-java. O fluxo de execução do programa está esquematizado na Figura 3.3. Através da API Esri/geometry-api-java, o campo *"coordinates"* da estrutura GeoJSON foi convertido em suas respectivas geometrias a partir da função "OperatorImportFromGeoJson". Após esta operação, todas as geometrias descritas no arquivo foram importadas para o SGBD Cassandra a partir dos pontos e do tipo das geometrias. Para cada Operação lógica entre as geometrias, foram recuperadas todas as geometrias do banco e comparadas uma a uma entre elas. Todas as geometrias com teste positivo foram convertidas para o formato GeoJSON a partir da função "OperatorExportToGeoJson".

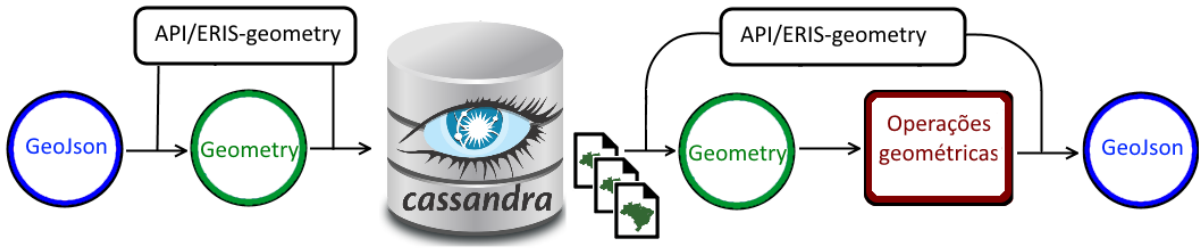


Figura 3.3: Esquema de Teste de Consultas Vetoriais Espaciais.

O resultado foi armazenado em um arquivo e com o apoio do software livre QGis[6] foi possível visualizar diretamente os resultados obtidos. Os resultados positivos estão ressaltados em laranja com pontos ou traços e as demais geometrias estão em preto ou branco.

Foram testados os seguintes operadores lógicos par a par entre todas as geometrias: *Contains*, *Crosses*, *Disjoint*, *Intersects*, *Overlaps*, *Touch* e *Within*. As geometrias que fazem parte de um conjunto (multi-ponto, multi-linha e multi-polígono) são tradadas como uma só geometria e, portanto, possuem resultado positivo se qualquer um dos subconjuntos retornar o resultado positivo.

[6]http://www.qgis.org/

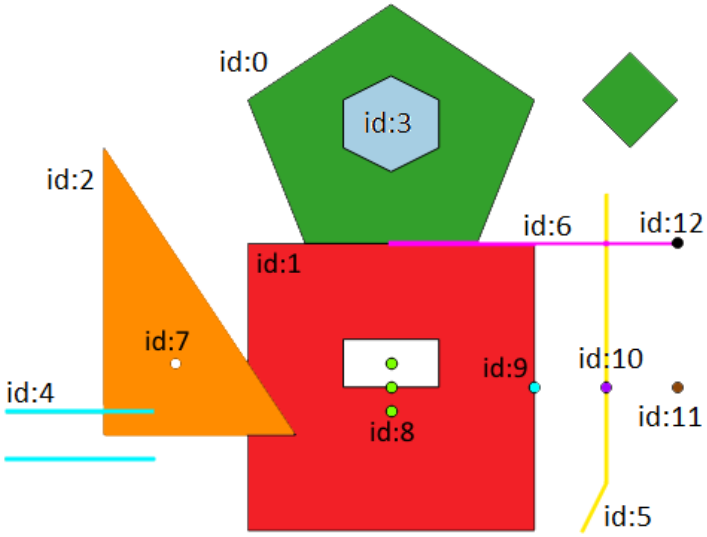


Figura 3.4: Geometrias - GeoJSON.

Ø Operador *Contains*

"Uma geometria contém outra se a outra geometria é um subconjunto desta e seus interiores possuem, pelo menos, um ponto em comum.*Contains* é o inverso de *Within*" [10].

Para o exemplo, o resultado é mostrado na Figura 3.5 em que as gravuras representam as geometrias que contém pelo menos uma geometria. O triângulo 2 contém o ponto 7, a linha 5 contém o ponto 10, e o pentágono 0 contém o hexágono 3.

A API Esri/geometry-api-java trata os dois pontos extremos das linhas como pontos de borda, que difere de pontos do interior, deste modo, o ponto 12 não é considerado interior da linha 6.

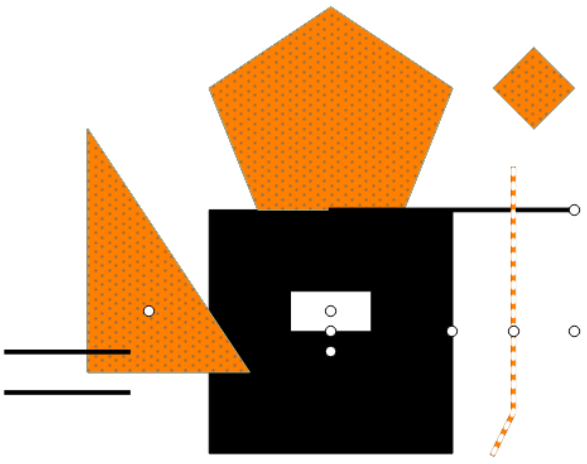


Figura 3.5: Geometrias - Operador *Contains*.

Operador *Crosses*

"Duas linhas se cruzam se elas se encontram somente em pontos, e no mínimo um dos pontos compartilhados é interno de ambas linhas. Uma linha e um polígono se cruzam se

uma parte conectada da linha está parcialmente dentro e parcialmente fora" [10].

Para o exemplo, a Figura 3.6 mostra que o triangulo 2 está cruzando com a multi-linha 4, as linhas 5 e 6 também estão se cruzando e por fim o quadrado 1 e o multi-ponto 8 estão se cruzando. Note que parte do multi-pontos está numa região fora do polígono e parte está dentro.

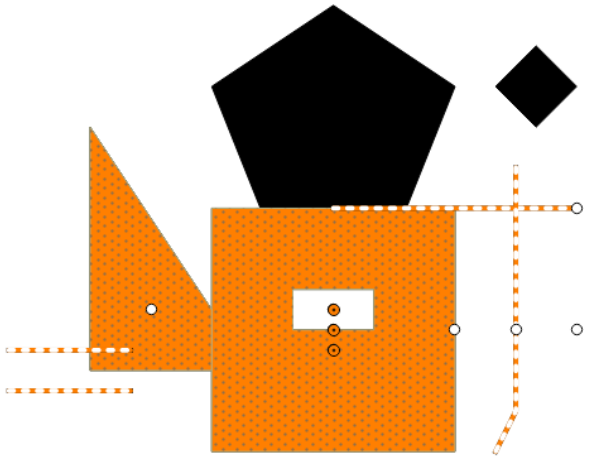


Figura 3.6: Geometrias - Operador *Crosses*.

Operador *Disjoint*

"Duas geometrias são disjuntas se elas não possuem nenhum ponto em comum. *Disjoint* é o inverso de *Intersect*. *Disjoint* é o operador mais eficiente e é garantido funcionar mesmo em geometrias complexas " [10].

Com este operador, todas as geometrias serão selecionadas, pois existe pelo menos uma geometria o qual qualquer outra geometria não possui pontos em comum. O ponto 11 é suficiente para retornar um resultado positivo para todo o conjunto, conforme mostra a Figura 3.7.

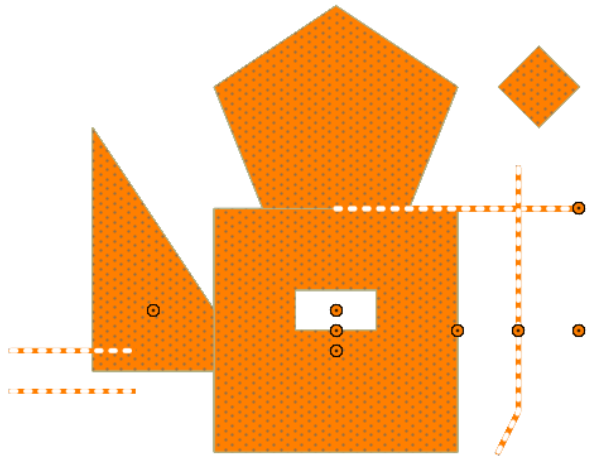


Figura 3.7: Geometrias - Operador *Disjoint*.

Operador *Intersects*

"Duas geometrias se intersectam se elas compartilham no mínimo um ponto em comum.*Intersects* é o inverso de *Disjoint*" [10].

O resultado desta operação, apresentado na Figura 3.8, mostra que apenas o ponto 12 não possui outra geometria em comum.

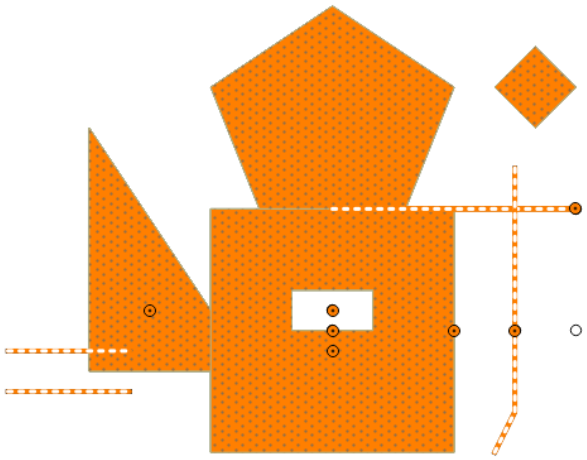


Figura 3.8: Geometrias - Operador *Intersects*.

Operador *Overlaps*

"Duas geometrias se sobrepõem se elas possuem a mesma dimensão e a intersecção delas também possuem a mesma dimensão, mas é diferente de ambas geometrias" [10].

Esta operação é facilmente visualizado pela Figura 3.9 em que o triangulo 1 e o quadrado 2 estão sobrepostos. Esta operação também trabalha com linhas.

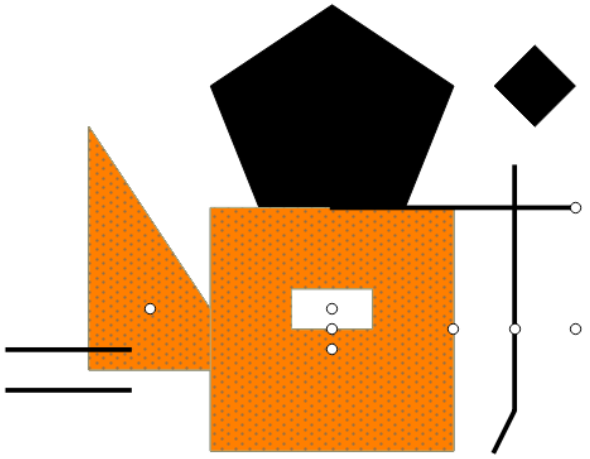


Figura 3.9: Geometrias - Operador *Overlaps*.

Operador *Touch*

"Duas geometrias se tocam se a intersecção dos seus interiores é vazia, mas a intersecção das duas geometrias não é vazia" [10].

O pentágono 0, o quadrado 1 e a linha 6 estão se tocando, o ponto 9 está tocando o quadrado 1 e a linha 6 está tocando o ponto 12. O ponto 12 está na extremidade da linha 6 e por este motivo é considerado que não está no interior da linha. O resultado é mostrado na Figura 3.10.

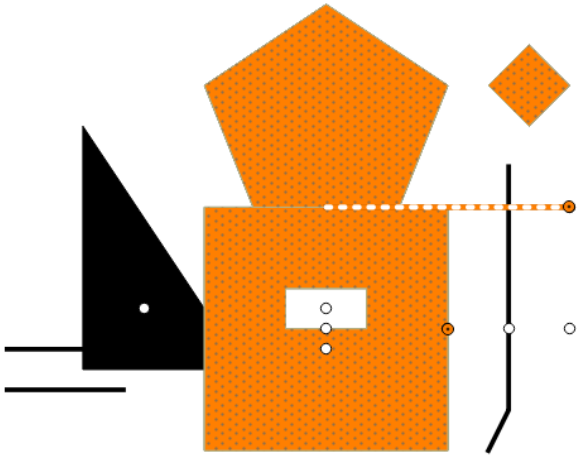


Figura 3.10: Geometrias - Operador *Touch*.

Operador *Within*

"Uma geometria está dentro de outra se esta é um subconjunto de outra geometria e seus interiores possuem pelo menos um ponto em comum. *Within* é o inverso de *Contains*" [10].

O hexágono 3 está dentro do pentágono 0, o ponto 7 está dentro do triangulo 2 e o ponto 10 está dentro da linha 5. A Figura 3.11 mostra o resultado obtido deste operador sobre as geometrias importadas.

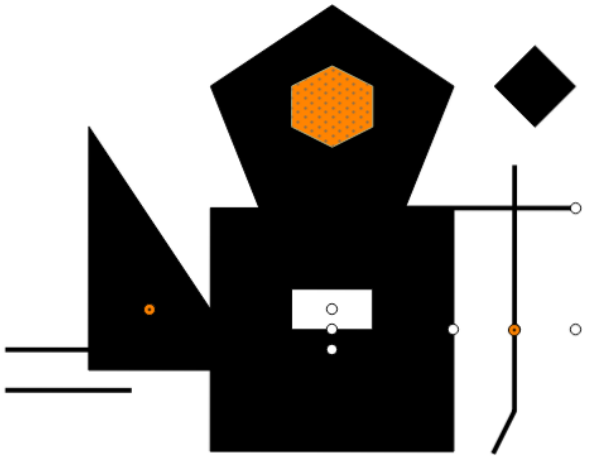


Figura 3.11: Geometrias - Operador *Within*.

3.4 Avaliação do Banco de Dados Geográfico com Dados Reais

Também foram feitos alguns testes com dados reais para validar o armazenamento geográfico com esta abordagem de modelagem espacial no banco NoSQL Cassandra. Foram utilizadas as informações de base de dados reais do OpenStreetMap[7].

3.4.1 Dados do OpenStreetMap

O OpenStreetMap é uma plataforma livre contendo dados geográficos que é mantida e atualizada pelos próprios usuários. Esta plataforma possui uma comunidade com mais de 2 milhões de contribuidores registrados, os quais ajudam na construção dos mapas. O uso dos mapas é aberto, devendo apenas ao desenvolvedor creditar autoria ao OpenStreetMap e aos colaboradores envolvidos [21].

Os dados nativos desta plataforma estão no formato XML. Esses dados também são encontrados em outros formatos (e.g. PBF, o5m, JSON, Level0L) e existem ferramentas de conversão entre esses formatos (e.g. Osm4j, Osmium, Osmconvert). O processo de importação dos dados geográficos está esquematizado na Figura 3.12. Os dados foram primeiramente obtidos diretamente do site do OpenStreetMap no formato XML. O segundo passo foi converter esses dados para o formato GeoJSON, e para isto foi utilizada uma ferramenta pública denominada "osmtogeojson[8]". Com o arquivo em formato GeoJSON, a transformação para geometrias é por meio da API Esri/geometry-api-java, da mesma maneira como é feito com os testes na Seção 3.3. Após esse processo, as geometrias são importadas para o banco Cassandra a partir dos pontos que compõem as geometrias.

Os dados utilizados para a importação foram a área do Distrito Federal dividido em doze arquivos no formato XML, e as sub-regiões foram selecionadas à mão e exportada do OpenStreetMap[7], tentando ser fiel ao território real. Após a inserção, foi feito a exportação de toda a informação no formato GeoJSON. A representação visual do Distrito Federal, gerada pela ferramenta QGis a partir do arquivo GeoJSON, é apresentada na Figura 3.13.

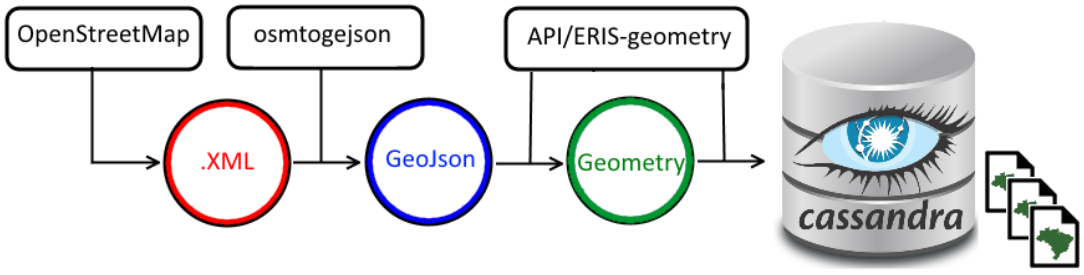


Figura 3.12: Esquema de Conversão de Dados.

A Tabela 3.1 mostra o tamanho de cada arquivo e a quantidade de geometrias importadas para o banco. O arquivo "df.geojson" corresponde ao arquivo com todas as geometrias importadas para o banco.

[7] https://www.openstreetmap.org/
[8] https://github.com/tyrasd/osmtogeojson

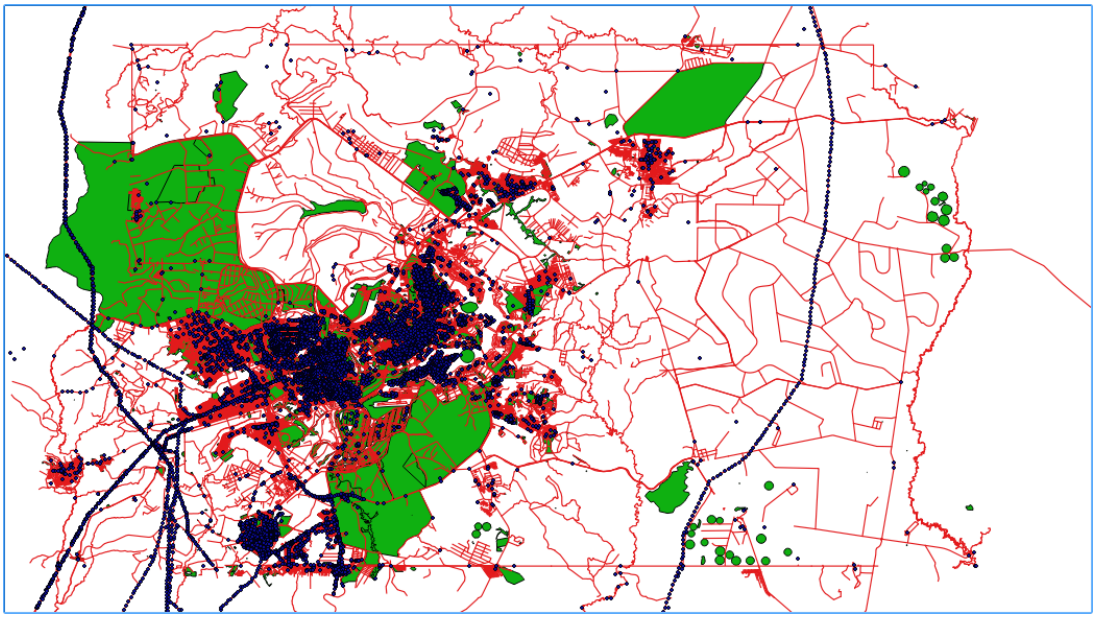


Figura 3.13: Representação dos Dados do DF.

Apesar de conseguir armazenar os dados no banco Cassandra e representar espaços geográficos a partir do formato GeoJSON, não é possível fazer consultas da mesma maneira a qual é feita com a linguagem SQL convencional. Diferentemente da extensão espacial do PostgreSQL (PostGIS), que aceita a própria linguagem SQL para criar consultas espaciais além de possuir muitas funções dedicadas ao processamento geométrico, o banco Cassandra não possui uma extensão espacial nativa e por esse motivo somente serão criadas as funções necessárias para uma dada aplicação. A seguir são apresentados alguns exemplos de consultas que foram criadas. Os cálculos de distância são feitos com base nas coordenadas geográficas (latitude e longitude). Assim, foi utilizado um fator de conversão de 111,13 quilômetros para 1 grau de latitude/longitude.

3.4.2 Teste Espacial de Distância

Este teste tem o objetivo de medir o comprimento de todas as vias selecionadas no mapa da Figura 3.14 em laranja. Primeiro foi feito uma busca no banco de dados Cassandra com consulta na linguagem CQL. Como pode ser observado, a linguagem CQL é semelhante a linguagem SQL. A consulta é apresentada no Código 3.9:

Código 3.9: Consulta Espacial 1.

```
SELECT *
FROM geom_label
WHERE propriedades contains key 'highway';
```

Esta consulta retorna todos os registros que possuem alguma propriedade com a chave *"highway"*. *Highway* é uma *tag* genérica do OpenStreetMap e serve para designar vários

Tabela 3.1: Arquivos GeoJSON.

Arquivo	Tamanho(*Megabytes*)	Geometrias
df0.geojson	1,12	5.400
df1.geojson	3,34	15.270
df2.geojson	3,05	12.970
df3.geojson	2,93	12.138
df4.geojson	2,72	11.268
df5.geojson	2,32	8.474
df6.geojson	3,47	14.734
df7.geojson	3,39	13.524
df8.geojson	2,54	9.829
df9.geojson	1,84	6.754
df10.geojson	0,9	2.852
df11.geojson	0,23	138
df.geojson	21,4	89.974

tipos de caminhos como por exemplo rodovias primárias, rodovias secundárias, ciclovias, rotatórias, ligação entre pistas e ruas residenciais. Após essa consulta, foram separados para o cálculo do comprimento somente as geometrias que são do tipo linha/multi-linha. A função "calculateLength2D()" pertencente à API Esri/geometry-api-java realiza o cálculo do comprimento para qualquer geometria. A quantidade de geometrias calculadas foi de 43.493 e o comprimento total obtido foi de 13.224,3724 quilômetros de extensão.

3.4.3 Teste Espacial de Proximidade

Aqui é avaliado o cálculo de proximidade entre as geometrias. Foi criada uma função que recebe uma coordenada de referência, a distância máxima em metros e uma chave para o campo propriedades para realizar a busca deste tipo de geometria. A consulta deste teste é apresentado no Código 3.10:

Código 3.10: Consulta Espacial 2.

```
SELECT *
FROM geom_label
WHERE propriedades contains key 'school';
```

A coordenada geográfica utilizada foi -47.88322,-15.79401, que é o centro de Brasília, e a distância foi de 10000, ou seja, 10 quilômetros. Assim, todas as escolas num raio de 10 quilômetros deste ponto são selecionadas. O operador "OperatorDistance" da API Esri/geometry-api-java foi utilizado para calcular a distância entre o ponto e as geometrias recuperadas do banco. Foram recuperadas 220 geometrias com a propriedade "school". A Figura 3.15a mostra o resultado obtido em laranja, e a Figura 3.15b é o mesmo resultado

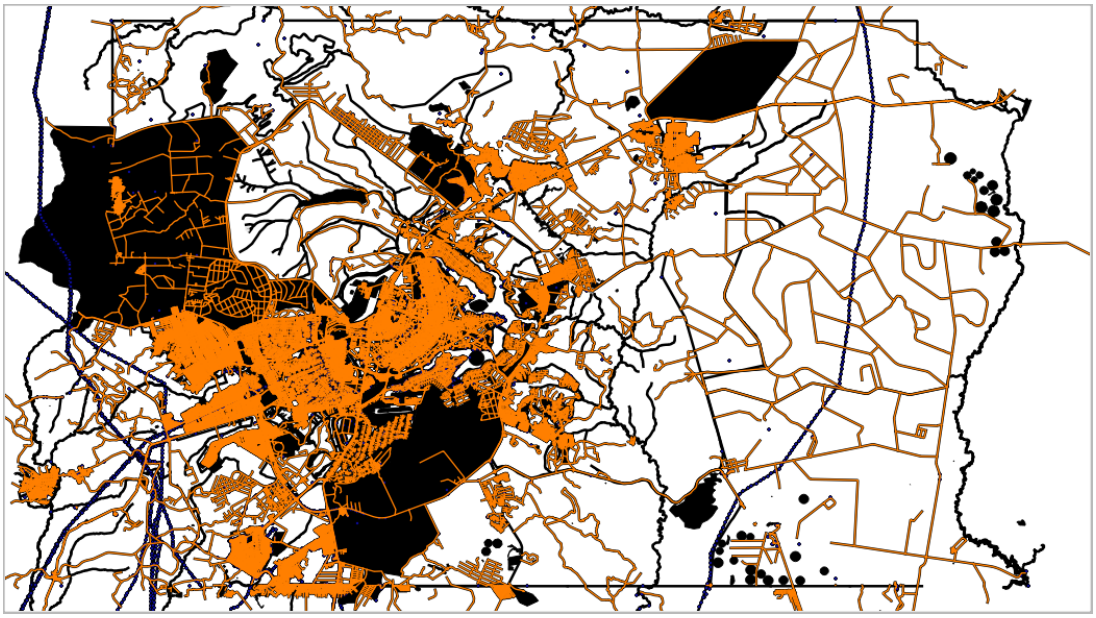


Figura 3.14: Vias do DF.

com maior *zoom*. Note que as escolas podem ser representadas tanto como pontos quanto por polígonos.

3.4.4 Teste Espacial de Área

Este último teste foi feito para determinar a área total do campus Darcy Ribeiro e a área total construída. Primeiro foi recuperado a geometria que representa a área do campus com a consulta apresentada no Código 3.11:

Código 3.11: Consulta Espacial 3.

```
SELECT *
FROM geom_label
WHERE label='Universidade de Brasília - Campus Universitário Darcy Ribeiro'
```

Após isso, foi desenvolvida uma função para comparar as outras geometrias. Uma geometria foi comparada a outra usando o operador *Contains* para achar as geometrias contidas na geometria do campus.

A Figura 3.16 mostra a geometria do campus em bege e o resultado desta operação em laranja. Com a função "calculateArea2D()" pertencente à API Esri/geometry-api-java a área do campus e a área construída foram obtidas. A área total do campus é de 2,9735642 quilômetros quadrados (note que este registro possui a área apenas da Gleba A do campus) e a área total construída foi de 0,6592683 quilômetros quadrados com 439 geometrias contabilizadas. A porcentagem de área construída no campus Darcy Ribeiro é de 22.171%.

3.5 Análise dos Resultados

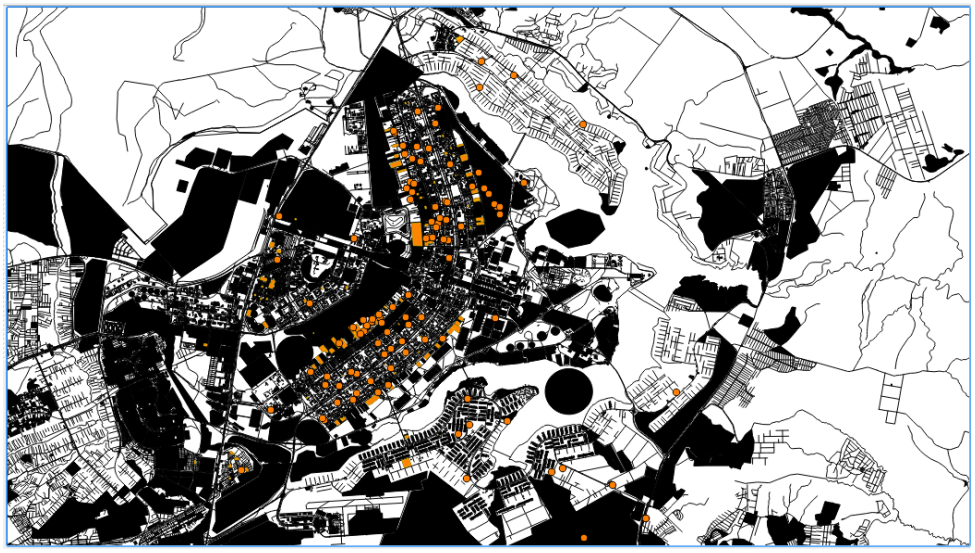
Como pode ser observado a partir dos teste realizados, a arquitetura proposta suporta o armazenamento tanto de dados convencionais quanto de dados geográficos e, em conjunto com uma camada de processamento espacial, é possível desenvolver consultas espaciais com esses dados geográficos. Foi demonstrado neste capítulo as funcionalidade dos operadores da API Esri/geometry-api-java sobre geometrias e também que é possível realizar consultas espaciais utilizando mapas com dados reais compostos de milhares de geometrias.

Todavia, nen todos os operadores são intuitivos, e para utilizá-los deve-se observar todas as regras que envolvem intersecção, união e bordas. Além disso, a API Esri/geometry-api-jav também fornece mecanismos para criar novas operações relacionais entre geometrias, além de suportar outros formatos de dados como: WKT, WKB, REST Json e Esri Shape.

É interessante notar que a empresa privada ESRI[9] (Environmental Systems Research Institute) foi fundada em 1969 e sempre desenvolveu e vendeu softwares com alta qualidade voltados para o geoprocessamento, entretanto, aparentemente está mudando sua política e criando projetos abertos com ferramentas para auxiliar desenvolvedores, como é o caso deste trabalho.

[9] www.esri.com

(a) Escolas em laranja.



(b) Escolas em laranja (*zoom*).

Figura 3.15: Escolas a até 10km do Centro de Brasília.

Figura 3.16: Área Construída do Campus Darcy Ribeiro.

Capítulo 4

Estudo de Caso

Para avaliar a arquitetura proposta nesta monografia, este capítulo apresenta um estudo de caso com um sistema de informação geográfico com participação popular a partir de um aplicativo móvel denominado "ConsultaOpinião" [8]. O objetivo deste software é "obter opiniões dos usuários através de um sistema capaz de georreferenciar esses dados, e exibi-los ao gestor desses serviços em um mapa". Ele possui cadastro de contas de usuário, estabelecimentos, formulário de avaliação de estabelecimento, e tanto a avaliação quanto o estabelecimento possuem uma tabela para definir seu tipo [8]. O objetivo específico do estudo de caso apresentado neste capítulo é mensurar o tempo da inserção e da recuperação de dados. O início deste capítulo apresenta a arquitetura da aplicação. A Seção 4.3 mostra o desenvolvimento da aplicação. A Seção 4.4 descreve os testes realizados.

A aplicação já foi anteriormente desenvolvida em [8] para coletar informações sobre instituições públicas, avaliar com uma nota estas instituições e calcular uma nota final que será apresentada a um gestor do serviço público.

4.1 Arquitetura da Aplicação

A aplicação ConsultaOpinião foi desenvolvida com a linguagem Java e funciona com uma interface móvel no dispositivo final do usuário com sistema operacional Android[1], o qual apresenta três *layouts* diferentes: tela de autenticação; tela de cadastro e tela de avaliação. Outra interface é apresentada para o gestor público que pode ser visualizado por qualquer *browser*. Tanto a aplicação móvel quanto a visualização por parte do gestor público fazem requisições para um banco de dados a partir de uma conexão *HTTP* entre o usuário/gestor e o servidor. A Figura 4.1 mostra a arquitetura da aplicação composta por três camadas: a Camada de Persistência, a Camada de Comunicação e Interface Móvel, e a Interface Web [8].

A Interface Móvel apresenta as informações recebidas e envia as avaliações para o do banco de dados. Esta parte não teve alterações em relação à aplicação do ConsultaOpinião.

A Interface Web apresenta as informações fornecidas pelo banco de dados ao gestor público.

[1] https://www.android.com/

A Camada de Comunicação recebe as informações geradas pelos usuários e envia as informações para um servidor central onde está localizado o banco de dados. As requisições da aplicação e do gestor público são recuperadas no banco e enviadas.

A camada da base é a Camada de Persistência. Originalmente, foi desenvolvida suportando o banco de dados relacional PostgreSQL em conjunto com a sua extensão espacial, o PostGIS. No estudo de caso apresentado nesta monografia, foi modificada a Camada de Persistência. A Camada de Persistência é composta pela instalação de um sistema gerenciador de banco de dados NoSQL Cassandra. Para a parte de dados geográficos foi utilizada a arquitetura apresentada na Seção 3.2 do capítulo anterior, e a modelagem final ficou conforme a Figura 4.4.

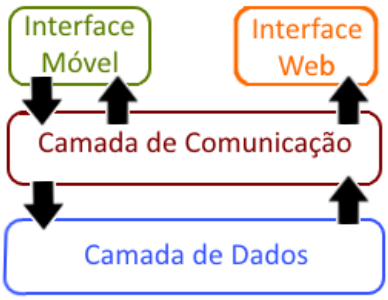


Figura 4.1: Arquitetura da Aplicação.

4.2 Modelagem do Banco

O banco Cassandra é capaz de suportar múltiplas *keyspaces*. Um *cluster* é formado por várias *keyspaces* que podem ser acessadas ao mesmo tempo por uma aplicação. Cada *keyspace* comporta inúmeras tabelas, e para este trabalho foram desenvolvidas duas *keyspaces*, uma para os dados convencionais e outra para os dados geográficos

4.2.1 Modelagem dos Dados Convencionais

Os dados convencionais deste estudo estão apresentados na Figura 4.2 para um SGBD relacional. Este modelo representa o banco relacional da aplicação móvel ConsultaOpinião. [8].

No artigo proposto por Lu et al. [35], uma ferramenta de modelagem de dados chamada KDM (*The Kashlev Data Modeler*) é proposta para auxiliar a modelagem de um banco de dados Cassandra a partir de um modelo relacional padrão.

O Apêndice A.1 mostra o *script* CQL (*Cassandra Query Language*) gerado pela ferramenta. As tabelas geradas por esta ferramenta estão representadas na Figura 4.3. Para esse modelo relacional, o mesmo usuário usuário pode ter várias avaliações no mesmo estabelecimento, então o tipo *map* foi trocado pelo tipo *tuple* que suporta repetições nos dois campos.

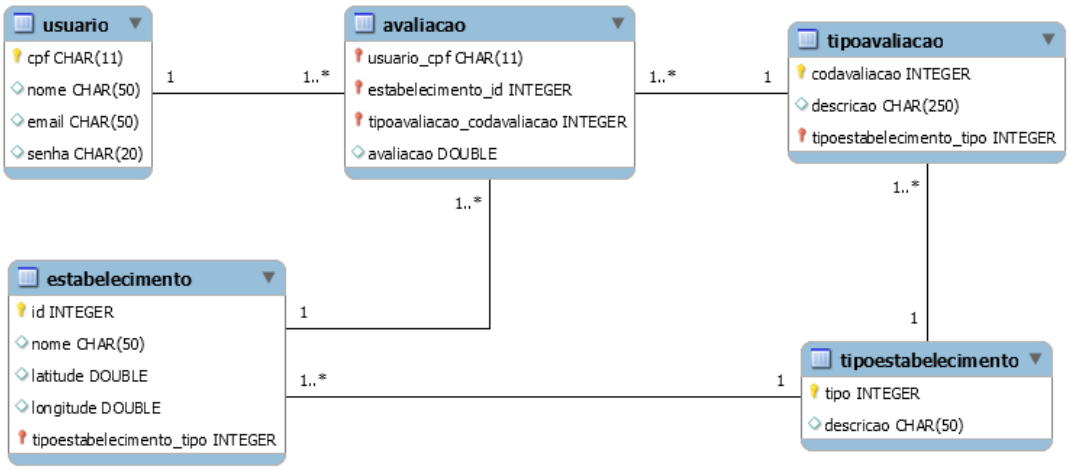


Figura 4.2: Modelo Relacional Utilizado.

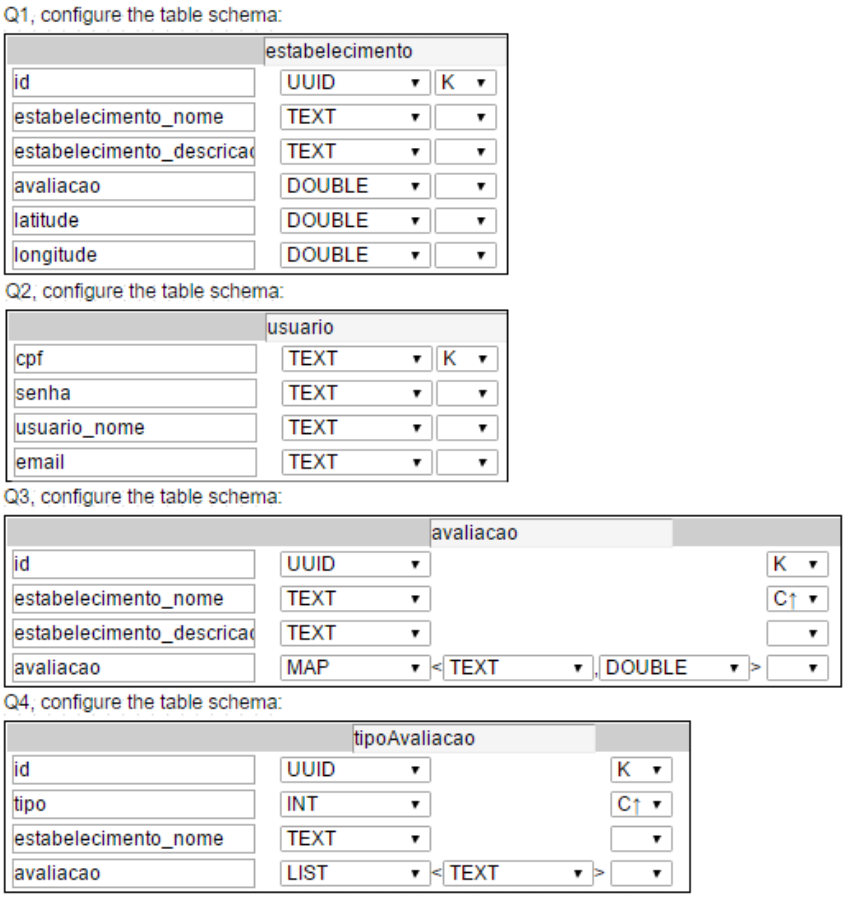


Figura 4.3: Banco Cassandra Gerado pela Ferramenta KDM.

4.2.2 Modelagem Final do Banco

As tabelas de dados geográficos neste estudo de caso são as mesmas desenvolvidas na Seção 3.4. A Figura 4.4 mostra como ficou a estrutura final do banco desenvolvido.

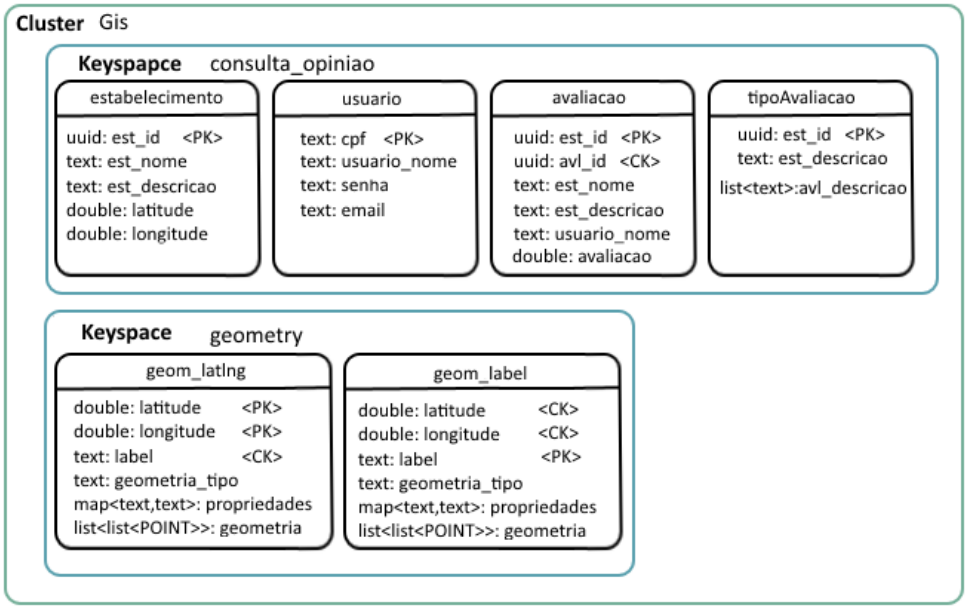


Figura 4.4: Modelagem de Dados do Banco Cassandra.

4.3 Fluxo de Processamento do ConsultaOpinião

A aplicação final segue o fluxograma da Figura 4.5, na qual é possível observar que a Interface Móvel apresenta o *layout* de *login*, cadastro e de avaliação do estabelecimento, além de apresentar o mapa do OpenStreetMap com os devidos estabelecimentos cadastrados [8].

Na tela de autenticação, o usuário tem a opção de colocar seus dados ou de se cadastrar. Caso não possua um conta, ele é redirecionado a um novo *layout* ao pressionar o botão "Cadastrar" (veja a Figura 4.6a). O usuário informa seu nome, CPF e senha para se cadastrar e essas informações são enviadas ao servidor central (mostrado na Figura 4.6b).

Após colocar seus dados e autenticar com o servidor central, o usuário recebe as informações dos estabelecimentos cadastrados em conjunto com o mapa da API do OpenStreetMap para dispositivos móveis (Figura 4.7a). Esta aplicação cadastrou escolas e hospitais públicos. Neste momento o usuário pode selecionar o estabelecimento que queira avaliar e caso esteja muito longe do estabelecimento selecionado, ela não poderá avaliar esse estabelecimento. Se estiver dentro de uma distância mínima, o usuário tem a opção de prosseguir para o *layout* seguinte (Figura 4.7b), e dar notas de 0 a 5 para cada pergunta que este estabelecimento possui. Ao final, o usuário envia sua avaliação para o servidor pressionando o botão enviar.

A Interface Web também apresenta um mapa urbano com os ícones dos estabelecimentos e recuperar as avaliações e nome do usuário para qualquer estabelecimento selecionado. Ao carregar o mapa, cada estabelecimento é preenchido por um círculo verde ou vermelho, dependendo da nota que o estabelecimento possui (Figura 4.8a). O gestor público seleciona qualquer estabelecimento e é redirecionado para uma nova página onde é listado o nome do estabelecimento em conjunto com todas as avaliações feitas pelos usuários (Figura 4.8b).

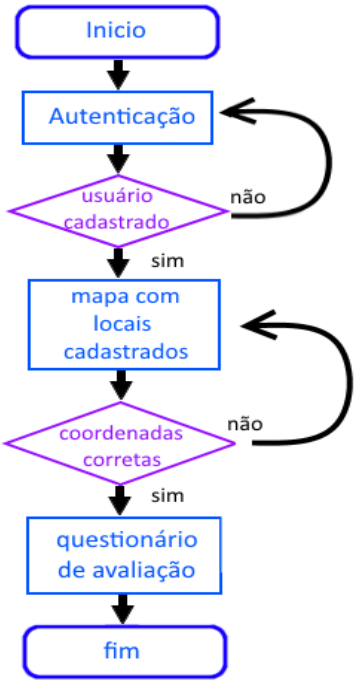
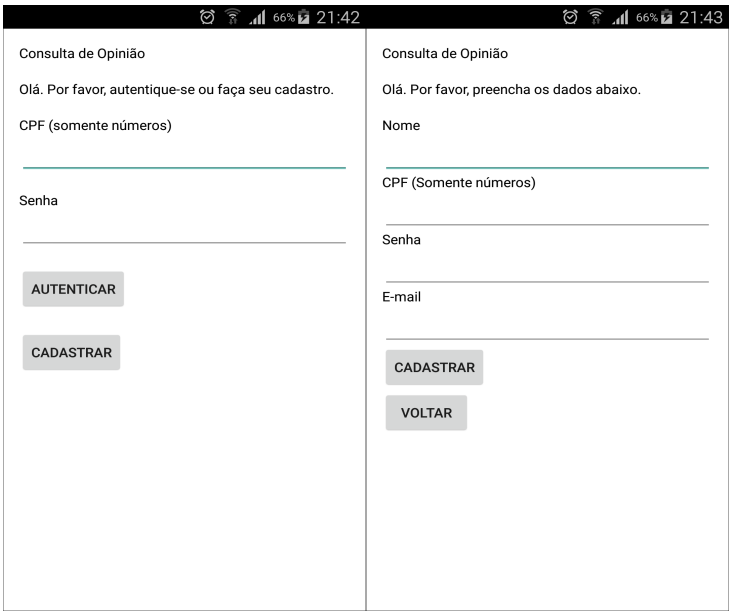


Figura 4.5: Fluxograma da Aplicação Móvel.

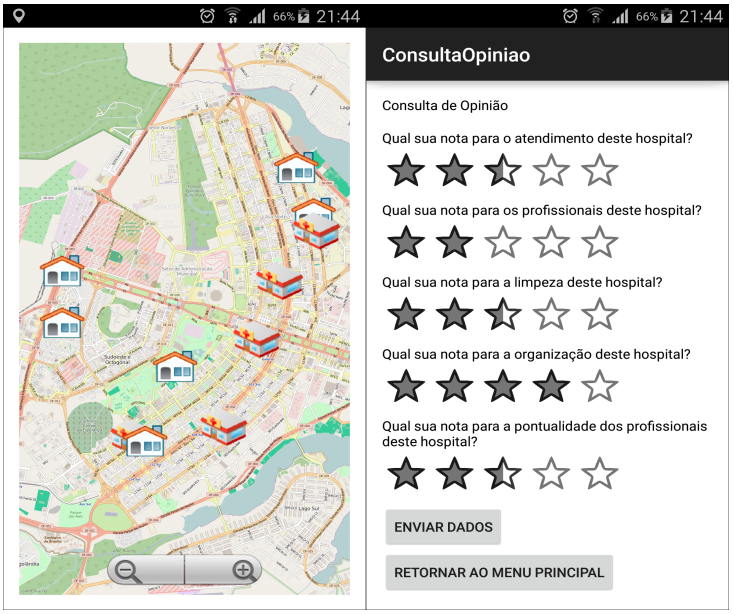


(a) Tela de Autenticação. (b) Tela de Cadastro.

Figura 4.6: Interface da Aplicação Móvel.

4.4 Testes

Para avaliar o desempenho da arquitetura com um SIGPP com muitos usuários, foi desenvolvido um simulador para gerar avaliações para cada estabelecimento. Os valores das avaliações são gerados aleatoriamente e o usuário da avaliação é selecionado arbitrariamente, a partir dos usuários cadastrados no banco.



(a) Mapa Virtual. (b) Tela de Avaliação.

Figura 4.7: Interface da Aplicação Móvel.

Foram feitos testes de inserção e de recuperação dos dados do banco Cassandra para avaliar a performance perante diferente número de máquinas e fatores de replicação dos dados. A quantidade de nós utilizados foram 1, 2, 4 e 6 no *cluster* montado, e o fator de replicação foi 1, 2 e 3. A configuração das máquinas é: sistema operacional Ubuntu 14.04 LTS 64-bits, processador Intel Core i5-4570 3,2 GHz e 8 GB de memória RAM.

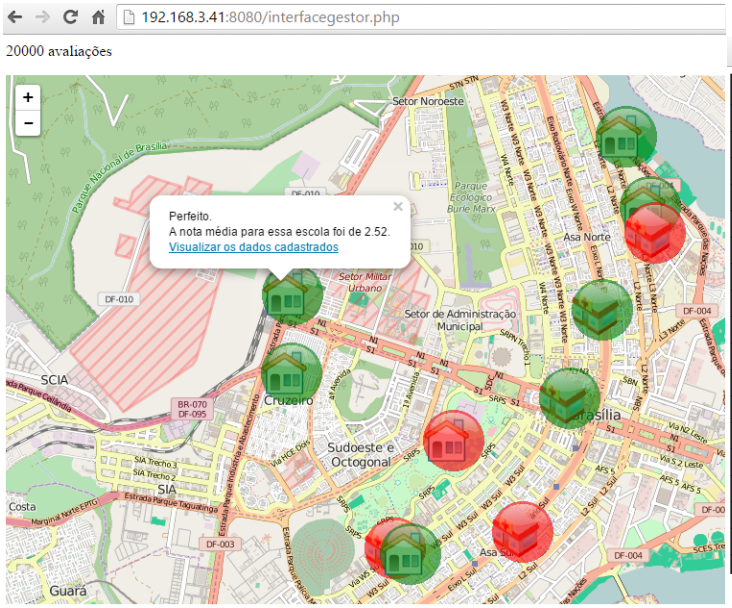
Além disso, foi criado um *script* em PHP, mostrado no Apêndice C.2 para simular a criação de dados pela aplicação "ConsultaOpinião", gerando avaliações aleatórias para todos os estabelecimentos registrados. A quantidade de avaliações geradas foram 10 mil, 100 mil e 500 mil avaliações. O *script* utiliza a extensão do cassandra para PHP e, primeiramente, é obtido todos os estabelecimentos e usuários, e então armazenados em *arrays* para serem utilizados na criação das avaliações. Um *prepared statment* foi criado para utilizar um *batch* na execução de uma quantidade de *queries* de avaliação, assim evitando executar a *query* para uma só avaliação.

A busca pelos dados foi executada em cima das avaliações adicionadas no teste de inserção e também foram com 10 mil, 100 mil e 500 mil registros. A busca foi efetuada com a função do *script* PHP do Apêndice C.3 que pertence a interface do gestor representada na Figura 4.8.

4.5 Resultados

4.5.1 Resultado da Inserção de Dados

As Tabelas 4.1, 4.2 e 4.3 mostram os resultados para a inserção de 10 mil, 100 mil e 500 mil registros, respectivamente. Os testes foram executados 10 vezes para cada quantidade de máquinas em cada fator de replicação. Os resultados apresentados é a média dos valores obtidos em cada teste.



(a) Mapa Virtual.

192.168.3.41:8080/listaravaliacoes.php?id=aa72eeb3-c5

Local	Nome	Média da avaliação (de 0 a 5)
Centro Educacional 01 do Cruzeiro	Matheus	0.6
Centro Educacional 01 do Cruzeiro	Maria	1.6
Centro Educacional 01 do Cruzeiro	Maria	2.8
Centro Educacional 01 do Cruzeiro	Joao	1.2
Centro Educacional 01 do Cruzeiro	Maria	2.4
Centro Educacional 01 do Cruzeiro	Matheus	3.4
Centro Educacional 01 do Cruzeiro	Maria	2.8
Centro Educacional 01 do Cruzeiro	Matheus	2
Centro Educacional 01 do Cruzeiro	Matheus	1.6
Centro Educacional 01 do Cruzeiro	Matheus	1.2
Centro Educacional 01 do Cruzeiro	Matheus	3.8
Centro Educacional 01 do Cruzeiro	Joana	4
Centro Educacional 01 do Cruzeiro	Joao	1.6
Centro Educacional 01 do Cruzeiro	Matheus	2.8
Centro Educacional 01 do Cruzeiro	Joao	2.4
Centro Educacional 01 do Cruzeiro	Joao	1.8
Centro Educacional 01 do Cruzeiro	Matheus	2.4
Centro Educacional 01 do Cruzeiro	Maria	2.2

(b) Tela com Avaliações.

Figura 4.8: Interface do Gestor.

Tabela 4.1: Teste de Inserção: 10 mil Registros.

Cluster \Fator Replicação	FR1	FR2	FR3
1 máquina	0,185969 s	-	-
2 máquinas	0,339603 s	0,258773 s	-
4 máquinas	0,386807 s	0,299202 s	0,289655 s
6 máquinas	0,322666 s	0,304750 s	0,295995 s

Tabela 4.2: Teste de Inserção: 100 mil Registros.

Cluster \Fator Replicação	FR1	FR2	FR3
1 máquina	2,158286 s	-	-
2 máquinas	2,860142 s	2,8384917 s	-
4 máquinas	3,630348 s	3,031991 s	2,951645 s
6 máquinas	3,326106 s	3,058647 s	2,881690 s

Como pode ser observado na Tabela 4.1, com 10 mil avaliações o tempo para inserção dos dados no Cassandra não teve uma variação considerável, isso pode ser explicado pelo fato de a quantidade de dados não ser expressiva para uma análise considerando um grande volume de dados e a latência da rede também pode afetar os resultados.

É possível perceber que com maiores fatores de replicação o tempo de inserção tende a ser menor. Isto se deve a consistência definida: "*ONE*". Esta consistência exige que apenas uma replica seja escrita e as demais replicas sejam sincronizadas, posteriormente.

Tabela 4.3: Teste de Inserção: 500 mil Registros.

Cluster \Fator Replicação	FR1	FR2	FR3
1 máquina	12,58702 s	-	-
2 máquinas	17,710352 s	15,735126 s	-
4 máquinas	19,759899 s	17,504078 s	15,777883 s
6 máquinas	17,411148 s	16,253611 s	15,253688 s

4.5.2 Resultado da Recuperação dos Dados

O mesmo foi feito para a busca destes registro, como mostram as Tabelas 4.4, 4.5 e 4.6.

Tabela 4.4: Teste de Busca: 10 mil Registros.

Cluster \Fator Replicação	FR1	FR2	FR3
1 máquina	0.032132 s	-	-
2 máquinas	0.069502 s	0.104366 s	-
4 máquinas	0.105437 s	0.095996 s	0.073475 s
6 máquinas	0.286052 s	0.104752 s	0.093525 s

Tabela 4.5: Teste de Busca: 100 mil Registros.

Cluster \Fator Replicação	FR1	FR2	FR3
1 máquina	0.271520 s	-	-
2 máquinas	0.485382 s	0.469534 s	-
4 máquinas	0.557769 s	0.512811 s	0.429155 s
6 máquinas	0.904004 s	0.622473 s	0.643169 s

Tabela 4.6: Teste de Busca: 500 mil Registros.

Cluster \Fator Replicação	FR1	FR2	FR3
1 máquina	1.463495 s	-	-
2 máquinas	2.534864 s	2.200382 s	-
4 máquinas	2.554678 s	2.163938 s	1.987452 s
6 máquinas	3.185994 s	2.019001 s	2.561405 s

Os resultados apresentados nas Tabelas 4.4, 4.5 e 4.6 mostram que, em geral, para quanto mais nós no *cluster* mais tempo leva para o banco retornar os registros da consulta. Este resultado pode ser explicado pela latência gerada pela conexão. Predominantemente

o tempo de busca foi menor para fatores de replicação maiores. Com a consistência definida por *ONE*, o banco Cassandra recupera o registro da replica mais próxima como resultado, recuperando, assim, mais rapidamente o registro desejado.

A consistência *ONE* é o padrão do banco Cassandra, entretanto, esse parâmetro pode ser alterado por exemplo para *TWO*, *THREE*, *QUORUM* e *ALL*, gerando diferentes níveis de consistência e, consequentemente, novos resultados [6].

Capítulo 5

Conclusão e Trabalhos Futuros

Banco de dados geográficos são importantes devido ao grande número de áreas que utilizam dados geográficos e o universo de aplicações que podem ser desenvolvidas, como é o caso do Sistema de Informação Geográfico que realiza pesquisas geográficas para desenvolver soluções urbanas e rurais. Outro fato é o crescente volume de dados gerados por usuários por utilizarem a *Internet* como uma plataforma na qual são gerados e armazenados cada vez mais informações. Isto justifica a contínua pesquisa na área de banco de dados geográficos e bancos NoSQL.

O objetivo principal deste trabalho foi o armazenamento de dados geoespaciais no banco de dados NoSQL Cassandra. O banco Cassandra nativamente não possui uma extensão espacial para armazenar e realizar consultas geométricas. Para contornar esse problema foi desenvolvida uma modelagem para armazenar as geometrias com a criação de listas de um *datatype* criado, denominado "POINT". As consultas devem ser combinadas com a aplicação para realizar as consultas espaciais desejadas, e para isso a API Esri/geometry-api-java foi utilizada para realizar o geoprocessamento. Alguns exemplos de consultas espaciais foram realizadas para confirmar a abordagem adotada para este trabalho.

Um estudo de caso mostrou uma aplicação que depende de um banco de dados geográfico para realizar consultas e armazenar estabelecimentos públicos, mostrando, assim, que é possível utilizar o banco Cassandra em aplicações com a abordagem de Sistema de Informação Geográfico com Participação Popular.

Como trabalhos futuros, a quantidade de dados geográficos também pode ser analisada já que pode ser considerada um grande volume de dados quando se leva em conta todo o globo terrestre, ou ainda quando se utiliza diferentes camadas para o mesmo espaço. Por exemplo, o projeto do OpenStreetMap[7] dispõe das camadas: padrão, ciclístico, transporte público, *MapQuest Open* e humanitário, além de o quantidade de detalhes ser crescente e os dados frequentemente atualizados. Outras abordagens que podem ser exploradas é a utilização de dados espaciais com *raster* ao invés de dados espaciais vetoriais, incorporação de outros formatos de aquivos de entrada, como por exemplo, XML e Shapefile, incorporação de dados heterogêneos e o desenvolvimento de outras consultas espaciais com a arquitetura apresentada neste trabalho.

Apêndice A

Scripts CQL

A.1 Script CQL gerado pela ferramenta KDM

```
CREATE KEYSPACE ConsultaOpiniao
WITH replication = {'class':'SimpleStrategy', 'replication_factor':3};
USE ConsultaOpiniao;

// Q1:
CREATE TABLE estabelecimento
(id UUID,
estabelecimento_nome TEXT,
estabelecimento_descricao TEXT,
avaliacao DOUBLE,
latitude DOUBLE,
longitude DOUBLE,
PRIMARY KEY (id));
/* SELECT estabelecimento_nome, latitude, longitude,
    estabelecimento_descricao FROM estabelecimento WHERE id=?; */

// Q2:
CREATE TABLE usuario (cpf TEXT,
usuario_nome TEXT,
senha TEXT,
email TEXT,
PRIMARY KEY (cpf));
/* SELECT usuario_nome, senha, email FROM usuario WHERE cpf=?; */

// Q3:
CREATE TABLE avaliacao
(id UUID,
estabelecimento_nome TEXT,
estabelecimento_descricao TEXT,
avaliacao MAP<TEXT,DOUBLE>,
PRIMARY KEY (id,estabelecimento_nome))
WITH CLUSTERING ORDER BY (estabelecimento_nome ASC);
```

```
/* SELECT estabelecimento_nome, estabelecimento_descricao, usuario_nome FROM
    avaliacao WHERE id=?; */

// Q4:
CREATE TABLE tipoAvaliacao
(id UUID,
tipo INT,
estabelecimento_nome TEXT,
avaliacao LIST<TEXT>,
PRIMARY KEY (id,tipo))
WITH CLUSTERING ORDER BY (tipo ASC);
/* SELECT tipo, estabelecimento_nome FROM tipoAvaliacao WHERE id=?; */
```

A.2 Script CQL das tabelas geométricas

```
CREATE KEYSPACE geometry WITH
replication= {'class':'SimpleStrategy', 'replication_factor':3};

CREATE TABLE IF NOT EXISTS geometry.geom_latlng
(latitude DOUBLE,
longitude DOUBLE,
nome text,
geometria_tipo text,
propriedades map<text,text>,
geometria list< frozen< list< frozen<POINT> >>>,
PRIMARY KEY((longitude,latitude),nome))
with CLUSTERING ORDER BY (nome ASC);

CREATE TABLE IF NOT EXISTS geometry.geom_label
(latitude DOUBLE,
longitude DOUBLE,
nome text,
geometria_tipo text,
propriedades map<text,text>,
geometria list< frozen< list< frozen<POINT> >>>,
PRIMARY KEY(nome, latitude,longitude))
with CLUSTERING ORDER BY (latitude ASC, longitude ASC);


CREATE INDEX if not exists prop_value_1 ON geom_label (propriedades);
CREATE INDEX if not exists porp_key_1 ON geom_label (KEYS(propriedades));
CREATE INDEX if not exists prop_value_2 ON geom_latlng (propriedades);
CREATE INDEX if not exists porp_key_2 ON geom_latlng (KEYS(propriedades));
```

Apêndice B

Arquivo GeoJson

B.1 Descrição dos dados GeoJson testados - Parte 1

```
{"type":"FeatureCollection","features":
   [
   {"type":"Feature","id":"0", "properties":{"name":"Pentágono"},
      "geometry":{"type":"MultiPolygon","coordinates":
   [[[[1.8,3],[3,6],[0,8],[-3,6],[-1.8,3],[1.8,3]]],
   [[[4,6],[5,7],[6,6],[5,5],[4,6]]]]},

   {"type":"Feature","id":"1","properties":{"name":"Quadrado"},
   "geometry":{"type":"Polygon","coordinates":
   [[[3,-3],[3,3],[-3,3],[-3,-3],[3,-3]],[[1,1],[1,0],[-1,0],[-1,1],[1,1]]]},

   {"type":"Feature","id":"2","properties":{"name":"Triangulo"},
   "geometry":{"type":"Polygon","coordinates":
   [[[-2,-1],[-6,5],[-6,-1],[-2,-1]]]},

   {"type":"Feature","id":"3","properties":{"name":"Hexágono"},
   "geometry":{"type":"Polygon","coordinates":
   [[[0,4.5],[1,5],[1,6],[0,6.5],[-1,6],[-1,5],[0,4.5]]]},

   {"type":"Feature","id":"4","properties":{"name":"Linha dupla"},
   "geometry":{"type":"MultiLineString","coordinates":
   [[[-8,-0.5],[-5,-0.5]],[[-8,-1.5],[-5,-1.5]]]},

   {"type":"Feature","id":"5","properties":{"name":"Linha curva"},
   "geometry":{"type":"LineString","coordinates":[[4.5,4],[4.5,-2],[4,-3]]},

   {"type":"Feature","id":"6","properties":{"name":"Linha"},
   "geometry":{"type":"LineString","coordinates":[[0,3],[6,3]]},
   ]
}
```

B.2 Descrição dos dados GeoJson testados - Parte 2

```
{"type":"FeatureCollection","features":
   [
   {"type":"Feature","id":"7","properties":{"name":"Ponto dento triangulo"},
   "geometry":{"type":"Point","coordinates":[-4.5,0.5]}},


   {"type":"Feature","id":"8","properties":{"name":"Pontos centro"},
   "geometry":{"type":"MultiPoint","coordinates":[[0,0],[0,0.5],[0,-0.5]]}},

   {"type":"Feature","id":"9","properties":{"name":"Ponto borda quadrado"},
   "geometry":{"type":"Point","coordinates":[3,0]}},

   {"type":"Feature","id":"10","properties":{"name":"Ponto centro linha"},
   "geometry":{"type":"Point","coordinates":[4.5,0]}},

   {"type":"Feature","id":"11","properties":{"name":"Ponto fora"},
   "geometry":{"type":"Point","coordinates":[6,0]}},

   {"type":"Feature","id":"12","properties":{"name":"Ponto borda linha"},
   "geometry":{"type":"Point","coordinates":[6,3]}}
   ]
}
```

Apêndice C

Scripts PHP

C.1 Script PHP para conectar com banco

```
<?php//conexao.php
   $cluster = Cassandra::cluster()
            ->withContactPoints('127.0.0.1')
            ->withPort(9042)
            ->build();            // connects to localhost by default
   $session = $cluster->connect('consulta_opiniao'); //conecta ao keyspace
?>
```

C.2 Script PHP para gerar avaliações

```
<?php
   ini_set('max_execution_time', 300); //300 seconds = 5 minutes
   $tempoInicial = array_sum(explode(' ', microtime()));

   require('conexao.php');
   $session = $GLOBALS['session'];

   $executar = 'SELECT usuario_nome from usuario';
   $future = $session->executeAsync(new Cassandra\SimpleStatement($executar));
   $result = $future->get();

   $usuarios = array();
   if(!empty($result))
      $num_usuarios = 0;
   foreach($result as $row){
      $usuarios[] = $row['usuario_nome'];
      $num_usuarios++;
   }
```

```
$executar = 'SELECT estabelecimento_id, estabelecimento_nome,
    estabelecimento_descricao from estabelecimento';
$future = $session->executeAsync(new Cassandra\SimpleStatement($executar));
$result = $future->get();

$estabelecimentos = array();
if(!empty($result))
   $num_estabelecimentos = 0;

foreach($result as $row){
   $estabelecimentos[]
       =array($row['estabelecimento_id'],$row['estabelecimento_nome'],
      $row['estabelecimento_descricao']);
   $num_estabelecimentos++;
}

$prepared = $session->prepare(
"INSERT INTO avaliacao (estabelecimento_id, avaliacao_id,
    estabelecimento_nome,estabelecimento_descricao,
    usuario_nome,avaliacao) " .
"VALUES (?, ?, ?, ?, ?,?)");
$options = new Cassandra\ExecutionOptions(array('consistency' =>
    Cassandra::CONSISTENCY_ALL));
$batch = new Cassandra\BatchStatement(Cassandra::BATCH_LOGGED);

for($i=0; $i < 10001; $i++){

   $usuario_r = rand(0,($num_usuarios-1));
   $estabelecimento_r = rand(0,($num_estabelecimentos-1));
   $pergunta1nota=rand(0,5);
   $pergunta2nota=rand(0,5);
   $pergunta3nota=rand(0,5);
   $pergunta4nota=rand(0,5);
   $pergunta5nota=rand(0,5);

   /*Pega informacoes do estabelecimento, sem o _id e localizacao*/
   $resultEstabelecimento = $estabelecimentos[$estabelecimento_r];

   /*Pega informacoes do usuario, sem o _id e a senha*/
   $resultUsuario = $usuarios[$usuario_r];;

   /*Calcula media das notas*/
   $mediaNotas = ($pergunta1nota + $pergunta2nota + $pergunta3nota +
       $pergunta4nota + $pergunta5nota) / 5;

   try{
      if(($i %370==0 && $i!=0) || $i==10000){
         if($i==10000)
            echo "fim ".$i."<br>";
```

```
            $future=$session->execute($batch);
            $batch = new Cassandra\BatchStatement(Cassandra::BATCH_LOGGED);
        }
        $batch->add($prepared, array(
        'estabelecimento_id' => $resultEstabelecimento[0],
        'avaliacao_id' => new Cassandra\Uuid(),
        'estabelecimento_nome' => $resultEstabelecimento[1],
        'estabelecimento_descricao' => $resultEstabelecimento[2],
        'usuario_nome' => $resultUsuario,
        'avaliacao' => (double)$mediaNotas
        ));
      } catch (Cassandra\Exception $e) {
        /*erro ao inserir avaliacao*/
        $session->close();
        echo "<br>". get_class($e) . ": " . $e->getMessage();
        exit(1);
      }
    }
    $session->close();
    $tempoTotal = array_sum(explode(' ', microtime())) - $tempoInicial;
    echo $tempoTotal."<br>";
?>
```

C.3 Script PHP para recuperar avaliações

```
<?php
    ini_set('max_execution_time', 300); //300 seconds = 5 minutes
    $tempoInicial = array_sum(explode(' ', microtime()));


    function retornarPosicao(){
        require('conexao.php');
        $session = $GLOBALS['session'];

        $totalavaliacoes=0;
        $posicao = array();

        if(empty($result)){
            echo 'busca vazia';
        }else{

        $i=0;
        foreach($result as $row){

            $posicao[$i] = $row['latitude'];    //latitude
            $posicao[$i+1] = $row['longitude'];    //longitude
            $posicao[$i+2] = $row['estabelecimento_descricao'];
```

```
            $posicao[$i+4] = (string)$row['estabelecimento_id'];



            $executar = 'SELECT avaliacao from avaliacao where
                estabelecimento_id='.$posicao[$i+4];
            $future = $session->executeAsync(new
                Cassandra\SimpleStatement($executar));
            $result2 = $future->get();

            $soma=0.0;
            $j=0;

            while($result2){
               foreach($result2 as $row){
                  $soma+=$row['avaliacao'];
                  $j++;
               }
               $result2=$result2->nextPage();
            }
            $totalavaliacoes+=$j;

            if($j>0)
               $posicao[$i+3] = number_format((float)$soma/$j, 2, '.', '');
            else
               $posicao[$i+3] =0.0;
            $i++;
            $i++;
            $i++;
            $i++;
            $i++;
         }
         $tempoTotal = array_sum(explode(' ', microtime())) - $tempoInicial;
         echo $totalavaliacoes." avaliações <br>";
         }

         return $posicao;
      }

      $posicao = retornarPosicao();

   ?>
```

Referências

[1] Gilbert. S. & Lynch. N. A. Perspectives on the cap theorem. *Computer*, 45:1, January 2012. 13

[2] Huisman .O & By. R. A. *Principles of Geographic Information Systems.* Number 0 in Textbooks Series. The International Institute for Geo-Information Science and Earth Observation(ITC), 2009. 3, 5, 8

[3] Turner A. *Introduction to Neogeography.* O'Reilly Media, 2006. 8, 9

[4] Greg Brown. Public participation gis (ppgis) for regional and environmental planning: Reflections on a decade of empirical research. *Urban and Regional Information Systems Association*, 25:7–18, August 2012. 9

[5] Strawn G. & Strawn C. Relational databases: Codd, stonebraker, and ellison. *IT Professional*, 18:63 – 65, march 2016. 10

[6] Datastax Corporation. *Apache Cassandra^{TM} 2.2 Documentation.* Open Geospatial Consortium, http://docs.datastax.com/en/cassandra/2.2/pdf/cassandra22.pd, 2015. 21, 22, 23, 24, 52

[7] Chandra G. D. Base analysis of nosql database. *Future Generation Computer Systems*, 52:13–21, may 2015. 10, 11, 12, 14, 15, 22

[8] Maia D. C. M. & Camargos. B. D. Voluntary geographic information systems with document-based nosql databases. *CISTI'2016 - 11ª Conferencia Ibérica de Sistemas y Tecnologías de Información*, June 2016. 44, 45, 47

[9] The Enlightened DBA. *DBA's Guide to NoSQL: Apache Cassandra,.* Datastax. 21, 23, 24

[10] Esri. Esri on github - relational operators (http://esri.github.io/geometry-api-java/doc/relationaloperators.html), abril de 2016. 26, 33, 34, 35, 36

[11] Esri. Esri on github (https://esri.github.io/), novembro de 2015. 26

[12] Chang. F. & et al. Bigtable: A distributed storage system for structured data. *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, pages 205–218, November 2006. 10

[13] Câmara G. & et al. *Introdução à Ciência da Geoinformação.* Instituto Nacional de Pesquisas Espaciais (INPE), 2001. 3, 6

[14] Han J. & et al. Survey on nosql database. *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, 1:363 – 366, October 2011. 13, 16, 17, 18, 19, 21

[15] Khan M. A. & et al. Seven v's of big data understanding big data to extract value. *Proceedings of 2014 Zone 1 Conference of the American Society for Engineering Education (ASEE Zone 1)*, pages 1 – 5, April 2014. 12

[16] Longley P. A. & et al. *Geographic Information Systems and Science*. Number 2. Wiley, 2005. 8

[17] Lourenco J.R.& et al. Nosql in practice: a write-heavy enterprise application. *Big Data (BigData Congress), 2015 IEEE International Congress on*, pages 584 – 591, july 2015. 1

[18] Wang P. & et al. Mapbase – map service extensions embbed in spatial database. *Geoscience and Remote Sensing Symposium, 2004. IGARSS '04. Proceedings. 2004 IEEE International*, 5:2957 – 2959, September 2005. 6

[19] Zhang X. & et al. An implementation approach to store gis spatial data on nosql database. *Geoinformatics (GeoInformatics), 2014 22nd International Conference on*, 1:1–5, June 2014. 1, 8, 11

[20] Goodchild M. F. Geographic information systems and environmental modeling. *Proceedings, Seventh International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, pages 5–10, 1990. 4, 5, 8

[21] Goodchild M. F. Citizens as sensors: the world of volunteered geography. *GeoJournal*, 69:211–221, August 2007. 1, 9, 37

[22] Goodchild M. F. Citizens as voluntary sensors: Spatial data infrastructure in the world of web 2.0. *International Journal of Spatial Data Infrastructures Research*, 2:24–32, August 2007. 8, 9

[23] Raj. P. & Deka. C. G. *Handbook of Research on Cloud Infrastructures for Big Data Analytics, 1º Edição*. IGI Global, 2014. 10

[24] Vaish G. *Getting Started with NoSQL*. Number 11. Packt Publishing, 2013. 10, 16, 17, 18, 19

[25] GeoJson. Geojson (http://geojson.org/), novembro de 2015. 26, 28

[26] Wang G. & Tang J. The nosql principles and basic application of cassandra model. *Computer Science & Service System (CSSS), 2012 International Conference on*, 1:1332 – 1335, August 2012. 13, 15

[27] George. L. *HBase: The Definitive Guide, 1º Edição*. O'Reilly Media, 2011. 10

[28] Yue P. & Jiangu L. Biggis: How big data can shape next-generation gis. *Agro-geoinformatics (Agro-geoinformatics 2014), Third International Conference on*, 1:1–6, August 2014. 1, 10, 11

[29] Hossain S. A. & Moniruzzaman A. B. M. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *International Journal of Database Theory and Applicatio*, 6:1–14, August 2013. 1, 10, 13, 16, 17, 18, 19

[30] Lee J. & Kan M. Geospatial big data: Challenges and opportunities. *Big Data Research*, 2:74–81, june 2015. 1, 8

[31] Sekar G. & Elango N. M. The next generation database language with base properties. *Journal of Theoretical and Applied Information Technology*, 61:363 – 366, march 2014. 10, 14, 15, 19

[32] Clementini E. & Di Felice P. Management of spatial data: basic issues and some solutions. *Applied Computing, 1990., Proceedings of the 1990 Symposium on*, pages 297 – 303, 1990. 1

[33] Lakshman A. & Malik P. Cassandra: a structured storage system on a p2p network. *SPAA '09 Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 47–47, 2009. 19

[34] Lakshman A. & Malik P. Cassandra - a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, April 2010. 19, 20, 22, 23

[35] Chebotko A. & Kashlev A. & Lu S. A big data modeling methodology for apache cassandra. *A Big Data Modeling Methodology*, pages 238 – 245, june 2015. 45

[36] Murugesan. S. Understanding web 2.0. *IT Professional*, 9:34 – 41, August 2007. 1, 10

[37] E. Schlossberg M. & Shuford. Delineating "public" and "participation" in ppgis. *Urban and Regional Information Systems Association*, 16(2), 2005. 9

[38] OGC Standards. *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option.* Open Geospatial Consortium, http://portal.opengeospatial.org/files/?artifact_id=25354, 2010. 7

[39] OGC Standards. *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture.* Open Geospatial Consortium, http://portal.opengeospatial.org/files/?artifact_id=25355, 2011. 7

[40] C. Strauch. *NoSQL Database.* Stuttgart Media University, 2011. 21

[41] Coppock J. T. ; Rhind D. W. History of gis. *Geographical Information Systems: Principles and Applications*, 1:21–43, 1991. 1