UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO VELOSO TOMIO

UTILIZANDO A TECNOLOGIA DE BANCO DE DADOS NOSQL: UM CASO PRÁTICO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2015

GUSTAVO VELOSO TOMIO

UTILIZANDO A TECNOLOGIA DE BANCO DE DADOS NOSQL: UM CASO PRÁTICO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná.

Orientadora: Profª. Drª. Simone de Almeida

PONTA GROSSA

2015

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

UTILIZANDO A TECNOLOGIA DE BANCO DE DADOS NOSQL: UM CASO PRÁTICO

por

GUSTAVO VELOSO TOMIO

Este Trabalho de Conclusão de Curso foi apresentado em 27 de maio de 2015 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Simone de Almeida
Prof.(a) Orientador(a)

_____
Geraldo Ranthum
Membro titular

_____
Tarcizio Alexandre Bini
Membro titular

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Dedico este trabalho aos meus pais e minha namorada, pelo apoio incondicional em todos os momentos.

AGRADECIMENTOS

Em primeiro lugar quero agradecer a Deus por me iluminar nesta caminhada.

Gostaria de agradecer a toda minha família, pelo apoio em todos os sentidos possíveis da palavra, pelo carinho e por acreditarem em mim. Em especial agradeço aos meus pais, Lucilia e Claudinei Tomio, no qual tiveram participação fundamental ao longo desse período.

Agradeço minha namorada Mariana Todorovski Barbosa pelo amor e carinho destinado a mim em todos os momentos, pois ela sempre esteve ao meu lado e sua presença se tornou essencial para a realização deste trabalho. Sou grato também pelo apoio e acolhimento de sua mãe Teodora Todorovski para comigo.

Um agradecimento mais do que justo a minha orientadora Prof.ª Dr.ª Simone de Almeida, onde a mesma dedicou seu tempo e sua sabedoria para me ajudar da melhor forma possível, se tornando também uma inspiração para mim ao longo da graduação.

Aos meus amigos em geral que de alguma forma me influenciaram positivamente para a conclusão dessa dissertação.

Enfim, a coordenação do curso, os professores do departamento de informática e a todos os funcionários da instituição que de alguma forma me ajudaram nessa caminhada.

Não importa quanto a vida possa ser ruim, sempre existe algo que você pode fazer, e triunfar. (HAWKING, Stephen, 2012)

RESUMO

TOMIO, Gustavo Veloso. **Utilizando a tecnologia de banco de dados NoSQL:** Um caso prático. 2015. 119 folhas. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

Este trabalho aborda a tecnologia NoSQL no que diz respeito aos quatro modelos disponíveis, Família de Colunas, Documentos, Grafos e Chave-Valor. Foram definidas quatro ferramentas que os implementam, respectivamente Cassandra, MongoDB, Neo4j e o BerkeleyDB. Foram realizados testes considerando os principais comandos da linguagem SQL para a definição e manipulação de dados no SGBD PostgreSQL, objetivando diferenciar seus comandos dos propostos pela tecnologia NoSQL, como por exemplo, a descrição do procedimento realizado para a criação de uma base em um ambiente relacional convencional em comparação com o executado para a criação de uma base na tecnologia NoSQL, diferenciando a operação para cada ferramenta testada. Os resultados obtidos identificam comandos similares e outros totalmente distintos dos comandos executados em um banco de dados relacional.

Palavras-chave: Tecnologia NoSQL. Modelo de Grafos. Modelo de Família de Colunas. Modelo de Documentos. Modelo de Chave-Valor.

ABSTRACT

TOMIO, Gustavo Veloso. **Using NoSQL database technology:** A pratical study. 2015. 119 papers. Term paper (Bachelor of Computer Science) - Federal Technology University Parana. Ponta Grossa, 2015.

This paper addresses the NoSQL technology with regard to the four models available, Column Family, Documents, Graphs and Key-Value. Four tools were defined that implement respectively Cassandra, MongoDB, Neo4j and BerkeleyDB. Tests were carried out considering the main commands of the SQL language for defining and manipulating data in PostgreSQL DBMS, aiming to differentiate their commands those proposed by NoSQL technology, such as the description of the procedure performed to create a base in a relational environment compared with the conventional run for creating a base in NoSQL technology, differentiating the operation for each tested tool. The results identify similar commands and other totally distinct from commands executed in a relational database.

Keywords: NoSQL Technology. Graph Model. Column Family Model. Document Model. Key-Value Model.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A utilização em grande escala de Banco de Dados Relacionais nas últimas décadas mostra por si só sua importância, onde o mesmo garante ao utilizador a recuperação de falhas, integridade, concorrência, rapidez em consultas, segurança de acesso aos dados, entre outros benefícios (BRITO, 2014). Apesar da constante evolução tecnológica, esse modelo parece ainda dominar o ramo de negócios empresariais e está presente diariamente em diversos níveis de usuário.

O modelo de Banco de Dados Orientado a Objetos objetiva estruturar as informações em objetos, esses só podem ser acessados por métodos específicos, os quais são impostos pela classe na qual o objeto está associado. A criação desse modelo tem sido uma tendência ao se trabalhar com dados mais complexos, por isso é bastante usado nas áreas científicas, espaciais e de telecomunicações (GALANTE *et al*, 2014).

Para unir a estabilidade obtida do modelo relacional e os benefícios das linguagens e dos bancos de dados orientados a objetos, foi criado o Banco de Dados Objeto-Relacional. Nele foram combinados as principais e melhores características dos dois modelos, criando praticamente uma tecnologia nova. Os benefícios desse modelo seguem desde um significativo aumento nas funções do sistema gerenciador, como na consistência, permitindo definições de padrões e o reuso de código (BONFIOLI, 2006).

Uma nova forma de armazenamento de dados é o modelo NoSQL, cujo termo surgiu no final dos anos 90. Entretanto, o conceito que é visto hoje foi reformulado em 2009, trazendo quatro modelos de dados, o Chave-Valor, Documentos, Famílias de colunas e Grafos (FOWLER e SADALAGE, 2013). Ele surgiu em meio a necessidade de se trabalhar com grandes volumes de dados semiestruturados ou desestruturados juntamente com *clusters*. Entre suas principais características estão, o não uso do modelo relacional, seu código na maioria das vezes é aberto, não possuem esquema definido e são usados para propriedades *Web* (FOWLER e SADALAGE, 2013).

O modelo de Chave-Valor pode ser visto como uma simples tabela *Hash*, ou seja, associa chaves de pesquisa a valores. Ele é considerado útil

quando o acesso ao banco de dados se dá por meio da chave primária, a qual pode se fazer uma analogia com uma tabela comum em um gerenciador relacional (FOWLER e SADALAGE, 2013).

O Modelo de Documentos tem uma certa semelhança com o primeiro, pois ele armazena documentos na parte do valor, tem uma estrutura de dados que se assemelha a árvore, vindas de mapas, coleções e valores escalares (FOWLER e SADALAGE, 2013).

O terceiro modelo, Famílias de colunas, permite ao usuário armazenar seus dados em chaves mapeadas para valores, no qual estes são agrupados em diversas famílias de colunas, como um mapa (FOWLER e SADALAGE, 2013).

No banco de dados de Grafos, os nodos são conhecidos como entidades, as arestas do grafo são os relacionamentos e para percorrer a estrutura do grafo é realizada uma consulta nos nodos (FOWLER e SADALAGE, 2013).

1.1 OBJETIVOS

Esta seção apresenta os objetivos do trabalho, sendo que na primeira subseção é apresentado o objetivo geral e na segunda os objetivos específicos.

1.1.1 Objetivo Geral

Aplicar a tecnologia NoSQL em um experimento prático.

1.1.2 Objetivos Específicos

- Distinguir os quatro modelos existentes da tecnologia NoSQL: Grafos, Famílias de Colunas, Documentos e Chave-Valor;
- Selecionar uma ferramenta que atende as especificidades de cada tipo de modelo;

- Realizar testes experimentais de cada ferramenta selecionada;
- Identificar as principais características obtidas das ferramentas experimentadas;
- Diferenciar as características apresentando suas restrições e benefícios.

1.2 JUSTIFICATIVA

No cenário atual, observa-se um aumento significativo na quantidade de dados que os profissionais da área de sistemas gerenciadores de banco de dados devem tratar, e isso só vem aumentando. São milhares de e-mails trocados por dia, inúmeras transações bancárias pelo mundo, incontáveis registros feitos por uma operadora de telefonia, entre outros. Esse grande volume de dados é denominado de *Big Data* (ALECRIM, 2013).

A dificuldade de se utilizar dados desestruturados, é um fator que leva os modelos de Banco de Dados Relacionais a não serem recomendados quando se precisa trabalhar com um volume significativo de dados (ALECRIM, 2013). Por esta razão surge a tecnologia NoSQL, pois é um modelo considerado flexível e que exige menor custo computacional, estando já otimizado para se trabalhar com processamento paralelo, isso para atender uma maior demanda de dados existentes.

Para explorar o funcionamento dos modelos de banco de dados NoSQL, será realizado um experimento prático, que será implementado com cada ferramenta selecionada que represente os respectivos modelos disponíveis pela tecnologia NoSQL. As principais características obtidas dos modelos abordados, suas vantagens e desvantagens, serão apresentados em um quadro comparativo, que apresentará os benefícios do uso de um banco de dados não convencional.

As ferramentas estudadas possuem licença gratuita, pois essa concepção de *software* livre facilita o estudo e concede maior liberdade para o usuário explorar mais detalhes da aplicação. Além disso, a documentação para sua instalação e utilização é mais explorada em artigos e em fóruns, facilitando a realização de testes. Dessa forma, o representante do modelo de Chave-

Valor será o BerkeleyDB (BERKELEYDB,2015), para Documentos o MongoDB (MONGODB, 2015), Famílias de colunas o Cassandra (CASSANDRA, 2015) e para modelos de Grafos o Neo4j (NEO4J, 2015). Além disso, esses sistemas gerenciadores foram escolhidos devido ao fato deles implementarem somente um único modelo NoSQL, diferente de outras ferramentas dessa tecnologia.

Na proposta inicial o banco de dados escolhido para representar o modelo Chave-Valor tinha sido o Riak (BASHO, 2015), porém durante sua instalação surgiram algumas dificuldades que inviabilizaram sua utilização neste trabalho. A primeira é o fato de não existir versão disponível para o ambiente *Windows*, a segunda diz respeito a arquitetura de processador usada pelo mesmo, oferecendo acesso somente ao fabricante AMD quando trabalha-se na plataforma *Linux*. Devido a essas razões decidiu-se substituir o Riak pelo BerkeleyDB.

A aplicação dos modelos em um experimento prático será de fundamental importância para entender a diferença entre os modelos disponíveis e suas aplicabilidades quando se utiliza a tecnologia NoSQL, incentivando novos estudos e o uso do mesmo para distintos projetos, onde a flexibilidade desse banco de dados é um dos principais fatores que podem contar positivamente para essa questão. Ressalta-se ainda que empresas como a Google, o Amazon e o Facebook vêm utilizando de forma crescente o conceito de banco de dados não convencional.

1.3 METODOLOGIA

A pesquisa a ser realizada tem natureza tecnológica, pois parte do conhecimento teórico e prático de modelos de banco de dados convencionais, a fim de obter informações importantes para serem usadas na explicação e no prosseguimento de um estudo mais detalhado sobre o banco de dados não convencional NoSQL. Neste caso a pesquisa bibliográfica é fundamental, sendo necessária para o embasamento da pesquisa.

O trabalho consiste também em uma pesquisa experimental, pois será necessário realizar testes práticos utilizando um sistema gerenciador que implemente os modelos atualmente existentes. A finalidade desse experimento

é utilizar as ferramentas disponíveis para levantar as características específicas das mesmas por meio dos ensaios.

Além disso, uma pesquisa explicativa será realizada sob os resultados obtidos na fase experimental, com o objetivo de identificar os fatores determinantes que nortearam a escolha da ferramenta, explicando os resultados obtidos, apontando onde o modelo NoSQL é mais eficiente e mostrando a importância do mesmo na área da tecnologia da informação.

1.4 ESTRUTURA DO TRABALHO

O trabalho está dividido em quatro capítulos principais e os apêndices, no qual a primeira diz respeito a introdução, apresentando os objetivos do trabalho e justificativa. Além disso, uma seção discute a motivação para o estudo da tecnologia NoSQL, e outra seção descreve as metodologias que serão empregadas, a fim de dar coesão ao que foi proposto.

O segundo capítulo, faz menção ao referencial teórico do trabalho, ou seja, o estudo realizado dos modelos disponíveis da tecnologia NoSQL. O objetivo deste é abordar as principais características dos bancos de dados que foram testados, onde estão divididos em consistência, transações, disponibilidade, escalabilidade e processamento de consultas.

O terceiro capítulo apresenta a parte experimental do projeto, dividida entre os bancos selecionados. Destaca-se a linguagem e a implementação de diversos comandos em seus respectivos sistemas gerenciadores.

O último capítulo encontra-se dividido em duas seções, sendo que a primeira relata a conclusão do trabalho, levando em consideração os resultados obtidos. A segunda seção identifica os trabalhos futuros, propondo alguns temas que podem ser estudados por outros estudantes, professores e pesquisadores interessados nessa área.

2 MODELOS DE BANCO DE DADOS NOSQL

Neste capítulo são abordados os bancos de dados NoSQL escolhidos para experimentar cada um dos modelos atualmente disponíveis nessa tecnologia. O objetivo é entender o funcionamento de cada um dos modelos. O capítulo está dividido em seis seções, sendo a primeira a caracterização da tecnologia NoSQL, as seções dois a cinco apresentam os modelos disponíveis e a última seção aborda os principais pontos de cada um dos sistemas gerenciadores de bancos de dados NoSQL.

2.1 CARACTERÍSTICAS DA TECNOLOGIA NOSQL

A tecnologia NoSQL se iniciou por projetistas de grandes organizações, onde os mesmos desejavam desenvolver um banco de dados que não precisasse de estruturas e regras presentes no modelo tradicional e suportasse uma grande quantidade de dados, ou seja, um sistema gerenciador mais simples, porém robusto. Esse novo método por um lado perdeu as características ACID (Atomicidade, Consistência, Isolamento e Durabilidade), mas por outro ganhou em performance e flexibilidade (BRITO, 2010).

Uma das primeiras implementações utilizando a tecnologia NoSQL foi registrado quando a Google lançou o *BigTable* em 2004, objetivando promover maior escalabilidade e robustez no armazenamento de suas informações. Nos anos posteriores, foram lançados outros gerenciadores de banco de dados, e esses tinham características distintas do *BigTable*, pois foram lançados para suprir necessidades específicas de cada empresa. Assim o NoSQL começou a ser dividido nos modelos de Grafos, Famílias de Colunas, Documentos e Chave-Valor (BRITO,2010).

O NoSQL consiste em um modelo que além de agregar as particularidades ACID, utiliza o BASE (*Basically Avaliable, Soft State, Eventual Consistency*), neste modelo pode haver perca de consistência, porém há um ganho de disponibilidade. O BASE tolera falhas parciais, não comprometendo o sistema todo de uma vez. A escolha do modelo a ser seguido será de acordo com a aplicação, um exemplo são transações bancárias, onde dificilmente será

usado o modelo BASE pelo fato da persistência ser fundamental nesse tipo de transação (BOSCARIOLI e SOARES, 2012).

2.2 BANCO DE DADOS DE GRAFOS

Em meio à dificuldade de realizar algumas pesquisas complexas em banco de dados relacionais, foi que surgiu a ideia de se trabalhar com a teoria dos grafos, pois essas buscas podem ser feitas utilizando a estrutura de um grafo. Apesar de ter um foco diferente de modelos tradicionais, alguns bancos desse modelo de grafos atrelado a tecnologia NoSQL promete ao usuário manter as propriedades ACID, comuns ao modelo relacional (ALMEIDA, 2012).

A fundamentação desse modelo é interpretar os dados do esquema e transformá-los em um grafo dirigido. Os nodos, que são considerados os vértices do grafo e os relacionamentos, estes representados pelas arestas e a propriedade, representando o atributo, são componentes básicos dessa aplicação. De acordo com Ianni (2013), o Neo4j, Infinite Graph (OBJECTIVITY, 2014), InfoGrid (INFOGRID, 2014), HyperGraphDB (HYPERGRAPGDB, 2014), são alguns exemplos de nomes de sistemas gerenciadores conhecidos que tomam por base o modelo orientado a grafos.

Além da melhora nas consultas como apresentado neste tópico, o índice de abstração da aplicação aumenta, onde o usuário terá uma facilidade maior em manipular a base. As principais aplicações que parecem se beneficiar do modelo de grafos são as de redes sociais, redes de informação, redes tecnológicas e redes biológicas (MACEDO, 2011).

2.2.1 Grafos - Neo4j

Este banco de dados, escolhido para representar o modelo de grafos por ser considerado puro, ou seja, não implementa nenhum outro tipo de modelo NoSQL. É um *software* multiplataforma escrito em linguagem de programação Java. Por se preocupar com as transações de forma a garantir as propriedades ACID, esse programa parece ter a solução mais completa para

banco de dados NoSQL, podendo ser comparada aos sistemas gerenciadores que se enquadram no modelo relacional (CUSIN e DE SOUZA, 2014).

A Figura 1 mostra o processo de armazenamento de informações realizada por esse tipo de banco de dados, onde os vértices servirão para o armazenamento do conteúdo, e as arestas indicam a relação entre os vértices.

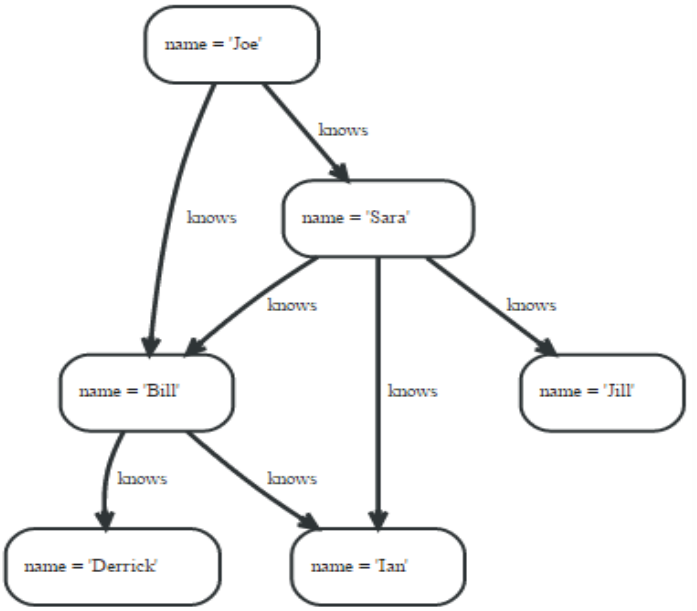

Figura 1 – Armazenamento Neo4j
Fonte: Neo4j (2013)

O Neo4j implementa algoritmos famosos na área da programação para resolver problemas de consulta ao banco, como Dijkstra, *Shortest Path* (Caminho mínimo), *All Simple Paths* (Todos os caminhos simples), *All Paths* (Todos os caminhos) e o A*. Ele é considerado líder em seu segmento, possuindo sua própria linguagem de consulta, denominada *Cypher* (MULLER, 2012).

2.2.1.1 Tratamento de consistência

Este tipo de banco, principalmente no Neo4j, não suporta uma organização de dados em inúmeros servidores, pois com o esquema montado para se trabalhar com um grafo, os nós são interligados uns aos outros, dificultando essa quebra. Apesar disso, como os nodos tem que estar acoplados a somente um servidor, esses dados estarão sempre consistentes.

Assim essa consistência é garantida via transação, existindo sempre na aplicação um nodo inicial e outro final, ou seja, nada pode ficar pendente (FOWLER e SADALAGE, 2013).

Esse banco de dados utiliza o esquema mestre e escravo para a replicação de dados, onde haverá sempre um único mestre para a aplicação. Atualizações para os escravos são eventualmente consistente por natureza, se ocorrer uma falha enquanto o mestre executa alguma transação, essa será revertida e novas operações serão bloqueadas ou falharão até que um novo mestre se torne disponível (NEO4J, 2014).

O Neo4j é tolerante a falhas, podendo continuar operando em várias máquinas ou em apenas uma. Nós escravos serão automaticamente sincronizados com o mestre em operações de escrita, se o mestre falhar um novo mestre será eleito automaticamente. O *cluster* controla automaticamente instâncias que se tornem indisponíveis, aceitando-as novamente como membros do *cluster* quando elas estiverem disponíveis novamente (NEO4J, 2014).

2.2.1.2 Controle de transações

As transações desse gerenciador tratam os dados como o modelo relacional faz, garantindo as propriedades ACID e a atomicidade de suas transações (ALMEIDA, 2012).

Qualquer consulta que atualize o grafo será executada em uma transação, o que garante que se nenhum problema ocorrer durante sua execução, todas as atualizações do grafo serão confirmadas, ou nada será feito, nunca havendo a opção de realizar somente uma parte da transação. A linguagem *Cypher* irá criar uma nova transação, ou executar dentro de uma já existente (NEO4J, 2014).

2.2.1.3 Disponibilidade

A partir da versão 1.8 ocorreram mudanças dentro do gerenciador que possibilitaram maior disponibilidade. A mesma foi desenvolvida para ser

utilizada no modo mestre e escravo. No modo escravo as réplicas são configuradas com o serviço **EmbeddedGraphDatabase**, e para fornecer uma disponibilidade maior, a configuração deve ser mudada para o modo **HighlyAvailableGraphDatabase** (FRIESS, 2013).

Para manter códigos da última transação que foi persistida no mestre atual e em seus escravos, o Neo4j utiliza um serviço chamado *Apache ZooKeeper*, no qual serve para coordenar as distribuições. Quando o servidor começa a funcionar, este se conecta com o serviço *Apache ZooKeeper*, para encontrar o nó mestre, caso não exista, este nodo será o principal. Como já visto anteriormente, em razão de alguma falha, o *cluster* escolherá um novo mestre de acordo com os nodos *online*, mantendo assim uma alta disponibilidade (FOWLER e SADALAGE, 2013).

2.2.1.4 Escalabilidade

Uma técnica para se conseguir que um banco de dados seja escalável, seria usando fragmentação, onde a carga de registros seria dividida em inúmeros servidores. Mas se tratando do gerenciador estudado nesse modelo, a prática da fragmentação se torna complicada, pois o mesmo é orientado a relacionamentos, ou seja, pode haver uma quebra das relações caso ocorra uma fragmentação, o que irá reduzir seu desempenho (FOWLER e SADALAGE, 2013).

Com o constante aumento do volume de dados, o modelo de grafos disponibiliza três técnicas para obter melhor resultado em sua utilização. A primeira é adicionar memória RAM (*Randon Access Memory*) de tal maneira que a aplicação esteja totalmente dentro da memória. Pode-se também colocar todas as operações de leitura nos escravos e a escrita somente no mestre, melhorando assim as consultas, essa técnica é válida quando a memória existente não é suficiente para comportar o banco de dados integralmente (FOWLER e SADALAGE, 2013).

A última forma de manter a escalabilidade na base de dados poderá parecer contraditória de acordo com os conceitos apresentados anteriormente a respeito da fragmentação. No caso do Neo4j essa técnica pode ser usada a

nível de aplicativo, quando a replicação se torna inviável. Para a fragmentação ocorrer, pode ser feita uma divisão por regiões na aplicação, onde os nodos de uma região ficam armazenados em servidores distintos como ilustrado na Figura 2. A aplicação precisa ter conhecimento dessa divisão física (FOWLER e SADALAGE, 2013).

A Figura 2 representa um exemplo de como pode-se realizar uma fragmentação a nível de aplicativo. A aplicação está dividida em duas regiões geográficas, assim a América do Norte compõe os nós que serão usados em sua região, ou seja, nenhum serviço da Ásia terá interesse nos nodos da outra região, e vice versa, garantido assim alto desempenho e um banco escálavel em ambas as partes.

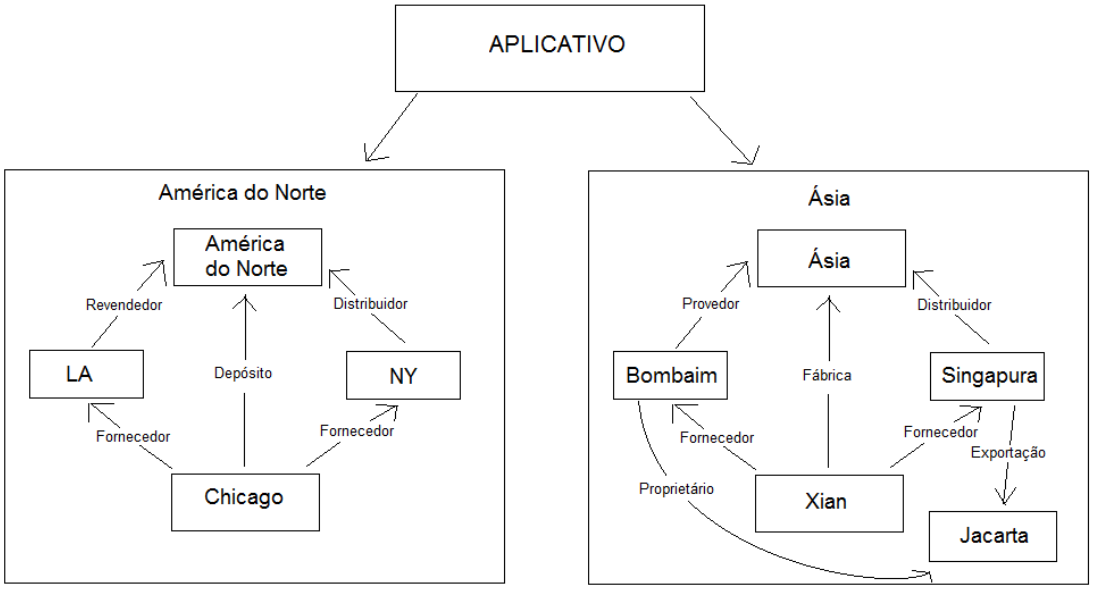

Figura 2 – Fragmentação de nodos em nível de aplicativo
Fonte: Fowler e Sadalage (2013)

2.2.1.5 Processamento de consultas

Os desenvolvedores do Neo4j estão apostando na migração do *software* para a nuvem (*Clouding Computing*), pois este conceito está sendo bastante usado atualmente na área da informática. A liguagem do Neo4j é denominada *Cypher*, a qual possui uma interface via *Web*, cuja mesma foi melhorada a fim de aumentar o desempenho de uma pesquisa no banco (LINUX MAGAZINE, 2012).

Como há várias ferramentas que possibilitam a consulta nesse banco de dados, o foco será no *Cypher*, pois a mesma é nativa do *software*. Poderá ser observada a diferença do modelo de grafos para os demais, pois entre os quatro modelos existentes, este parece ser o mais peculiar em sua forma de buscar as informações, em função de sua estrutura ser baseada em vértices e arestas.

O *Cypher* possui suas cláusulas, como qualquer outro tipo de linguagem de consulta e também outros tipos de palavras-chave, onde suas principais são:

START : mostra o nó inicial.

MATCH : apresenta o relacionamento.

WHERE : cláusula condicional**.**

RETURN : o que a pesquisa irá retornar.

ORDER BY : propriedades de ordenação.

SKIP : nós que serão ignorados.

LIMIT : cláusula que limita os resultados.

O exemplo a seguir foi adaptado de (FOWLER e SADALAGE, 2013). O mesmo ilustra o uso de algumas dessas cláusulas utilizando a estrutura de grafos, para encontrar todos os nós que estão conectados a ‘Todorovski’, sendo de entrada ou saída. O comando para resolver a consulta seria:

START Todorovski = node.nodeIndex (name = “Todorovski”)

MATCH (Todorovski) - - (connected_node)

RETURN connected_node

2.3 BANCO DE DADOS COM ARMAZENAMENTO EM FAMÍLIAS DE COLUNAS

O modelo em questão se contrapõe aos modelos mais tradicionais, como o relacional, onde os dados são mantidos em estruturas denominadas de tabelas. No armazenamento em famílias de colunas os dados ficarão em uma tripla, esta corresponde a linha, coluna e *timestamp*, no qual o último tem a função de distinguir diversas versões de um mesmo tipo de dado (ALMEIDA e BRITO, 2012).

Em meio a dificuldade de trabalhar com um grande volume de dados, esse modelo foi desenvolvido. Há vários tipos de gerenciadores que trabalham desse modo, como o Cassandra, HBase, Hypertable, AmazonSimpleDB, BigTable, entre outros.

Outra diferença para o modelo relacional é que cada uma das linhas terá colunas próprias. A exclusão ou inclusão de colunas em uma linha é aceita em qualquer momento, sendo que a linha irá possuir uma chave (DA SILVA e ETSCHEID, 2014).

2.3.1 Famílias de Colunas - Cassandra

As versões do banco de dados Cassandra são desenvolvidas em linguagem Java, sendo suportado em diversas plataformas, a fim de atender uma demanda maior de usuários. Seus principais quesitos que o fazem ser conhecido na área de tecnologia, podem ser resumidos em escalabilidade, confiabilidade e fácil gerenciamento dos dados (FARIA, 2014).

O Cassandra armazena suas informações em um esquema descrito como famílias de colunas, o qual se tem uma chave para cada uma dessas colunas para identificá-las dentro da aplicação. No corpo da coluna é guardado as informações reais, e há um campo para designar a data e a hora deste armazenamento, como pode ser observado um exemplo na Figura 3.

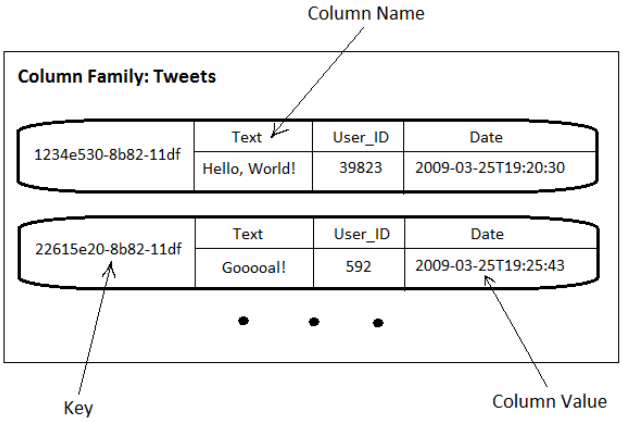

Figura 3 – Armazenamento Cassandra
Fonte: Otavio Gonçalves de Santana (2014)

Apesar dessas qualidades do *software*, ele é visto também como tolerante a falhas, descentralizado e de alto desempenho e disponibilidade. Ele não possui uma separação de mestre e escravos como em outros gerenciadores de banco de dados. Para esse gerenciador os servidores possuem igual importância. Grandes empresas reconhecidas mundialmente utilizam esse banco, onde pode-se citar o Facebook, Twitter, Netflix, Ebay e até mesmo a NASA (*National Aeronautics Space Administration*), dentre outras (ANICETO e XAVIER, 2014).

2.3.1.1 Tratamento de consistência

O Cassandra possui duas etapas para assegurar que uma gravação seja feita de forma segura. Inicialmente, os dados são gravados em um registro de operações (*commit log*), logo após há uma parte da memória denominada *memtable*, onde os dados são copiados, finalizando a gravação. Esse gerenciador possui várias formas de tratar a consistência de escrita e leitura, garantindo que os dados se mantenham corretos nas distintas réplicas (FOWLER e SADALAGE, 2013).

Na escrita existem níveis que podem ser configurados de acordo com a necessidade do cliente, onde será retratado alguns dos principais níveis. O comando *Any* se preocupa em gravar utilizando pelo menos um nodo. O nível *One* tem o objetivo de garantir a réplica em um único nó. *Quorum* é um nível de alta consistência, pois define que as réplicas serão feitas na metade da quantidade total de nós mais um. E o mais consistente e menos disponível é o nível *All*, o qual a escrita será feita em todos os nodos, porém se houver um erro em pelo menos um desses, a operação inteira falhará (ANICETO e XAVIER, 2014).

As operações de leitura no banco possuem níveis de consistência, assim como os processos de escrita. O nível *One* como na escrita é o que tem maior disponibilidade, retornando a resposta pela réplica mais próxima. Utilizando o *Quorum* para leitura, tem-se uma maior consistência, podendo ser tolerável a falhas. Por último o nível *All*, onde o mesmo possui um problema de disponibilidade, pois será pesquisado um retorno em todos os nós réplicas,

aumentando consideravelmente a chance de ocorrer um erro (ANICETO e XAVIER, 2014).

Uma técnica bastante interessante chamada de *hinted handoff*, é vista nesse banco de dados. A mesma serve para quando um determinado nó escolhido para um processo de armazenamento esteja indisponível, a solução é passar os dados para outros nodos. Esse método completa sua ação quando o nodo voltar a estar disponível, retornando ao mesmo as últimas atualizações, para assim resolver de forma eficiente possíveis problemas (FOWLER e SADALAGE, 2013).

2.3.1.2 Controle de transações

Relacionado a outros bancos e modelos da tecnologia NoSQL, o Cassandra trabalha com transações diferente dos bancos do modelo relacional, pois é um programa com foco na alta disponibilidade. O aplicativo suporta ao mesmo tempo o envio de mais de um dado para uma determinada família de coluna, e o mesmo usa o *timestamp* do campo para saber qual é a versão mais atual de um registro (SANTANA, 2014).

O *software* não possui a opção do usuário tomar a decisão de confirmar ou não as operações de gravação iniciadas, ele trabalha com atomicidade, onde poderá haver uma falha ou um retorno de êxito. Caso seja inserido diversos dados em colunas, os mesmos poderão ser tratados como uma única gravação, se forem referenciados por uma única linha com o valor da sua chave (FOWLER e SADALAGE, 2013).

2.3.1.3 Disponibilidade

A disponibilidade é o principal objetivo desse banco de dados. Essa disponibilidade irá variar de acordo com as opções do usuário nas configurações existentes para a replicação dos dados, onde o mesmo irá decidir dentro da aplicação se é a escrita ou a leitura que terá maior disponibilidade.

O fato do banco garantir uma boa disponibilidade de seus dados, está relacionado a como eles são replicados, sendo que os dados podem estar espalhados em diversos *clusters*. Uma das facilidades, vistas para atender o usuário, que unida ao modelo NoSQL surgiu para melhorar essa relação de disponibilidade, é a questão dos serviços de *cloud computing*, onde este serviço trabalha paralelamente com esse tipo de banco (PASQUALINI, 2014).

2.2.1.4 Escalabilidade

Um dos principais atrativos de empresas e estudiosos da área de tecnologia a migrarem para o uso de uma base de dados que utilize o modelo não relacional é a perspectiva de poderem manipular um grande volume desestruturados de dados. Atualmente pode-se observar que os dados vêm crescendo durante os últimos anos, dando ênfase ao conceito de *Big Data*.

O *software* analisado não trabalha com o conceito de mestre, porém é possível adicionar diversos nós aos *clusters* para assim aumentar a capacidade de armazenamento do banco. O mesmo utiliza escalabilidade horizontal, o qual quando um nó da aplicação está sobrecarregado, o mesmo divide seus registros com outros nodos livres de processamento, com o intuito de fazer o sistema continuar trabalhando de forma eficiente (PEREIRA, 2014).

2.3.1.5 Processamento de consultas

A otimização das colunas e famílias de colunas será de grande valia quando o processo de leitura executado, pois a linguagem de consulta do Cassandra não é considerada uma ferramenta poderosa. Ele utiliza uma linguagem própria para esse fim, o CQL (Cassandra *Query Language*), com alguma semelhança com a tradicional SQL (*Structured Query Language*), porém com menos recursos (FOWLER e SADALAGE, 2013).

Para seleções básicas, pode-se perceber quão parecido as duas linguagens são, como citado no parágrafo anterior, onde para selecionar todas as colunas do cliente, o comando é o seguinte: **SELECT * FROM cliente**. Ou para se pesquisar apenas colunas específicas dos clientes cadastrados, as

quais podem ser o nome e o sexo, o código pode ser construído dessa maneira: **SELECT nome, sexo FROM cliente**.

As cláusulas *where* são as principais questões para se distinguir as linguagens de consulta dos modelos relacionais e do Cassandra, sendo nesse ponto que faltam maneiras mais complexas de se buscar um registro na base. Um exemplo utilizando o comando *where*, no qual a cláusula irá buscar todos os clientes que moram na cidade de São Paulo, mostrando as colunas do nome e estado, e para isso o comando CQL usado pode ser como segue: **SELECT nome, uf FROM cliente WHERE cidade = 'São Paulo'**.

Os comandos de inserção, atualização e exclusão, também são semelhantes ao tradicional SQL, como pode ser observado nos próximos exemplos. Assim para inserir uma informação, o código é o seguinte: **INSERT INTO cliente (nome, cidade, uf) VALUES ('Veloso', 'Ponta Grossa', 'PR')**. Um código para uma alteração seria: **UPDATE cliente SET cidade = 'Curitiba' WHERE nome = 'Veloso'**. E utilizando o comando *delete*, segue a seguinte sintaxe: **DELETE from cliente WHERE nome = 'Veloso'**.

No comando *insert*, é inserido o nome, cidade e o uf na família de colunas cliente. Os valores escritos na sequência devem estar na mesma ordem das colunas. Na atualização é selecionada a família cliente e alterado o nome da cidade para o cliente de nome Veloso. E finalmente para o código de exclusão, escolhe-se a família, novamente a cliente e na cláusula *where* é colocado o critério de exclusão, onde nesse caso é o nome do cliente.

Uma limitação em termos de espaço, são as restrições no tamanho das colunas, onde o valor de cada uma não pode ultrapassar os 2GB (*Gigabyte*). Outro fator importante e não muito trivial ao usuário, é que o mesmo deverá configurar os índices, pois o Cassandra não os cria automaticamente, ficando a critério das pessoas envolvidas no projeto solucionar esse problema da melhor forma (MACÊDO *et al*., 2011).

2.4 BANCO DE DADOS DE DOCUMENTOS

Este modelo de banco de dados difere das formas tradicionais, em função do armazenamento de dados a ser realizado em uma estrutura

denominada documento. Sendo que no modelo relacional é comum esses dados estarem em estruturas representadas por relações, considerada eficiente. Diferentemente, um documento pode ser manuseado de forma mais maleável, pois não possui um esquema (LENNON, 2011).

Por não possuírem esquema definido e trabalharem com dados não estruturados, esses bancos de dados resolvem problemas que o modelo relacional tem dificuldade de trabalhar. Para o seu armazenamento, utilizam-se documentos do tipo XML (*eXtensible Markup Language*), JSON (*JavaScript Object Notation*), ou outro semelhante, assim essas estruturas suportam arquivos dentro de arquivos (DIANA e GEROSA, 2010).

Embutir documentos pode gerar uma duplicação de dados, o que pode ser um problema dentro da estrutura. Existem vários gerenciadores correspondentes a esse modelo de tecnologia NoSQL. Neste trabalho será utilizado o MongoDB.

2.4.1 Documentos - Mongodb

O MongoDB é um *software* livre, escrito em uma linguagem de programação C++, podendo ser utilizado em vários tipos de cenários, sendo que sua linguagem de consulta é considerada como um diferencial, por ser baseada em documentos (LENNON, 2011). Segundo o autor, é um programa fácil de instalar e usar. Outra vantagem é que o mesmo possui suporte para diversas plataformas.

A Figura 4 tem por objetivo ilustrar de forma simples como o MongoDB organiza suas informações, ou seja, o esquema feito pelo banco de dados para armazenar as requisições do usuário. Assim, os dados são armazenados dentro de coleções, e dentro de uma coleção a disposição das informações parece semelhante a uma tabela do modelo relacional.

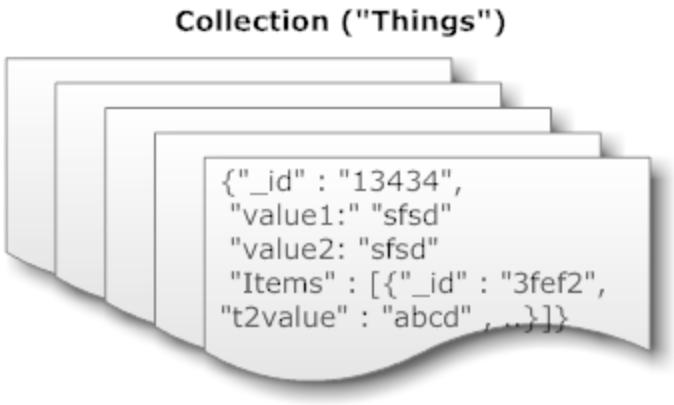

Figura 4 – Armazenamento MongoDB
Fonte: Paulo Fagundes (2014)

Para se compreender esse banco de dados, pode-se descrever uma sequência de fatos, onde a base de dados tem por objetivo guardar diversas coleções, estas por sua vez se encarregam de armazenar os documentos, sendo estes um agregado de campos, um destes campos é considerado como um par chave-valor. A chave seria simplesmente um caractere ou cadeia dos mesmos, e o valor pode ser relacionada ao tipo da chave, como ponto flutuante, um vetor, entre outros (LÓSCIO *et al*., 2011).

2.4.1.1 Tratamento de consistência

Para entender como o MongoDB realiza o tratamento de consistência, é necessário saber como a replicação do banco está configurada. Este gerenciador trabalha com o modelo *master* e *slave* (mestre e escravo), onde o servidor chamado de mestre é aquele no qual pode-se executar comandos de criação e alteração. Já os servidores conhecidos como escravos estão prontos para obter os dados do servidor principal, para deixar os mesmos sincronizados.

A replicação de dados para todos os nós faz com que caso exista algum tipo de falha no mestre, algum dos servidores considerados escravos poderão assumir o controle da aplicação. Para saber qual nodo assumirá o lugar do principal, dependerá do esquema montado, pois os escravos são configurados a fim de determinar uma prioridade dentro da aplicação caso ocorra erros nos servidores (FAGUNDES, 2014).

Sua configuração padrão é utilizada com o objetivo de obter maior desempenho em seu processo de persistência, porém pode parecer inseguro, devido as informações poderem ser perdidas devido ao fato dos dados não serem armazenados de maneira instantânea quando executado um processo de escrita. Caso haja necessidade de uma maior durabilidade, é recomendado especificar outra configuração disponibilizada pelo *software*, a de gravação segura, onde a mesma se certificará que os dados sejam passados a um dispositivo não volátil antes de confirmar o armazenamento dos dados (FAGUNDES, 2014).

Em termos de configuração, é importante conhecer as necessidades do aplicativo existente ou que será criado, os tipos de dados que vão ser tratados e a importância deles para o negócio em questão. Com essas questões esclarecidas, pode-se colocar em prática uma configuração que seja tolerando e flexível a aplicação, principalmente em termos de segurança.

2.4.1.2 Controle de transações

Comparado com sistemas gerenciadores de banco de dados relacionais, onde se utiliza comandos de atualização da base como *insert*, *update* e *delete* para se gerar uma ou mais transações, a tecnologia NoSQL se faz semelhante no caso do MongoDB. A diferença mais crucial desses modelos é o fato de os sistemas gerenciadores relacionais utilizarem os comandos *commit* e *rollback*, usados para confirmar ou desfazer uma ação realizada pelo usuário, respectivamente, o que não ocorre no NoSQL (FOWLER e SADALAGE, 2013).

O MongoDB não possui transações e por padrão os processos são retornados como bem-sucedidos. Existem duas alternativas para esse empecilho, o primeiro seria realizar as operações sendo atômicas, ou seja, de forma binária, a outra caso a primeira não tenha sido suficiente, pode ser utilizado método da gravação em duas fases (PIGA, 2013).

2.4.1.3 Disponibilidade

Em detrimento do modo que o MongoDB faz suas replicações, onde vários servidores irão possuir os dados vindo de um servidor mestre, garantindo a disponibilidade da base em inúmeros nodos. Isso torna o banco com uma alta disponibilidade, pois tem várias alternativas caso a máquina principal apresente algum tipo de problema, sendo passado a outro nó a função de mestre (FOWLER e SADALAGE, 2013).

Apesar de na teoria parecer bom o método que o banco utiliza para replicar os dados, mantendo uma alta disponibilidade, há limites para essa replicação, suportando apenas doze nodos. Para tentar ultrapassar essa quantidade de nós, uma alternativa seria utilização da configuração mestre e escravo, portanto, a mesma parece não ser recomendada pelos desenvolvedores do *software*, onde caso o servidor principal "caia" será necessário eleger um novo mestre manualmente, enquanto isso as instâncias do banco devem ser paradas (WEISSMANN, 2013).

2.4.1.4 Escalabilidade

Um dos principais benefícios para convencer o usuário do uso do MongoDB que podem ser vistos em artigos, revistas e textos em geral, é o fato de pessoas que trabalham com banco dados afirmarem que o mesmo possui uma alta escalabilidade. Para que essa premissa seja considerada verdadeira o gerenciador necessita de duas práticas, a replicação, a qual foi vista anteriormente e também de um novo conceito, denominado *sharding* (LENNON, 2011).

O método *sharding* não possui uma tradução ao certo para o português, seria como se fosse "quebrar os dados em pequenos fragmentos". Esse banco escala seus elementos horizontalmente com a utilização de um *sharding* automático, o qual é executado sempre que necessário sem a intervenção de um usuário. Ele irá funcionar para um balanceamento de carga, suportando diversos nodos (LENNON, 2011).

Essa quebra é importante para atender um possível crescimento da quantidade de dados, o que pode acarretar falta de espaço de armazenamento. Com o *sharding* automático os dados podem ser divididos em diversos *clusters*, para deixar os nodos balanceados, não sobrecarregando nenhum. Pode ser configurada essa fragmentação de acordo com o local geográfico do cliente, deixando dados pertinentes a essa pessoa em equipamentos próximos a ele, facilitando o acesso aos dados (FAGUNDES, 2014).

2.4.1.5 Processamento de consultas

O modelo de documentos apresenta várias soluções para consulta na base de dados, entretanto, para o MongoDB, o qual tem uma linguagem para a realização de consultas baseada no JSON, possuindo uma estrutura de extração de dados que podem ser combinadas com algumas cláusulas disponíveis nessa linguagem, podendo dessa forma construir uma consulta (FOWLER e SADALAGE, 2013).

Um exemplo simples de consulta no MongoDB seria a utilização do comando: **db.pedido.find()**. O pedido seria o nome de um documento e *find* o comando de consulta, sendo que o resultado produzido pelo comando será todos os pedidos existentes. Caso o usuário necessite restringir a consulta para um único cliente, o comando poderá ser executado da seguinte forma: **db.pedido.find({customerCod: “1010”})**.

Outro tipo de restrição que poderá ser aplicado em uma consulta seria a seleção de determinados campos diferenciando do exemplo anterior, onde todos os campos são selecionados. Um exemplo desse tipo de restrição onde poderá haver interesse apenas na consulta do código e data do pedido, seria: **db.pedido.find({customerCod: “1010”}, {orderCod: 1, orderDate: 1})**. O número 1 indica que o retorno dos documentos será de forma crescente, caso contrário, coloca-se -1.

Um dos modos de inserção de dados nesse programa é usando o método *save*, como a seguir: **db.cliente.save({posição: 1, nome: “Gustavo”})**. Assim o nome Gustavo será inserido na coleção cliente, na posição 1. Para alterar uma informação existente, o comando será:

db.cliente.update({id: 2}, {$set: {nome: “mariana”}}). O banco irá atualizar o nome onde o id seja 2, e o operador *set* serve para dizer que somente o nome dessa coleção será alterado. Também podem ser criados objetos para essas solicitações.

Para exclusão será usado um exemplo com a utilização do objeto, onde é necessário fazer uma buscar com o comando *findOne*, retornando essa seleção para o objeto criado, feito isso utilizará o método *remove* para excluir a informação solicitada. O código para tal feito pode ser feito da seguinte maneira:

```
a = db.cliente.findOne({nome: “marcos”});
dp.cliente.remove(a);
```

Normalmente os documentos desse banco suportam um tamanho máximo de 16MB. Caso seja necessário trabalhar com um volume maior que este ou para receber algum retorno de consulta, existe uma coleção do tipo GridFS, onde o mesmo irá dividir o documento em várias partes, esses fragmentos tem o nome de *chunks*, e eles são gravados em arquivos diferentes, para assim guardar de forma adequada o tamanho que excede o documento original (BALLEM, 2014).

2.5 BANCO DE DADOS DE CHAVE-VALOR

De acordo com os modelos tratados neste trabalho, o de Chave-Valor é considerado o mais simples deles, permitindo sua visualização em uma tabela *hash*, ou seja, uma estrutura de dados na qual se associa chave de pesquisas a valores. A estrutura funciona da seguinte maneira, as chaves só podem estar relacionadas a um único valor, não importando o tipo do mesmo, onde o uso dessa técnica é conhecido por sua eficiência (FANTINATO *et al*., 2014).

Segundo a referência, esse modelo torna os dados cada vez mais escaláveis, comparado com os outros modelos da tecnologia NoSQL. Aliado a escalabilidade, o modelo Chave-Valor possui um bom desempenho em suas tarefas. Pode-se citar alguns gerenciadores como exemplos, o Risk, SimpleDB, Redis e o BerkeleyDB, onde o último será usado como base de estudo neste projeto (DIEGUES *et al*., 2013).

Em relação as vantagens, o banco de dados desse modelo tem a capacidade de gerenciar o tamanho da base. Seguindo essa linha, o sistema gerenciador é considerado rápido e tem o poder de processar uma grande quantidade de operações de escrita e leitura. Caso a operação a ser realizada com maior frequência for de consulta, esse modelo não é o mais indicado, sendo esta uma desvantagem dentro de uma aplicação (JUNIOR, 2014).

2.5.1 Chave-Valor - Berkeleydb

O banco de dados NoSQL BerkeleyDB é escrito em linguagem de programação estruturada C, podendo ser usado em diversos tipos de sistemas operacionais e arquiteturas. O programa possui API (*Application Programming Interface*) para diversas outras linguagens de programação, como Java, PHP, Python, C++, C#, entre outras. Esse sistema gerenciador tem a capacidade sustentar diversas *threads* relacionadas aos processos de manipulação dos dados (ORACLE, 2014).

O BerkeleyDB parece ter uma estrutura de armazenamento mais simples que os demais bancos analisados nessa pesquisa, por possuir somente uma chave e a outra parte se tratar da informação que será armazenada. Na Figura 5 pode-se observar um exemplo ilustrado de como esse banco guarda seus dados.

Order Primary Database	
Key	Data
9876543	19283746;3/17/08;138.50;3/18/09
	0003;2;130.00
	0019;1;8.50
8769887	19283877;3/16/08;1024.00;3/16/0810
	0092;10;1024.00
5430987	90867654;3/16/08;564.98;;5
	0003;3;195.00
	0092;1;102.40
	9902;1;267.58

Figura 5 – Armazenamento BerkeleyDB
Fonte: Gregory Burd (2011)

Esse banco de dados Chave-Valor possui código aberto. Permite criar índices para tabelas e outras estruturas de dados. Por exemplo, um registro poderia armazenar uma chave e seu valor, sendo que o valor pode possuir qualquer estrutura de dados, incluindo texto, uma imagem, um áudio ou vídeo de até 4GB de tamanho. Ao contrário de bancos de dados relacionais, BerkeleyDB não suporta consultas SQL. Todas as consultas e análises de dados deve ser realizada pela aplicação através da interface de programação (PCMAG, 2014).

2.5.1.1 Tratamento de consistência

A consistência deve ser mantida em várias fases dentro do banco, como na replicação, onde o BerkeleyDB suporta replicações em um único mestre e também híbrida entre o mestre e o escravo, podendo haver um mestre e vários escravos dentro da aplicação. O servidor principal irá fazer leituras, atualizações e exclusões, enquanto os escravos podem realizar operações de leitura ou ficar em espera. Nesse banco a replicação é feita a nível físico o que significa que as mudanças que acontecem no armazenamento são devidas as modificações no banco de dados, as quais são enviados através das réplicas (YADAVA, 2007).

Esse *software* é dividido em subsistemas, em método de acesso, *memory pool*, transação, *locking* e *logging*. O primeiro auxilia na criação e acesso a arquivos dentro da base de dados, sendo um dos mais importantes para a manutenção da consistência. O *memory pool* faz com que vários processos consigam utilizar a mesma região de memória. A transação garante atomicidade aos processos (BIANCO, 2008).

O subsistema *locking* está encarregado gerenciar regiões críticas do banco utilizando uma espécie de bloqueio caso algo esteja errado, garantindo a consistência e a performance das transações. Por último, o *logging* é mantido para dar suporte as transações, pois antes dos processos serem executados na base, eles são gravados em um *log*, onde se houver uma falha no sistema, o *log* será usado para reestabelecer o processo não concluído, ou também para cancelar essas mudanças não executadas (BIANCO, 2008).

2.5.1.2 Controle de transações

As transações do BerkeleyDB procuram garantir as propriedades ACID, como nos bancos relacionais. Seu suporte na área da transação é considerado o recurso mais poderoso dessa aplicação, pois o isolamento concedido pelo sistema não deixa transparecer falhas, onde caso aconteçam, a operação que está sendo realizada é desfeita. Outra operação útil nesse tipo de programa, é chamada de *hot backup*, ou seja, esse método faz uma cópia de segurança de toda a aplicação sem ter a necessidade de parar o banco de dados (BOSTIC *et al*., 2000).

Outro quesito tratado brevemente no tópico anterior mas que vale a pena ser ressaltado, o qual fala que as transações desse programa buscam realizar essas operações de forma atômica, onde as instâncias criadas serão escritas ou lidas em sua totalidade, não havendo a possibilidade de executar uma transação por partes (YADAVA, 2007).

2.5.1.3 Disponibilidade

A replicação usada pelo BerkeleyDB, chamada de único mestre, é importante para garantir a alta disponibilidade no banco. Esse *software* tem a capacidade de gerenciar os registros da base na memória, no disco, ou pode usar as duas formas ao mesmo tempo. A parte da administração do programa, como tempo de execução, é manuseado pelo próprio aplicativo, onde o programador não tem acesso a essas questões (ORACLE, 2011).

Esse banco foi desenvolvido para ser rápido, pequeno e confiável, no qual permite replicações em larga escala, e possui dispositivo tolerante a falhas para ter a garantia que o banco estará disponível. Assim, as réplicas têm um papel importante para a garantia do sistema estar sempre *online*, onde essa técnica consegue manter a disponibilidade caso haja uma falha de replicação, outro nó pode assumir. Se o mestre tiver algum problema, será escolhido outro servidor para sua função, e este será sincronizado com as replicações, não interrompendo o serviço (YADAVA, 2007).

2.5.1.4 Escalabilidade

Em razão do volume de tráfego processado por uma aplicação tender ao crescimento, a estrutura do banco de dados pode se torna um gargalo para o desempenho. Uma forma de lidar com o aumento da carga de dados é utilizando a replicação. Quando o volume de dados aumenta, cresce o número de réplicas, e cada uma poderá tratar uma carga de dados. Isso permite que o sistema possa gerenciar de forma mais eficiente o crescimento dos registros e de usuários ao longo tempo (YADAVA, 2007).

O BerkeleyDB é um banco escalonável em vários aspectos. A própria biblioteca de banco de dados é compacta, o que significa que é pequena o suficiente para ser executada em sistemas embarcados, mas ainda pode tirar proveito de *gigabytes* de memória e *terabytes* de disco, se você estiver usando um *hardware* que possui esses recursos (ORACLE, 2014).

2.5.1.5 Processamento de consultas

As consultas dos registros armazenados em um modelo de Chave-Valor, só podem ser processadas através da chave, pois caso seja necessário executar uma pesquisa passando por exemplo um atributo da coluna valores, não será possível fazer uso do banco de dados. Porém pode ser difícil encontrar qual a chave que o usuário deseja, sendo necessário algum método para contornar essa situação (FOWLER e SADALAGE, 2013).

Para realizar as três operações básicas, que são a inserção, alteração e a deleção de registros dentro do banco, é necessário criar um documento com a programação utilizando a linguagem C. Na busca de uma informação no banco não é diferente, pois é importante saber a chave que está sendo pesquisada, para assim criar uma programação lógica que faça a varredura no banco, pois esse gerenciador não possui linguagem de consulta nativa.

2.6 CONSIDERAÇÕES DO CAPÍTULO

Com um estudo mais aprofundando sobre cada modelo de banco de dados NoSQL e os respectivos gerenciadores escolhidos, é possível avançar para outra etapa, onde será feita as implementações nos bancos descritos no trabalho. Nota-se a diferença dos modelos, mesmo eles utilizando a mesma classe de banco de dados, a NoSQL, deixando mais interessante e instigante a próxima fase dessa pesquisa, a qual será possível comparar se a teoria é semelhante na prática.

O Quadro 1 mostra quais sistemas operacionais são suportados pelos sistemas gerenciadores escolhidos para o teste desse trabalho. Além disso, pode-se observar qual linguagem de programação foi utilizada para cada um deles, sendo que alguns garantem uma propriedade essencial aos modelos relacionais, a ACID.

SGBDs	**S.O Suportado**	**Linguagem**	**Propriedade ACID**
Neo4j	Linux, Mac e Windows	Java	Sim
Cassandra	Linux, Mac, Unix e Windows	Java	Não
MongoDB	Linux, Mac, Unix e Windows	C++	Não
BerkeleyDB	Linux e Windows	C	Sim

Quadro 1: Comparativo entre os SGBDs NoSQL
Fonte: Autoria própria

O próximo capítulo apresentará o processo de criação da base de dados, em cada banco escolhido como representante de um dos modelos disponíveis de Banco de Dados NoSQL, detalhando também o processo de manipulação dos dados, permitindo verificar na prática os processos de inclusão, exclusão, alteração e de recuperação de elementos dos bancos de dados.

3 EXPERIMENTO

Este capítulo apresenta os testes realizados com cada ferramenta que representa um modelo disponível na tecnologia NoSQL, estando dividido em seis seções. A seção 3.1.descrever os critérios adotados para a realização dos testes. A seção 3.2 discute os testes realizados no SGBD objeto-relacional PostgreSQL, com o objetivo de exemplificar os comandos que serão executados nas ferramentas NoSQL, identificando as semelhanças e diferenças com a relacional.

A seção 3.3 apresenta os testes na modelo de família de colunas Cassandra. Os experimentos no modelo de documentos são discutidos na seção 3.4, onde é utilizada a ferramenta MongoDB. A seção 3.5 descreve o modelo de Grafos, no qual foi utilizado o SGBD Neo4j. E a seção 3.6 diz respeito ao modelo de Chave-Valor, representado pela ferramenta BerkeleyDB.

A última seção apresenta um quadro resumo dos comandos apresentados neste trabalho.

3.1 CRITÉRIOS DE ANÁLISE

Em relação a arquitetura do computador utilizado para executar os comandos dos bancos de dados NoSQL, esta possui um processador Intel, modelo i7-45UU com velocidade de 1.80 GHz (*Gigahertz*) até 3.00 GHz. A memória RAM (*Random Access Memory*) instalada é de 8 GB (*Gigabytes*), o sistema operacional utilizado é o Windows 8.1 de 64 *bits* e a capacidade em disco rígido (HD) é de 01 TB (*Terabyte*).

Para realizar o experimento dos bancos de dados NoSQL Cassandra, MongoDB, BerkeleyDB e Neo4j, decidiu-se que as operações realizadas, também seriam feitas no banco de dados relacional PostgreSQL na versão 9.4, possibilitando a diferenciação dos comandos disponíveis na Linguagem de Definição de Dados (DDL) e Linguagem de Manipulação de Dados (DML).

Os primeiros comandos DDL testados são responsáveis pela estruturação da base de dados, do esquema físico propriamente dito, permitindo a criação do banco de dados e das tabelas no modelo relacional.

Na tecnologia NoSQL os comandos DDL têm diferentes nomes, dependendo do modelo. No chave-valor o que se relaciona com tabelas são os *buckets*, no banco de dados de documentos são as coleções e no modelo famílias de colunas são as famílias. No modelo de grafos a estrutura é bem distinta das demais, se baseando em vértices e arestas, assim não havendo correlação com as tabelas do modelo relacional.

Além de comandos relacionados a criação do banco, também são descritos os comandos que permitem alterar e excluir as estruturas do esquema do banco de dados. .

Também serão exemplificados os comandos de manipulação do banco de dados disponíveis na DML, ou seja, a inserção, alteração, exclusão e consulta dos dados armazenados no banco.

3.2 BANCOS DE DADOS RELACIONAL - POSTGRESQL

O primeiro passo para se utilizar o SGBD é a criação da base de dados, assim dando início a implementação dos comandos DDL e DML. Os comandos mostrados foram retirados da respectiva documentação oficial do banco de dados PostgreSQL (POSTGRESQL, 2015). O comando completo para a criação de uma base no banco PostgreSQL é feito como descrito na Figura 6.

```
CREATE DATABASE name
    WITH OWNER = user_name
      ENCODING = encoding
            LC_COLLATE = lc_collate
            LC_CTYPE = lc_ctype
            TABLESPACE = tablespace
            CONNECTION LIMIT = connlimit;
```

Figura 6 – Sintaxe do comando de criação do banco
Fonte: Autoria própria

Os parâmetros podem ser definidos dependendo da necessidade de utilização do usuário, caso contrário, o próprio SGDB irá definir valores padrões para os argumentos caso não sejam mencionados.

Name: nome dado à base de dados.

User_name: nome do proprietário do banco.

Encoding: codificação de caracteres que será usada no banco.

Lc_collate: ordem de comparação das *strings*.

Lc_ctype: classificação dos caracteres no banco criado.

Tablespace: nome da *tablespace* que será associada ao banco.

Connlimit: número de conexões simultâneas suportadas.

O PostgreSQL possui uma interface gráfica através da ferramenta pgAdmin III, a qual está ilustrada na Figura 7. E para essas implementações foi usada a linguagem SQL.

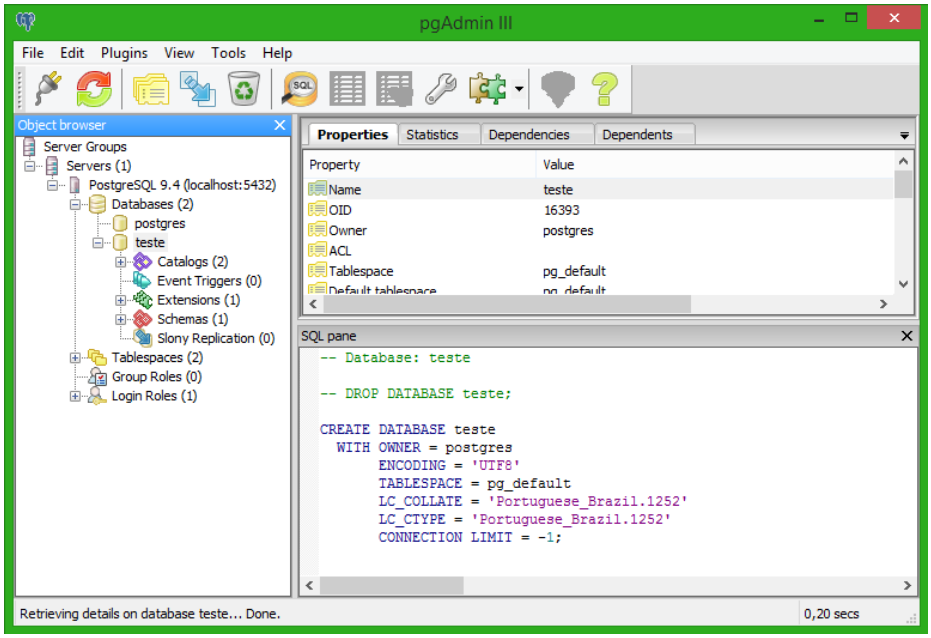

Figura 7 – Interface gráfica utilizando o pgAdmin III
Fonte: Autoria própria

O banco de dados para teste foi criado através da ferramenta de interface gráfica, sendo definido apenas o campo nome da base, mostrado na Figura 8, onde os demais parâmetros utilizaram como parâmetro dos valores *default* (padrão).

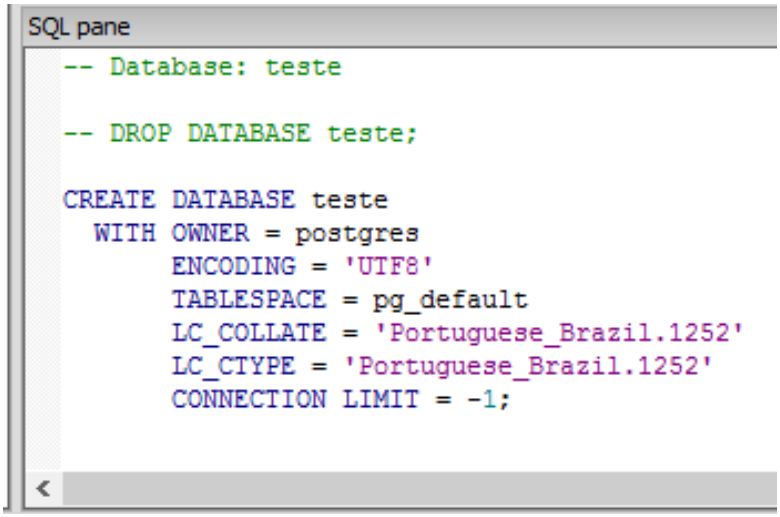

Figura 8 – Comando de criação da base de dados
Fonte: Autoria própria

As palavras em negrito são consideradas reservadas da linguagem SQL. Com diversos recursos e abordagens, o comando *create table* pode ser usado de inúmeras formas. A Figura 9 mostra de forma resumida as principais clausulas do comando *create*, sendo exemplificado o significado de cada um dos seus parâmetros, obrigatórios e opcionais, onde o último está entre colchetes.

CREATE [[**GLOBAL** | **LOCAL**] { **TEMPORARY** | **TEMP** } | **UNLOGGED**] **TABLE** [**IF NOT EXISTS**] table_name ([{ column_name data_type [**COLLATE** collation] [column_constraint [...]] | table_constraint | **LIKE** source_table }]);

Figura 9 – Sintaxe do comando de criação de tabela
Fonte: Autoria própria

O significado dos parâmetros na Figura 9 foram explicados da seguinte forma:

Table_name: nome da tabela criada.

Column_name: nome da coluna que será criada na tabela.

Data_type: tipo de dado determinado para todas as colunas.

Collation: é um parâmetro optativo que serve para agrupar colunas.

Column_constraint: restrição para uma única coluna.

Table_constraint: restrição para tabela destinada a uma ou mais colunas.

Source_table: especifica uma tabela para copiar os nomes das colunas, os tipos de dados e as restrições não nulas.

Usou-se o comando *create table* para demonstrar como ele é usado de dentro da ferramenta, a ideia foi implementar uma tabela com o nome de cliente com alguns atributos para exemplificar um dos usos desse comando, como pode ser visto na Figura 10.

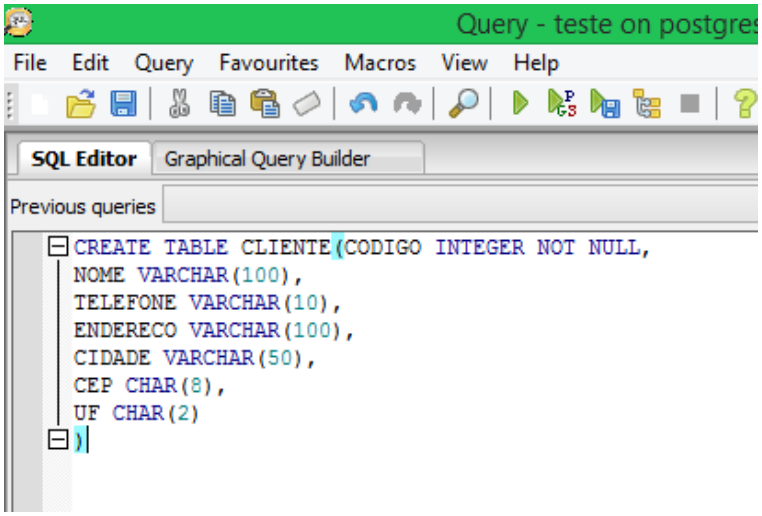

Figura 10 – Comando de criação de tabela
Fonte: Autoria própria

O comando *alter* tem diversas funções, e é utilizado para a inserção de uma chave primária e outra estrangeira, basicamente o comando é composto da seguinte forma de acordo com a Figura 11.

ALTER TABLE [**ONLY**] name [*]
action [...]

Figura 11 – Sintaxe do comando de alteração
Fonte: Autoria própria

O argumento *action* dessa sentença pode ser desmembrado para diversos valores, onde para esse teste será usado o comando *add* para adicionar as chaves. O *name* seria o nome da tabela que será feita uma alteração.

A Figura 12 ilustra como foi feito no PostgreSQL a criação das chaves primária e estrangeira de acordo com a tabela criada anteriormente, na qual para a chave estrangeira foi necessária a criação de uma outra tabela,

denominada “estado”, junto com a criação de uma chave primária para o campo “uf”.

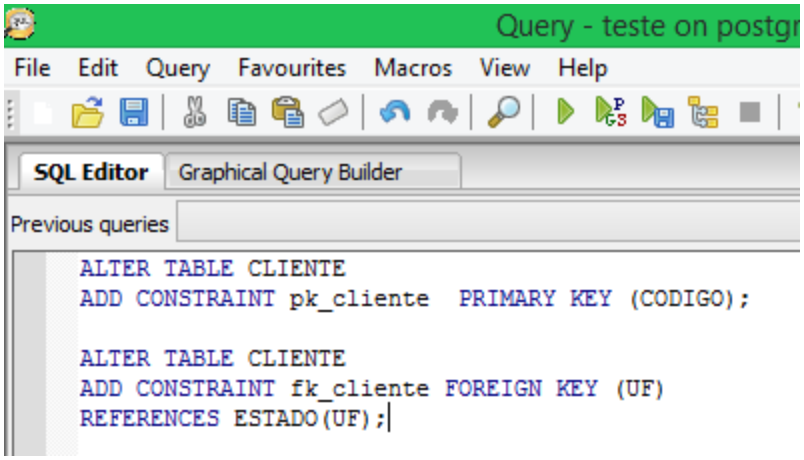

Figura 12 – Comando de alteração para criação das chaves
Fonte: Autoria própria

O último comando DDL a ser apresentado é o *drop*, o qual tem a utilidade de desfazer ou excluir estruturas criadas dentro do banco de dados. Sua estrutura básica para destruir uma tabela é composta de acordo com a Figura 13.

DROP TABLE [**IF EXISTS**] name [...] [**CASCADE** | **RESTRICT**]

Figura 13 – Sintaxe do comando drop
Fonte: Autoria própria

A maioria dos comandos são opcionais para se realizar a exclusão, onde o principal fator para o uso do comando é ter o *name*, o qual será o nome da tabela a ser desfeita da estrutura. A fim de exemplificar, foi usado o *drop* para uma tabela com o nome de cidade, que pode ser visto na Figura 14.

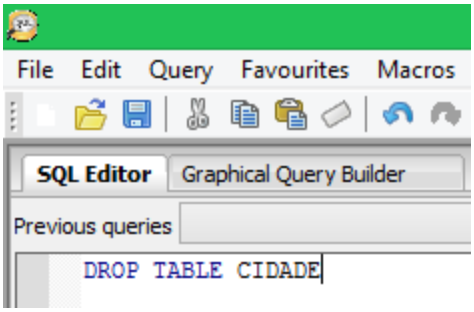

Figura 14 – Comando de exclusão de uma tabela
Fonte: Autoria própria

Se tratando dos comandos DDL, esses são os mais básicos e essenciais na utilização do SQL, onde a linguagem não se limita somente a esses comandos. O próximo passo é utilizar a linguagem DML, para mostrar as funções mais utilizadas do modelo relacional.

São implementados os comandos *insert*, *update*, *delete* e *select*, em conjunto com algumas cláusulas. Para começar, será mostrado como o comando *insert* é dividido, onde os comandos delimitados por colchetes são opcionais da linguagem, para deixar o código cada vez mais preciso dependendo das regras utilizadas pelo DBA (*DataBase Administrator*). Assim a descrição do código pode ser visto na Figura 15.

```
INSERT INTO table_name [ ( column_name [...] ) ]
                { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [...] ) [...]| query }
                [ RETURNING * |output_expression[[AS] output_name] [...] ]
```

Figura 15 – Sintaxe do comando insert
Fonte: Autoria própria

Abaixo estão os significados das expressões que podem ser usadas para inserir uma informação em determinada tabela.

Table_name: nome da tabela que será usada para o armazenamento.

Column_name: nome da coluna onde o valor será atribuído.

Expression: Valor que será incumbido na coluna informada no parâmetro column_name.

Query: Uma consulta que fornecerá as linhas para as informações serem inseridas.

Output_expression: Mostra o que foi inserido dependendo do argumento utilizado.

Output_name: Nome a ser usado para retornar uma coluna.

No teste do *insert* foi utilizada a tabela cliente para as informações serem guardadas. Vale ressaltar que para o campo “uf”, o mesmo deve estar armazenado na tabela estado, pois é chave estrangeira na tabela cliente, assim o comando ficou escrito conforme ilustrado na Figura 16.

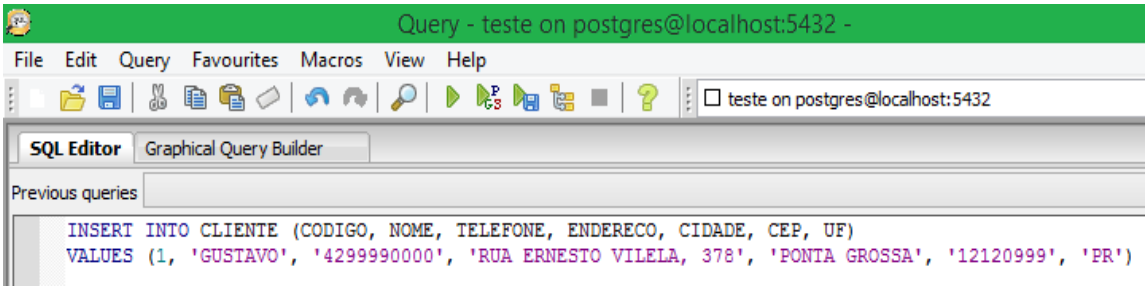

Figura 16 – Comando de inserção
Fonte: Autoria própria

O próximo caso é utilizar o comando *update*, o qual é usado quando deseja-se alterar uma ou mais informações dentro do banco, assim o usuário escolhe a tabela que deseja alterar, o campo e escreve o novo valor que será inserido, e este poderá ser alterado conforme as cláusulas impostas após a palavra reservada *where*. A sintaxe base do comando está ilustrada na Figura 17.

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
        SET { column_name  = { expression | DEFAULT } |
        ( column_name  [...] ) = ( { expression | DEFAULT } [...] ) } [...]
        [ FROM from_list ]
        [ WHERE condition | WHERE CURRENT OF cursor_name ]
                [ RETURNING  * | output_expression [ [ AS ] output_name ]
[...] ]
```

Figura 17 – Sintaxe do comando update
Fonte: Autoria própria

Os parâmetros iguais ao comando *insert* possuem a mesma função dentro dessa estrutura, somente o *from_list* é diferente, significando que pode ser usado para incluir outras tabelas, para que as colunas destas apareçam na cláusula *where*.

O *alias* é um apelido dado para substituir o nome original da tabela, e o parâmetro *condition* é o código retorna um tipo boleano, onde essa condição for verdadeira a informação será alterada. Logo após a palavra reservada *set* será dado o novo valor ao atributo escolhido, como na Figura 18, onde foi alterado o campo cep do cliente Gustavo. Cabe ressaltar que a linguagem SQL é *case sensitive*.

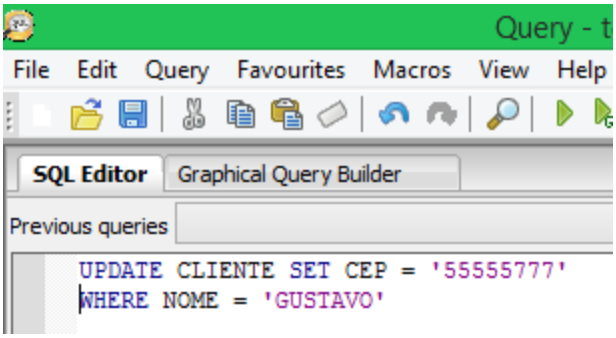

Figura 18 – Comando de alteração
Fonte: Autoria própria

O próximo comando utilizado foi o *delete***.** Sua estrutura básica tem parâmetros semelhantes aos dois códigos anteriores, mas sua utilização serve para se eliminar uma ou mais tuplas contidas na base de dados. O usuário deve especificar a tabela e a condição de exclusão logo após a palavra reservada *where*. Assim a sintaxe é mostrada na Figura 19.

DELETE FROM [**ONLY**] table_name [*] [[**AS**] alias]
[**USING** using_list]
[**WHERE** condition | **WHERE CURRENT OF** cursor_name]
[**RETURNING** * | output_expression [[**AS**] output_name]
[...]]

Figura 19 – Sintaxe do comando delete
Fonte: Autoria própria

A Figura 20 mostra um exemplo de exclusão considerando a tabela estado, e a condição é que o campo “uf” da tabela seja igual aos caracteres “AB”.

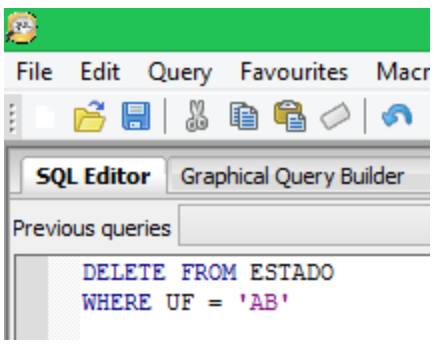

Figura 20 – Comando de exclusão
Fonte: Autoria própria

Um dos comandos mais importantes em um banco de dados é a consulta, ou seja, o comando *select*. Ele possui inúmeros parâmetros e argumentos obrigatórios e opcionais, podendo ou não serem utilizados pelo usuário em uma pesquisa, como ilustrado na Figura 21.

```
SELECT [ ALL | DISTINCT [ ON ( expression [...] ) ] ]
            [ * | expression [ [ AS ] output_name ] [...] ]
            [ FROM from_item [...] ]
            [ WHERE condition ]
            [ GROUP BY expression [...] ]
            [ HAVING condition [...] ]
            [ WINDOW window_name AS ( window_definition ) [...] ]
            [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
           [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS
{ FIRST | LAST } ] [...] ]
            [ LIMIT { count | ALL } ]
            [ OFFSET start [ ROW | ROWS ] ]
            [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
                    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY
SHARE } [ OF table_name [...] ] [ NOWAIT ] [...] ]
```

Figura 21 – Sintaxe do comando de seleção
Fonte: Autoria própria

Pelo fato do comando ter diversas funções e subcomandos, é difícil descrever os tipos de pesquisas que o *select* pode obter. Resumindo, o usuário pode selecionar uma ou mais tabelas, organizar por grupos, ordenar de forma crescente ou decrescente, começar a selecionar de um ponto específico, usar funções de agregação que permitem contar, obter o máximo e o mínimo valor de determinado campo, entre outros milhares de combinações.

Para este trabalho foi elaborado um comando *select* para mostrar o nome do estado representado pela sigla “PR”, conforme ilustrado na Figura 22.

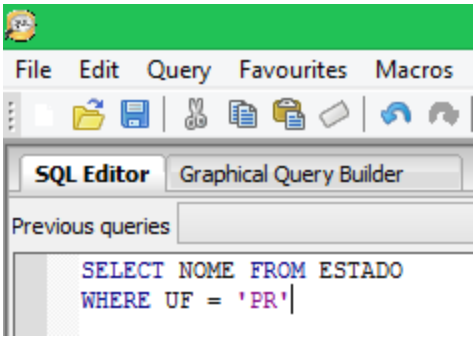

Figura 22 – Comando de seleção
Fonte: Autoria própria

O resultado do comando é apresentado na Figura 23.

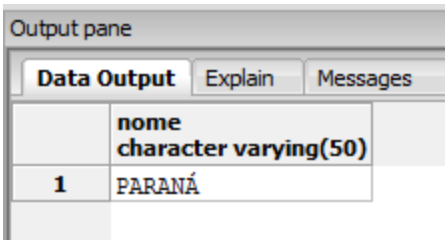

Figura 23 – Retorno da seleção
Fonte: Autoria própria

Outros recursos importantes e utilizados no modelo relacional são as *triggers* e as *functions* ou também chamadas de *stored procedures,* permitindo criar restrições no bancos de dados que atendam a condições específicas de um usuário e/ou aplicação. Mas as mesmas não foram abordadas no trabalho, pois nos modelos NoSQL, seus respectivos bancos não suportam essas aplicações.

3.3 MODELO NOSQL – NEO4J

A seção é dedicada aos bancos de dados de grafos da tecnologia NoSQL, sendo representados pelo Neo4j. Entre os SGBD's estudados e implementados, este é o que fica fora dos padrões habituais de um banco de dados relacional, dificultando sua diferenciação com os outros da mesma tecnologia e os que fazem parte do modelo relacional.

A interface apresentada na Figura 24 permite criar uma base de dados no SGBD Neo4j. Inicialmente deve-se selecionar a opção *browse* para

estabelecer o local onde será armazenado o banco de dados. Feita a escolha do local, deve-se clicar no botão *start* para se iniciar o SGBD e quando o mesmo estiver pronto, o campo *status* ficará verde, mostrando o *link* que precisa ser acessado para se iniciar a implementação da base, onde é feito por meio de interface *web*.

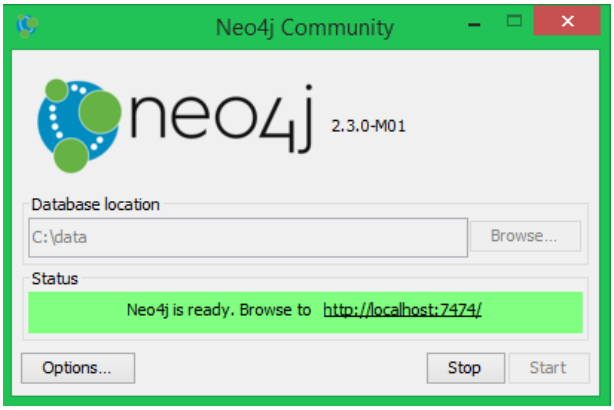

Figura 24 – Interface para inicialização do Neoj4
Fonte: Autoria própria

Ao abrir a página no navegador da *internet*, o usuário irá se deparar com a interface para trabalhar com o SGBD, como mostrado na Figura 25. Esta página é usada para a implementação de todos os códigos suportados pelo banco quando se trabalha no ambiente *Windows*. Para a realização dos testes no Neo4j, foi utilizada a linguagem Cypher, nativa do Neo4j.

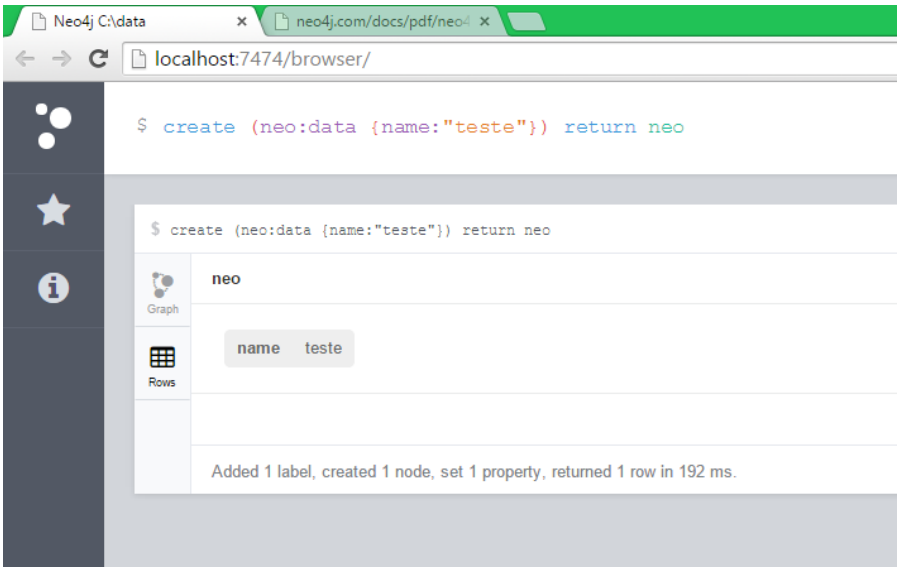

Figura 25 – Interface web do Neo4j
Fonte: Autoria própria

Como o Neo4j é um banco do modelo de grafos da tecnologia NoSQL, é complicado comparar suas características com as dos outros bancos. Por exemplo, ele não possui nenhuma estrutura semelhante a uma tabela, uma coleção, um documento ou até mesmo uma família de colunas. Sua estrutura é baseada em vértices e arestas, onde o primeiro é responsável por armazenar os registros e o segundo tem o objetivo de relacionar os registros. Os comandos mostrados foram inspirados na documentação oficial do Neo4j (NEO4J, 2015).

Logo após a interface do banco de dados ser aberta, pode-se começar a manipular o mesmo, sendo que o primeiro passo é a criação dos nodos. Neste SGBD existem diversas formas para o usuário inserir os nós, pois é possível criar desde um simples nodo até realizar um relacionamento com outro nodo no mesmo comando. Primeiramente foi enfatizado o código para a palavra reservada *create*, assim o Quadro 2 mostra três sintaxes utilizando esse comando.

Create (n)	Create (n:cliente {name:"Test "})	Create n-[:knows]->m
Opção a	Opção b	Opção c

Quadro 2: Sintaxe do comando *create*
Fonte: Autoria própria

Têm-se três sintaxes relacionadas aos comandos *create*, começando da esquerda para a direita, o primeiro código simplesmente cria um nodo, já o segundo comando, além de criar o nó atribuindo um rótulo para o mesmo de cliente, insere uma propriedade *name* com valor “Test”.

A última implementação cria os nodos n e m, relacionando ambos por uma aresta denominada *knows*. No teste foram implementadas duas dessas sintaxes e usado um recurso da interface para mostrar a estrutura do grafo formado, para isso no segundo teste foi usado o comando *return*. Assim os exemplos ficaram conforme as Figuras 26 e 27.

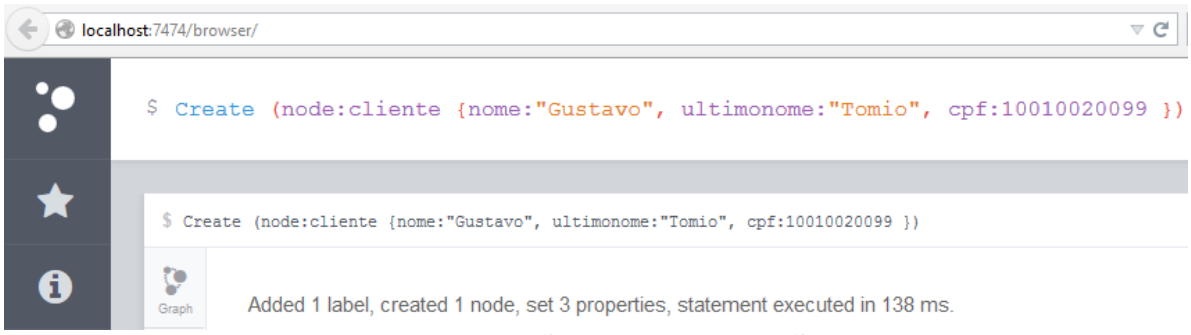

Figura 26 – Comando de criação de nodo e inserção de registros
Fonte: Autoria própria

O resultado produzido pelo comando apresentado na Figura 26 foi a criação de um nodo com rótulo cliente, onde o mesmo possui três registros incluídos neste nó.

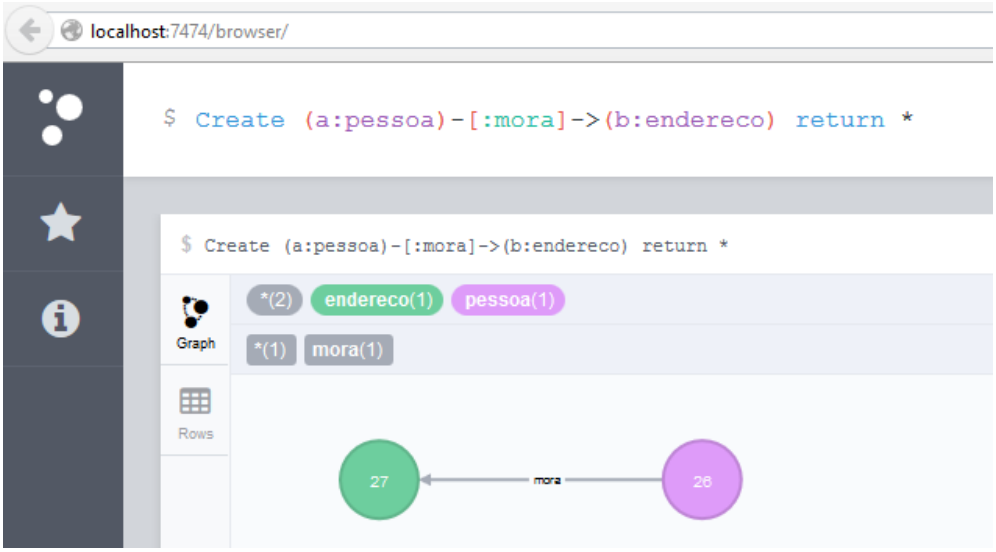

Figura 27 – Comando de criação de nós com relacionamento
Fonte: Autoria própria

Na Figura 27 ilustra o comando utilizado para criar dois nós, no qual ambos possuem rótulos pessoa e endereço, com um relacionamento denominado mora. Os números dentro da representação dos vértices são determinados pelo SGBD.

O Neo4j é basicamente divido em comandos de escrita e leitura, juntamente com cláusulas e algumas funções. Os testes iniciais concentraram-se nos comandos de escrita, e logo após os comandos de leitura.

O código a seguir é o *set*, sua sintaxe está ilustrada na Figura 28, tendo a função de atualizar nodos e relacionamentos.

```
MATCH (nodo { row1: valor_busca })

SET n.row2 = novo_valor
```

Figura 28 – Sintaxe para utilização do comando set
Fonte: Autoria própria

Para a utilização do comando *set* é necessário o uso do *match* antes, pois este serve para fazer uma busca na base, semelhante ao *select* no modelo relacional. Inicialmente deve-se especificar qual nó terá uma atualização e qual a propriedade que ele irá buscar, logo em seguida no comando *set* é possível escolher como a atualização será feita, pois pode ser atualizado um campo existente ou adicionado um novo campo com um respectivo valor. Este comando permite que mais de um campo possa ser alterado, ficando a critério do utilizador. A Figura 29 ilustra um teste utilizando o comando *set.*

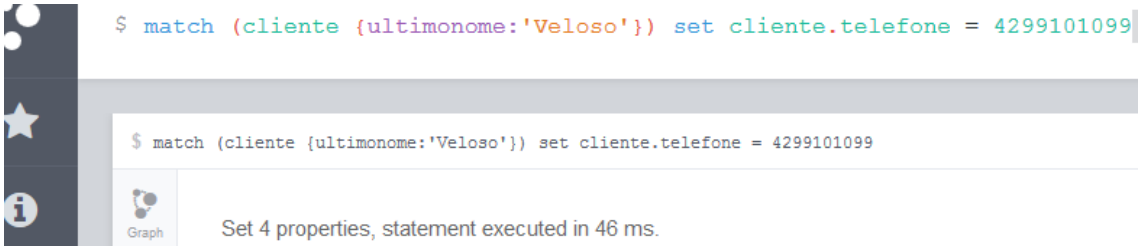

Figura 29 – Comando de atualização de nós
Fonte: Autoria própria

O teste mostra que nos nodos de nome cliente, houve uma busca através do campo “ultimonome”, no qual a pesquisa tem de retornar nodos cliente que possuem o valor “Veloso”, assim se encontrado, os nós serão atualizados com o campo “telefone” com o valor mostrado na Figura 29. Caso o telefone não exista no nodo, o mesmo será criado e se já existir, uma atualização é executada. Neste caso houve quatro mudanças e/ou criações do campo “telefone”.

O próximo comando testado no Neo4j foi o de apagar nodos e relacionamentos, onde o comando é representado pela palavra reservada *delete*. Ao usar o código do *delete,* este apagará a parte da estrutura e não

somente os dados acoplados aos nodos. Para exemplificar o uso desse comando, a Figura 30 mostra a sintaxe do mesmo.

```
MATCH (n: test)
DELETE n
```

Figura 30 – Sintaxe do comando delete
Fonte: Autoria própria

Novamente, como na utilização do comando de atualização *set*, é necessário usar o *match* para buscar os nodos e relacionamentos que o usuário deseja excluir do grafo. As Figuras 31 e 32 mostram a exclusão de nodos e relacionamento, respectivamente. Assim, a implementação demonstra como o comando é usado para mais de um fim dentro da aplicação.

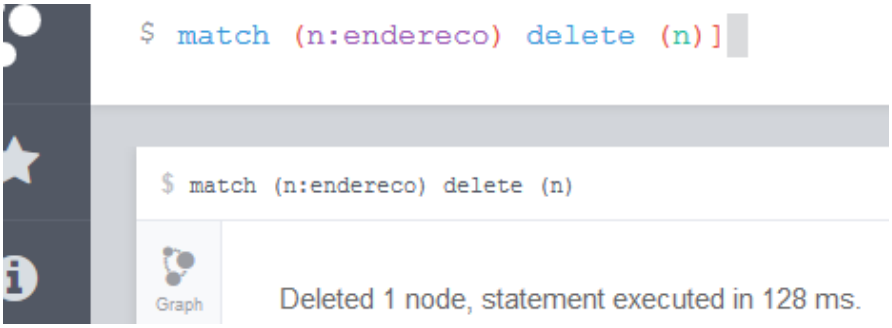

Figura 31 – Comando de exclusão do nodo
Fonte: Autoria própria

O teste apresentado na Figura 31 realizou a exclusão de todos os nodos com nome de endereço, e neste caso havia somente um nodo, o qual foi excluído do grafo. Caso houvesse algum relacionamento com o nodo endereço, nenhum nodo seria apagado utilizando esse comando.

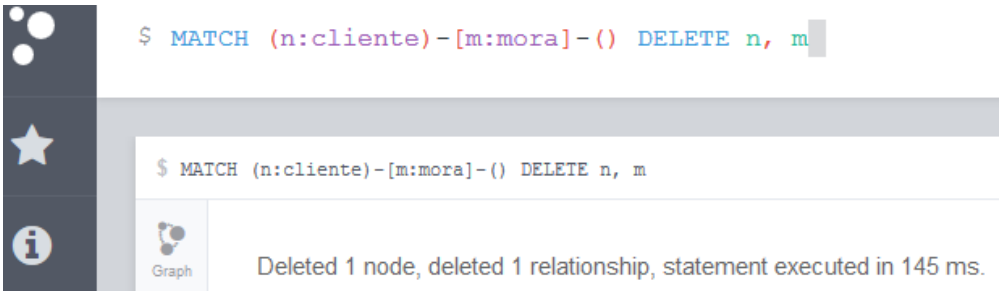

Figura 32 – Comando de exclusão de relacionamento e nodo
Fonte: Autoria própria

O comando especificado na Figura 32 permitiu excluir o nodo “cliente” com relacionamento “mora”, onde o cliente e o relacionamento foram excluídos da estrutura, onde os nodos que estavam interligados com “cliente” pelo relacionamento “mora” permanecem no grafo.

Seguindo essa linha de exclusões, o próximo comando a ser testado será o *remove*, o qual tem o objetivo de remover propriedades e rótulos dos nodos e não a estrutura de armazenamento. Pode-se perceber que na Figura 33, que mostra a sintaxe do comando *remove*, o comando *match* está presente novamente, ou seja, enfatiza sua importância em relação a maioria dos códigos utilizados no Neo4j.

```
MATCH (nodo { propriedades })

REMOVE n
```

Figura 33 – Sintaxe do comando delete
Fonte: Autoria própria

A sintaxe do comando demonstra que deve-se fazer uma busca no grafo para achar o local da remoção de rótulos e/ou das propriedades. Assim, em seguida o comando *remove* indica o que será removido do nodo. Para implementar e testar esse comando, foram feitos dois testes. O primeiro representado pela Figura 34, ilustrando como é feita a exclusão de uma propriedade de um nodo, e na Figura 35, como um rótulo é removido.

Figura 34 – Comando de remoção de propriedade
Fonte: Autoria própria

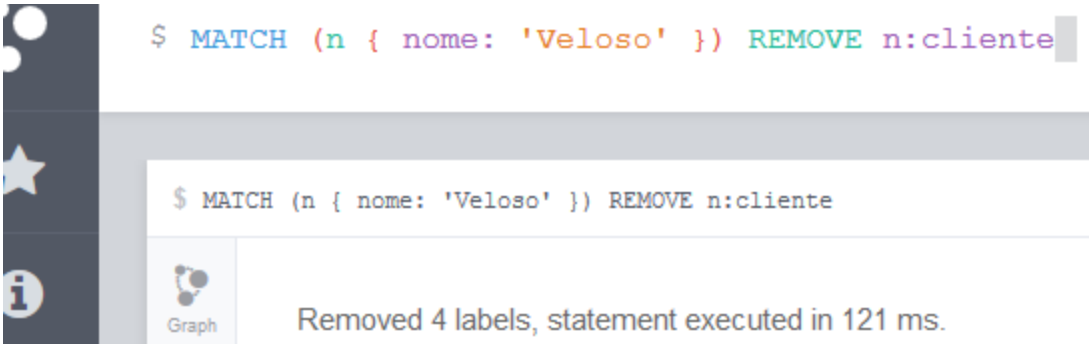

Figura 35 – Comando de remoção de rótulo
Fonte: Autoria própria

Na Figura 34 como explicado anteriormente, o teste foi usado para remover uma propriedade dentro do nodo escolhido, que seria os com rótulo “cliente”, assim foi removido o campo “cpf” de todos os nodos que tinham “ultimonome” “Veloso”. A Figura 35 excluiu o rótulo do nodo, ou seja, o cliente foi retirado dos nodos que possuíam nome “Veloso”.

Para finalizar os comandos testados no Neo4j, será exemplificado o comando *match*, onde o mesmo já foi visto anteriormente compondo outros códigos. A sintaxe mostrada na Figura 36 somente exemplifica a forma geral de busca nesse banco de dados, pois há inúmeras maneiras de realizar uma pesquisa dentro do SGBD.

```
MATCH (n)

RETURN n
```

Figura 36 – Sintaxe do comando match
Fonte: Autoria própria

O seguinte código descrito na Figura 36 demonstra simplesmente uma busca geral de todos os nodos do grafo, onde no lugar do ‘n’, qualquer outra letra ou cadeia de caracteres poderia ser usada. Vale ressaltar que a busca pode ser feita de muitas maneiras, através dos registros que estão nos nodos, dos próprios relacionamentos. A linguagem Cypher possui algumas cláusulas e expressões semelhantes ao modelo relacional. A Figuras 37 e 38 ilustram dois exemplos básicos de buscar utilizando o *match*.

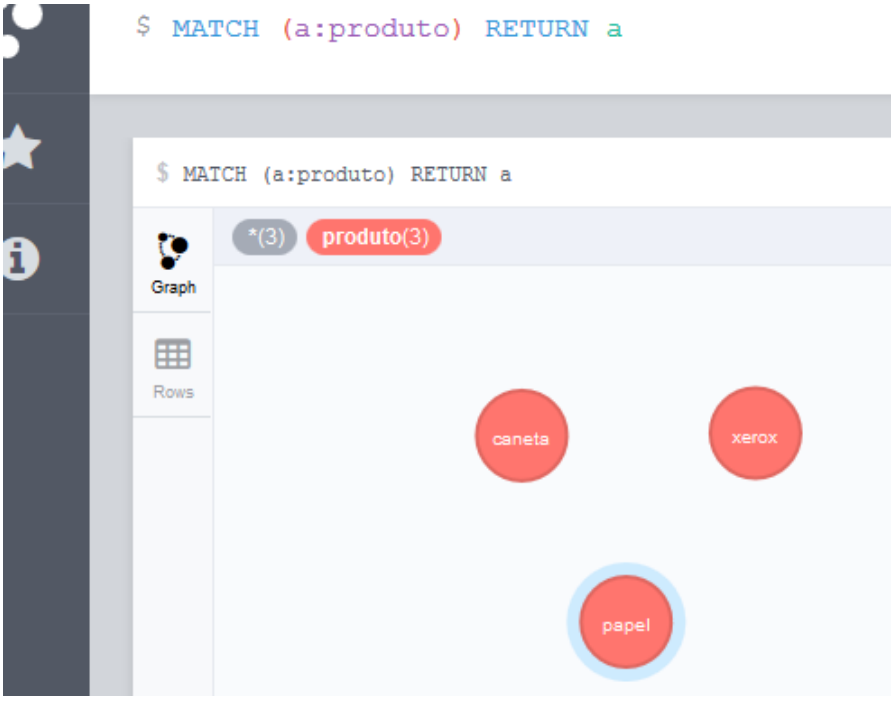

Figura 37 – Comando de busca de nodos
Fonte: Autoria própria

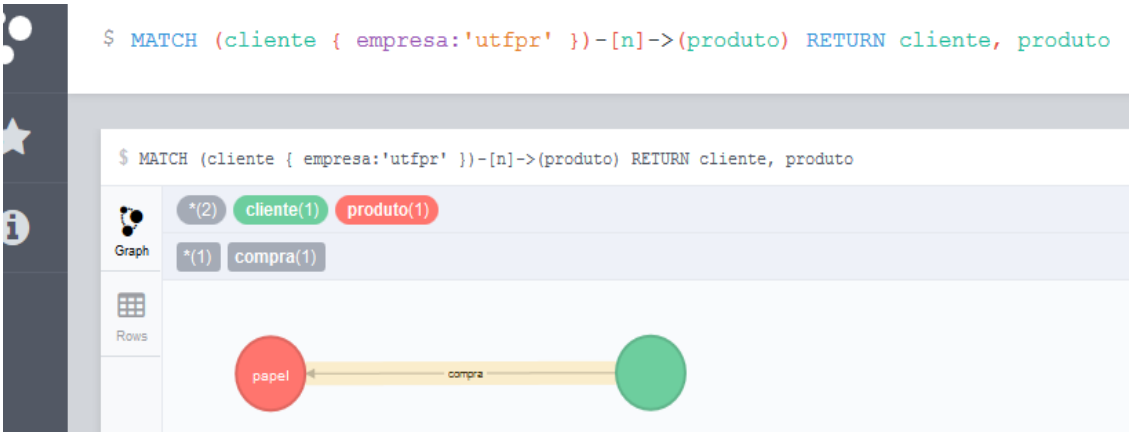

Figura 38 – Comando de busca de nodos com relacionamento
Fonte: Autoria própria

A Figura 37 faz uma busca em nodos com rótulos denominados produto, ou seja, o banco fez uma varredura no grafo e retornou todos os nodos que possuem o rótulo especificado, não possuindo mais nenhum tipo de cláusula aparente.

Já na Figura 38, a busca é feita em cima dos nodos cliente, porém somente aqueles que possuírem o campo empresa com valor ‘utfpr’, e ainda um relacionamento com nodo de “produto” serão retornados. Os comandos *return* servem apenas para mostrar de forma visual os nodos e arestas buscadas no grafo.

Passando o cursor do mouse sobre o nodo que possui nome igual a “xerox”, pode observar que logo abaixo a interface diz o que está guardado dentro do nodo, como pode ser observado na Figura 39. Outra questão interessante está na forma que é retornado a resposta ao usuário, pois além do desenho do grafo, também é possível ver as linhas e os registros das mesmas clicando do lado esquerdo no botão *rows*. Assim a Figura 40 ilustra como o usuário pode ver sua busca.

Figura 39 – Observação do conteúdo do nodo
Fonte: Autoria própria

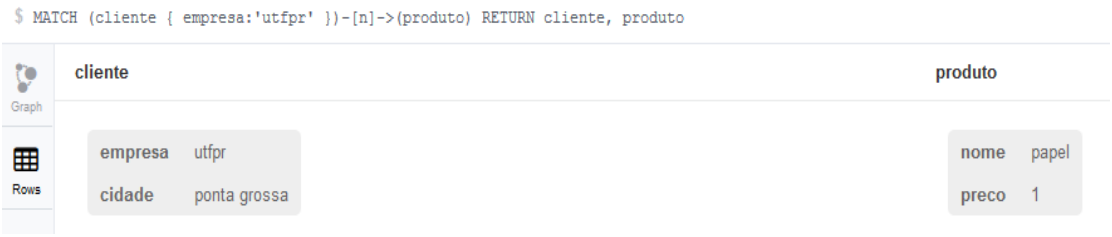

Figura 40 – Outro modo de observar a busca
Fonte: Autoria própria

Para concluir o modelo de grafos da tecnologia NoSQL, segundo a documentação do SGBD (NEO4J, 2015), o uso de *triggers* ou *procedures*, como usadas no modelo relacional, onde possuem papel expressivo dentro de uma base de dados, o Neo4j não possui esses tipos de recursos. Outro fator importante a ser lembrado, é o fato de que os comandos apresentados não esgotam o potencial do SGBD, pois a quantidade de comandos disponíveis é muito maior dos que foram realizados nesse experimento.

3.4 MODELO NOSQL – CASSANDRA

Conforme realizado no PostgreSQL, o mesmo padrão foi adotado para a tecnologia NoSQL, diferenciando os comandos. A primeira mudança começa

no nome do comando de criação da base de dados, sendo chamado de *keyspace*, porém o código é semelhante.

A linguagem utilizada neste SGBD é a CQL (Cassandra *Query Language*), com bastante semelhança a linguagem SQL, e para as sintaxes descritas nesse capítulo foi usada a documentação dos comandos CQL (CASSANDRA, 2015). A Figura 41 ilustra a sintaxe para o comando de criação da *keyspace*.

```
CREATE KEYSPACE <identifier> WITH <properties>
```

Figura 41 – Sintaxe de criação de uma keyspace
Fonte: Autoria própria

A *tag* <*identifier*> seria o nome que se dará a base, logo após vem o <*properties*>, os quais são as propriedades que o usuário vai utilizar em seu banco, podendo ser usadas para definir a replicação dos dados e a durabilidade das escritas, como pode ser visto no teste feito utilizando esse comando, na Figura 42.

As implementações nesse SGBD não serão feitas em uma interface gráfica, mas sim somente nas linhas de comando, a fim de praticá-los e consequentemente fixá-los de forma a aumentar o aprendizado desse banco.

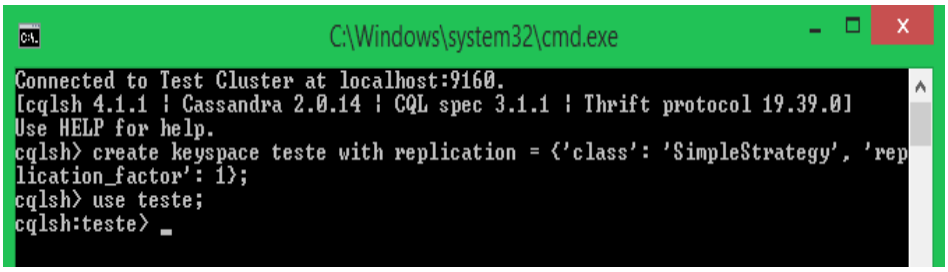

Figura 42 – Criação de um keyspace
Fonte: Autoria própria

A estratégia de replicação é definida no campo ‘class’, onde foi usado uma estratégia simples, ou seja, igual a todos os *clusters*. No campo ‘replication_factor’, o mesmo indica quantas cópias serão feitas, nesse caso apenas uma. O Cassandra possui uma interface *web* através do programa OpsCenter, onde o usuário pode gerenciar os *keyspaces* e também observar o comportamento dos mesmos. A Figura 43 ilustra essa interface.

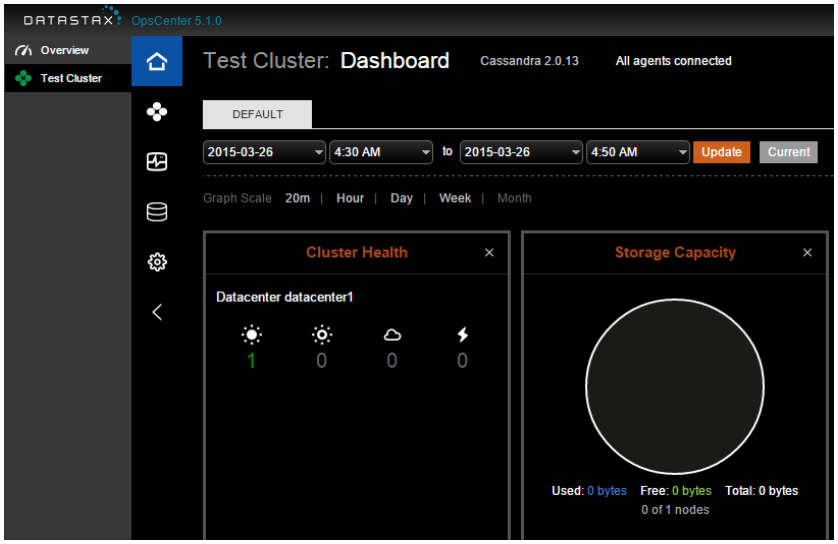

Figura 43 – Interface web do Cassandra
Fonte: Autoria própria

Para compreensão e diferenciação com o modelo relacional, foi usada a linguagem CQL para a o uso da sintaxe e implementações dos testes, assim pode-se inferir que há outras formas de realizar os mesmos comandos utilizando outros tipos de linguagens aceitas pelo Cassandra.

Esse banco de dados utiliza as famílias de colunas para armazenar seus dados, que é semelhante a uma tabela no banco de dados relacional. A única diferença é que no modelo relacional a coluna é a mesma para todas as linhas, entretanto no Cassandra, a coluna pode ser diferente por linha. A Figura 44 mostra o código para a criação dessa estrutura.

```
CREATE (TABLE | COLUMNFAMILY)
                    <tablename> '(' <column-definition> (',' <column-definition>
)* ')' (            WITH <option> ( AND <option>)* )?
```

Figura 44 – Sintaxe do comando de criação de tabela
Fonte: Autoria própria

O campo <*tablename*> equivale ao usuário colocar o nome da tabela ou família de coluna que será criada, o <*column-definition*> é onde ficará o nome da coluna e o seu tipo e na *tag* <*option*>, seria uma configuração especificada pelo usuário, caso o mesmo não queira usar a padrão, possuindo

várias formas de configuração. A Figura 45 mostra como foi feita a 45 do teste da família de coluna cliente, utilizando o comando *create*.

```
cqlsh> use teste;
cqlsh:teste> create columnfamily cliente(
           ... codigo int,
           ... nome text,
           ... telefone text,
           ... endereco text,
           ... cidade text,
           ... cep text,
           ... uf text,
           ... primary key(codigo));
cqlsh:teste> _
```

Figura 45 – Criação de uma família de colunas
Fonte: Autoria própria

Diferente do modelo relacional, onde o usuário pode declarar a chave primária dentro da criação da tabela ou deixar sem criar naquele momento, o Cassandra não permite que a tabela ou família de colunas seja criada sem que haja ao menos uma definição de chave primária. Pode-se observar semelhanças entre as linguagens CQL e SQL vista no tópico anterior. Outra diferença é que nesse SGBD não existe chave estrangeira.

O próximo comando a ser testado foi o *alter*, sendo necessário inicialmente conhecer sua sintaxe, para poder aplicá-lo em um exemplo prático. O código do *alter* é ilustrado na Figura 46.

ALTER (TABLE | COLUMNFAMILY) <tablename> <instruction>

A *tag* <instruction> abre diversas opções, ficando de modo que:

<instruction> = ALTER <identifier> TYPE <type>

| ADD <identifier> <type> | DROP <identifier>

| WITH <option> (AND <option>)*

Figura 46 – Sintaxe do comando alter no modelo de família de colunas
Fonte: Autoria própria

É necessário atribuir o nome da família de colunas com que se vai trabalhar, assim pode-se utilizar quatro comandos, para alterar o tipo de uma determinada coluna, adicionar uma nova coluna, remover uma coluna da tabela e atualizar suas opções, como por exemplo, escolher o modo de armazenamento. Como o tratamento da chave foi feito na criação da família de

colunas, o teste com o comando *alter* teve o objetivo de exemplificar o comando com a ideia de criar uma nova coluna para o cliente, ilustrado na Figura 47.

```
cqlsh> use teste;
cqlsh:teste> alter columnfamily cliente add cpf text;
cqlsh:teste>
```

Figura 47 – Adicionando uma nova coluna
Fonte: Autoria própria

O último comando DDL que foi testado é o *drop*, onde o mesmo é responsável pela exclusão de uma estrutura, como a *keyspace* ou uma tabela, por exemplo. Sua sintaxe é bem simples e está representada com a exclusão de uma família de colunas qualquer, como mostrado na Figura 48.

```
DROP COLUMNFAMILY <tablename>
```

Figura 48 – Sintaxe do comando drop no modelo família de colunas
Fonte: Autoria própria

No lugar do campo *columnfamily* poderiam estar outros argumentos, como *index*, *trigger*, *type* ou até mesmo *keyspace*, onde essa questão irá depender de que estrutura deseja-se eliminar. A Figura 49 mostra como foi usado o comando *drop*, no qual se excluiu uma família de colunas com nome de cidade.

```
cqlsh> use teste;
cqlsh:teste> drop columnfamily cidade;
cqlsh:teste>
```

Figura 49 – Exclusão de uma coluna no modelo família de colunas
Fonte: Autoria própria

Entrando nas codificações da linguagem DML, onde o objetivo principal é a manipulação das informações contidas no banco ou que farão parte do mesmo. Começando com o *insert*, cuja responsabilidade é adicionar dados ao *keyspace* que está sendo utilizado. Sua sintaxe é descrita na Figura 50.

```
INSERT INTO <tablename>
        '('<identifier> (',' <identifier> )* ')'
        VALUES '(' <term-or-literal> (',' <term-or-literal> )* ')'
                (USING <option> (AND <option>)* )?
```

Figura 50 – Sintaxe de inserção no modelo família de colunas
Fonte: Autoria própria

Primeiro deve-se escolher aonde os dados serão inseridos, logo após isso identificar as colunas que receberão as informações, finalizando com os dados propriamente ditos depois da palavra reservada *values*, utilizando a mesma ordem que as colunas foram escritas anteriormente. A questão da *tag* <option> foi vista no comando *alter* desse capítulo. E a título de exemplo, a Figura 51 demonstra como foi feita a inserção.

```
cqlsh> use teste;
cqlsh:teste> insert into cliente(
         ... codigo, nome, telefone, endereco, cidade, cep, uf, cpf)
         ... values(
         ... 1, 'GUSTAVO', '4299991010', 'RUA ERNESTO VILELA, 378',
         ... 'PONTA GROSSA', '10101-777', 'PR', '000.111.222-33');
cqlsh:teste>
```

Figura 51 – Comando de inserção
Fonte: Autoria própria

Caso ocorra a inserção de uma linha no modelo de família de colunas com o mesmo valor da chave primária, o SGBD não informará ao usuário que essa chave já está sendo usada como no modelo relacional, ele sobrescreverá os campos com novos valores.

Para dar continuidade aos comandos DML, o subsequente é o *update*, como o próprio nome diz, serve para a atualização de uma informação gravada em uma ou mais colunas de uma tabela. Seu modo de implementação pode ser visto na Figura 52.

```
UPDATE <tablename>
                (USING <option> (AND <option>)* )?
                SET <assignment> (',' <assignment>)*
                        WHERE <where-clause>
```

Figura 52 – Sintaxe de alteração no modelo família de colunas
Fonte: Autoria própria

Os pontos de interrogação indicam que a cláusula é opcional no comando, sendo que as demais são obrigatórias. O usuário pode definir condições na cláusula *where*, as quais, se atendidas permitirão a execução do comando de alteração, modificando os valores das colunas indicadas na cláusula *set*. A Figura 53 mostra o exemplo do uso dessa sintaxe.

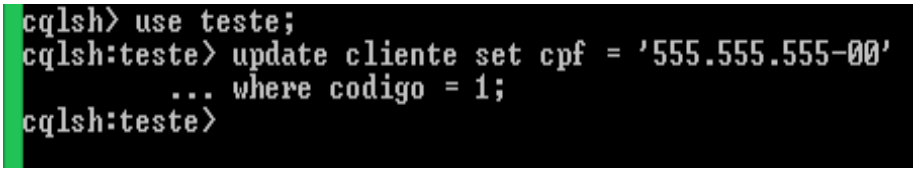

Figura 53 – Comando de atualização
Fonte: Autoria própria

Apesar da semelhança com o comando visto no PostgreSQL, esse código tem um fator importante na escolha do campo a ser buscado pela cláusula *where*, pois não é permitido estabelecer uma condição em uma coluna que não a chave primária.

Um exemplo seria a utilização da coluna nome ao invés da coluna código, da forma nome = 'GUSTAVO', a alteração não seria feita, pelo fato do nome não ser chave primária, ou seja, o gerenciador não encontraria o local para alteração.

O comando *delete* será usado quando o usuário necessitar excluir informações do banco de dados, para isso a Figura 54 demonstra como o comando deve ser usado.

```
DELETE (<selection> ( ','<selection> )* )?
        FROM <tablename>
        (USING TIMESTAMP <integer>)? WHERE <where-clause>
```

Figura 54 – Sintaxe de exclusão no modelo família de colunas
Fonte: Autoria própria

A instrução permite excluir linhas ou dados e uma ou mais colunas. Caso o usuário necessite excluir apenas o conteúdo de uma coluna ou mais, ele identificará essas colunas na *tag* <*selection*> informando ali quais colunas terão suas informações excluídas, mas caso seja necessário excluir as

informações de todas as colunas, nada será informando antes da cláusula *from*.

O critério aplicado para o comando *update* é válido também para o *delete*, onde na cláusula *where* deve conter apenas colunas que são chave primária, caso contrário o SGBD não conseguirá apagar os dados. A Figura 55 ilustra como foi feito o teste.

```
cqlsh> use teste;
cqlsh:teste> delete from cliente where codigo = 2;
cqlsh:teste> _
```

Figura 55 – Comando de exclusão
Fonte: Autoria própria

Nesse caso foram excluídos todos os valores das colunas referente a chave escolhida. Há uma instrução como no modelo relacional denominada *truncate*, a mesma apaga todos os dados de uma determinada tabela, e no caso do Cassandra pode ser de uma família de colunas também.

Se tratando da obtenção de informações no banco de dados, o Cassandra possui uma linguagem com diversos recursos, pois esse é um fator importante na qualidade de um SGBD, ou seja, o quão eficiente ele é na hora de se buscar dados. A codificação básica usada na linguagem CQL para realizar uma operação de busca, é definida com os parâmetros apresentados na Figura 56.

```
SELECT <select-clause>
        FROM <tablename>
        (WHERE <where-clause>)?
        (ORDER BY <order-by>)?
        (LIMIT <integer>)?
        (ALLOW FILTERING)?
```

Figura 56 – Sintaxe do comando de consulta
Fonte: Autoria própria

A instrução começa com a definição das colunas relevantes ao usuário, informando na cláusula *from* quais famílias de colunas serão necessárias para a seleção dos dados. Na cláusula *where* permite estabelecer as condições sob

as quais serão buscados os dados, sendo que as colunas informadas nesta cláusula devem pertencer a chave primária. Caso seja associado uma coluna não chave nesta cláusula, um erro ocorrerá em sua execução.

A *tag* <*order-by*> irá definir somente a ordem como esses dados serão mostrados na tela, podendo ser ordem crescente ou decrescente, dependendo da coluna ou das colunas colocadas como parâmetro.

O uso da palavra *limit*, possibilita estabelecer o máximo de linhas que serão retornadas na consulta. A cláusula *allow filtering* permite que o usuário faça uma busca utilizando outros parâmetros além das chaves, porém essas colunas devem ser criadas como índice para não prejudicar o desempenho das consultas (CASSANDRA, 2015).

A Figura 57 apresenta o comando e o resultado de uma pesquisa utilizando o banco dados Cassandra. A intenção do *select* é de fazer uma varredura na *keyspace* selecionada e mostrar três colunas, as quais foram discriminadas no começo da instrução, sendo elas: código, nome e cpf. Essas informações teriam que ser buscadas na família de colunas cliente, cuja a cláusula diz que deve ser procurado o cliente que possui código igual a 1.

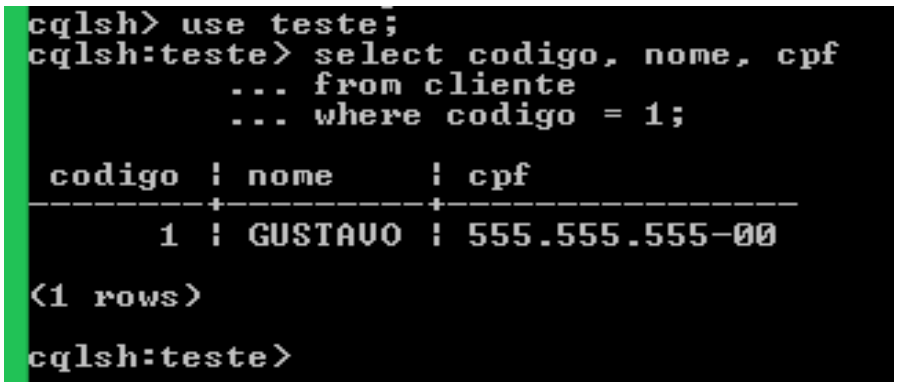

Figura 57 – Seleção e resultado
Fonte: Autoria própria

De acordo com (DATASTAX, 2013), a *trigger* no Cassandra está ainda em fase experimental, pois a mesma é programada utilizando-se de outros programas desenvolvidos, por exemplo, na linguagem Java, armazenados no computador, não possuindo recursos semelhantes aos atualmente disponíveis nos bancos de dados relacionais. A sintaxe no SGBD para essa estrutura é ilustrada na Figura 58.

```
CREATE TRIGGER  (<triggername>)?
                ON <tablename>
                USING <string>
```

Figura 58 – Sintaxe do comando de criação de trigger
Fonte: Autoria própria

O usuário não é obrigado a criar um nome para sua *trigger*, mas logo em seguida deve colocar qual tabela a mesma está relacionada, por último o *using* mostrará realmente onde está a codificação do gatilho, para assim ser executado. Há uma pasta com nome de *triggers* dentro das pastas de instalação do Cassandra, onde devem ser colocados os códigos criados para esse fim. Mesmo não tendo o arquivo com a programação da *trigger*, será mostrado na Figura 59 como é a estrutura de sua criação dentro do SGBD.

```
cqlsh> use teste;
cqlsh:teste> create trigger t_teste on cliente using 'apache.cassandra';
```

Figura 59 – Comando de criação de uma trigger
Fonte: Autoria própria

Funções e Procedimentos não estão disponíveis nesse banco de dados, e como pode-se perceber a *trigger* não é propriamente uma programação do banco e sim um recurso para executar um programa externo ao banco.

3.5 MODELO NOSQL – MONGODB

Em bancos de dados de documentos na tecnologia NoSQL, foi usado o MongoDB para realizar essa diferenciação, entre o modelo relacional e os outros SGBD's não relacionais citados no trabalho. A forma de utilização desse banco possui suas peculiaridades, tornando-o assim diferente dos outros, com poucas similaridades com o PostgreSQL, por exemplo, e até mesmo com os outros brancos que fazem parte da mesma tecnologia.

Duas informações importantes devem ser descritas para o funcionamento desse SGBD após a instalação. A primeira é a necessidade da

criação de uma pasta, nomeada como *data* no local onde o MongoDB foi instalado, e após a criação da pasta, é necessário criar uma subpasta para a mesma, com o nome de ‘db’. Essas pastas servirão para o armazenamento das informações guardadas no banco. O segundo passo é executar o servidor, onde é necessário digitar o comando ‘mongod’ no *prompt* de comando do *Windows*, para assim o servidor ficar disponível, como mostrado na Figura 60.

Figura 60 – Comando de inicialização do servidor MongoDB
Fonte: Autoria própria

Após as operações já descritas, o servidor estará pronto para ser usado pelo usuário, onde o mesmo pode executar o *shell* do banco. Há uma diferença notável para outros bancos de dados, pois o usuário pode acessar as instâncias do MongoDB através do comando *use*, porém caso a instância chamada não exista, a mesma será criada automaticamente pelo SGBD. A sintaxe do comando é ilustrada na Figura 61, valendo ressaltar o uso do manual no site oficial do MongoDB (MONGODB, 2015).

```
use <base_name>
```

Figura 61 – Sintaxe para utilização do banco
Fonte: Autoria própria

Diversos bancos de dados usam esse comando para mudar ou simplesmente usar uma base, mas para o MongoDB pode parecer um pouco diferente caso a base não exista e se coloque um nome na *tag* <*base_name*>. O nome será mostrado como o banco existisse, e quando o usuário começar a criar a primeira estrutura dentro da mesma, ela vai ser automaticamente criada dentro do MongoDB, ou seja, o comando não serve apenas para mudar de um banco para o outro. O exemplo implementado pode ser visto na Figura 62.

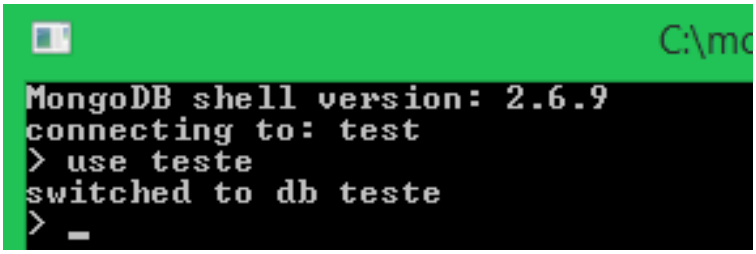

Figura 62 – Criação ou utilização de um banco
Fonte: Autoria própria

No modelo de documentos, os esquemas de armazenamento são divididos em coleções, as quais são equivalentes as tabelas no modelo relacional, e dentro dessas pode haver diversos documentos, onde estes podem ser interpretados como linhas de um SGBD relacional. Para a criação de uma coleção, a sintaxe do comando é como segue a Figura 63.

```
db.createCollection(<name>, { capped: <boolean>,
                              autoIndexId: <boolean>,
                              size: <number>,
                              max: <number>,
                              storageEngine: <document> } )
```

Figura 63 – Sintaxe de criação de uma coleção
Fonte: Autoria própria

Logo após o nome da criação, os parâmetros a seguir são considerados opcionais para o usuário, caso não forem preenchidos não acarretarão eventuais erros, apenas o banco irá definir valores padrões para os mesmos. Assim, os campos opcionais podem ser preenchidos da seguinte forma:

Capped: Define se será usado tamanho máximo nas coleções ou não.

AutoIndexId: Parâmetro *booleano* para criação ou não de índice automático no campo _id.

Size: Especifica um tamanho máximo em *bytes* para uma coleção.

Max: Define o número máximo de documentos na coleção.

StorageEngine: Permite que o usuário especifique a configuração por coleção.

Para a programação do teste para esse comando, foi criada uma coleção com nome de cliente, no banco de dados denominado teste, como

mostrado na Figura 64 e o retorno “ok”: 1, informando que a transação foi bem sucedida. Nota-se que não foram criados campos para essa coleção, pois esse banco não possui esquema definido, não sendo possível a criação dos mesmos utilizando esse comando.

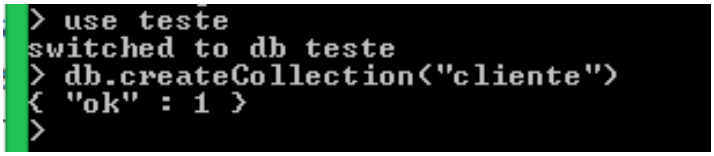

Figura 64 – Comando para criação da coleção
Fonte: Autoria própria

Neste banco de dados o comando para alteração de uma estrutura não existe, onde também o campo de chave, denominado *id* sempre vai existir, sendo atribuído valor automático pelo banco ou atribuído pelo usuário. Há possibilidade de realizar a renomeação de coleções e documentos, a qual a segunda situação será mostrada quando for utilizado o comando *update*. Primeiramente foi utilizado o comando *rename* para ser mudado o nome da coleção, como mostrado na Figura 65.

```
db.collection.renameCollection(target)
```

Figura 65 – Sintaxe de renomeação de uma coleção
Fonte: Autoria própria

No campo *collection* deverá ser colocado o nome atual da coleção que será renomeada. O parâmetro *target* é usado para o novo nome da coleção, e deve ser colocado entre aspas dupla. O exemplo utilizado para testar esse comando está ilustrado na Figura 66, cujo objetivo é renomear uma coleção “uf” para “estado”. Uma questão importante a respeito dos comandos são que os mesmos são *case sensitive*, assim como os seus parâmetros.

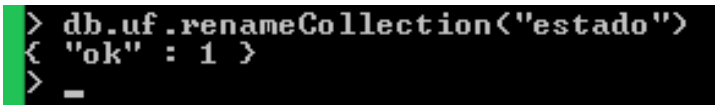

Figura 66 – Comando para renomear uma coleção
Fonte: Autoria própria

O comando *drop* é responsável por eliminar estruturas, e basicamente as estruturas do MongoDB é o próprio banco e suas coleções, assim o comando é mais utilizado para essas duas formas de exclusão. O campo *id* dentro das coleções não pode ser excluído ou alterado, o que pode ser feito é a manipulação do valor atribuído a ele. A sintaxe do comando *drop* mostrada na Figura 67 não possui argumentos, somente será colocado o nome da coleção.

```
dp.collection.drop()
```

Figura 67 – Sintaxe de exclusão de uma coleção
Fonte: Autoria própria

Como no teste anterior, o campo *collection* no código será substituído pelo nome do mesmo no banco de dados. Então para a implementação desse código, foi excluído uma coleção cujo nome é ‘colecao’. Podendo ser observado o teste na Figura 68, onde o retorno do comando é booleano para confirmar ou não a exclusão.

Figura 68 – Comando para exclusão de uma coleção
Fonte: Autoria própria

A partir dos comandos a seguir, pode-se notar uma diferença significativa com relação ao modelo relacional, mas também com os bancos de mesma tecnologia. Os campos inseridos dentro de uma coleção são chamados de documentos, e estes podem ser criados de diversas formas dentro de uma única coleção.

Para a inserção de dados é preciso também inserir os campos em que os dados são armazenados, pois cada documento pode ter campos únicos dentro de uma única coleção, por isso os campos são inseridos junto com os valores. Para a inserção de dados pode ser usado o comando *save* ou *insert*, possuindo o mesmo efeito, onde a sintaxe é mostrada na Figura 69.

db.collection.save(
<document>,
{
writeConcern: <document>
}
)
	db.collection.insert(
	<document or array of documents>,
	{
	writeConcern: <document>,
	ordered: <boolean>
	}
	)

Figura 69 – Sintaxe da inserção em uma coleção
Fonte: Autoria própria

A diferença da utilização da inserção com o comando *insert* é que o mesmo aceita a gravação de mais de um documento, enquanto o *save* aceita um documento por vez, em uma determinada coleção.

O *writeConcern* é opcional e seria uma programação que se preocupa se os dados vão ser de fato gravados de maneira correta, demorando um pouco mais no retorno da mensagem de sucesso da gravação. O *ordered* também é opcional e no comando *insert* serve para ordenar os documentos antes de sua gravação.

Para exemplificar a diferença dos dois comandos, foi realizado um teste com cada um deles, o resultado produzido pelo comando *save* é ilustrado na Figura 70, criando apenas um documento na coleção estado, porém na Figura 71 o comando *insert* é usado e cria três documentos na coleção cliente. Argumentos opcionais não foram utilizados nos testes.

```
> db.estado.save( { "uf" : "PR", "nome" : "Paraná" } )
WriteResult({ "nInserted" : 1 })
>
```

Figura 70 – Comando de inserção utilizando o save
Fonte: Autoria própria

Figura 71 – Comando de inserção utilizando o insert
Fonte: Autoria própria

As Figuras 70 e 71 ilustraram que os dados foram inseridos com sucesso em suas coleções. A coleção é equivalente a uma tabela no modelo relacional, mas no MongoDB nota-se que na inserção dos dados o usuário é livre para inserir qualquer campo dentro de um documento sem se preocupar com o tipo do valor do campo. Por esse fator os atributos são colocados apenas na inserção dos dados, pois um documento pode ter o campo nome e outro não é obrigado a ter o mesmo campo, ficando a critério do usuário essas escolhas.

O comando *update* neste SGBD, além de ter a função de atualização das informações contidas nas coleções, o mesmo pode estar associado ao comando *rename* permitindo renomear o campo de um documento. A Figura 72 mostra a sintaxe do comando utilizando o *rename* e a Figura 73 ilustra a sintaxe do *update*, no que diz respeito a alteração de valores.

```
db.collection.update(
        { <field> : value },
        { $rename: { <field1>: <newname1>, <field2>: <newname2>  ... }
        }
      )
```

Figura 72 – Sintaxe de renomeação de um ou mais campos
Fonte: Autoria própria

```
db.collection.update(
        <query>,
        <update>,                    {
          upsert: <boolean>,
          multi: <boolean>,
          writeConcern: <document>    }
      )
```

Figura 73 – Sintaxe de atualização de valores
Fonte: Autoria própria

O comando *update* vinculado ao *rename* necessita que o valor de um campo seja informado para que o banco possa buscar o local onde os campos do documento serão alterados.

Em seguida, o usuário deve informar os campos que serão alterados, e substituir as *tags* *<newname1>* e *<newname2>* pelos novos nomes dos campos.

O código da Figura 73 alterará os valores dos campos. Se o argumento opcional *upsert* for verdadeiro, o mesmo permite que seja criado um novo documento caso o que foi buscado não seja encontrado. O *multi* se verdadeiro atualiza mais de um documento, caso contrário, muda apenas um documento.

Para o teste com o comando que renomeia um campo existente em um documento, foi utilizada a coleção “estado”, onde a busca estava em cima de um documento que o campo “uf” correspondesse a “SC”, assim esse documento renomeou o campo “região” para “localização”, como ilustrado na Figura 74.

Se tratando da atualização de um ou mais valores, a Figura 75 mostra a mudança na coleção cliente, onde a busca foi através da empresa UTFPR, assim os documentos que tivessem como empresa a UTFPR, tiveram seus dados atualizados.

```
> db.estado.update( {uf: "SC"}, { $rename: { 'Região' : 'Localização' } } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

Figura 74 – Comando de alteração de campos
Fonte: Autoria própria

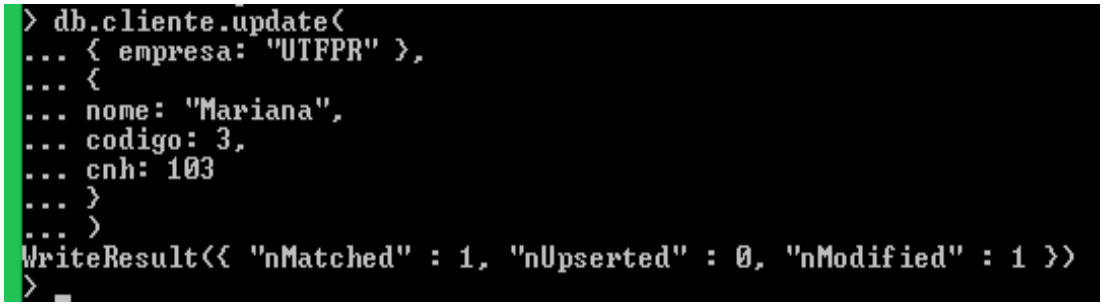

Figura 75 – Comando de alteração de valores e campos
Fonte: Autoria própria

Ainda se tratando dos comandos de manipulação de dados, tem-se o comando *remove*. O mesmo consegue remover todos os documentos de uma coleção. Além disso, pode buscar um determinado documento e removê-lo sem que haja necessidade de apagar todos os documentos da coleção, através de uma consulta específica no banco. A sintaxe para esse tipo de remoção pode ser observada na Figura 76.

```
db.collection.remove(
        <query>,
        {
          justOne: <boolean>,
          writeConcern: <document>
        } )
```

Figura 76 – Sintaxe de exclusão de documentos
Fonte: Autoria própria

O código mostra como é a estrutura básica do comando *remove*, a *tag* <*query*> como em outras implementações identifica onde o local para se informar a busca e o *justOne* como o *writeConcern* é um argumento opcional, onde o mesmo se verdadeiro se limitará a excluir apenas um documento, mas caso ele seja falso, apagará todos os documentos correspondentes ao critério de busca. A Figura 77 ilustra o uso do comando, onde há dois códigos, o primeiro removeu informações cujo nome era igual à “PR”, a segunda removeu todos os documentos da coleção “estado”.

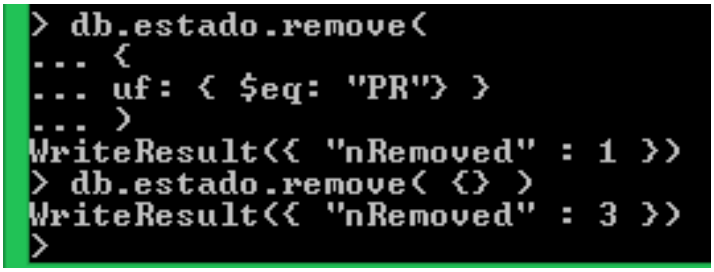

Figura 77 – Comando de remoção de documentos
Fonte: Autoria própria

Outro comando importante do MongoDB são as buscas que podem ser realizadas pelo SGBD, onde a sintaxe do comando é mostrado na Figura 78. O código *find* é responsável por engatilhar a busca.

```
db.collection.find(query,  projection)
```

Figura 78 – Sintaxe do comando find
Fonte: Autoria própria

O argumento *query* irá carregar as cláusulas para a busca, o qual o *projection* que é um argumento opcional, pode ser usado para retornar campos específicos de um documento, caso contrário, ele retornará todos os dados do documento. A fim de teste, se usou a coleção cliente, a qual a pesquisa teve a intenção de buscar um documento cujo campo nome fosse igual a Gustavo, e ao final do código foi colocada uma função chamada *pretty()*, sendo responsável por deixar o retorno da seleção mais organizado. Assim o teste está representado na Figura 79.

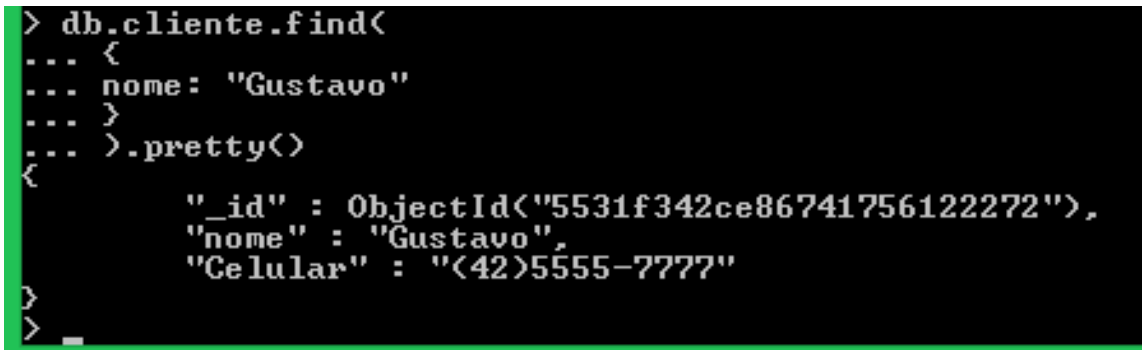

Figura 79 – Comando de seleção de documentos
Fonte: Autoria própria

Com relação ao uso de *triggers* e *procedures* como são implementadas no modelo relacional, segundo a documentação do MongoDB (MONGODB, 2015), o mesmo não suporta esses tipos de operações.

3.6 MODELO NOSQL – BERKELEYDB

Esta seção aborda o modelo de Chave-Valor da tecnologia NoSQL. Os comandos utilizados para criação, alteração, inserção, atualização e exclusão, são similares aos da linguagem SQL do modelo relacional, pois o SGBD utiliza uma biblioteca chamada SQLite, a qual é escrita em linguagem C (SQLITE, 2015). Assim, a biblioteca facilita o acesso aos arquivos que armazenam os registros do banco.

Para iniciar o processo de testes, é necessário saber que a base de dados ficará armazenada em um arquivo com extensão db. Foi utilizado o *prompt* de comandos do *Windows* para essa questão, onde o usuário deve escrever o nome da base de dados que deseja utilizar, porém caso a mesma não faça parte das bases armazenadas no SGBD, essa será criada automaticamente.

A sintaxe que permite a criação da base é descrita na Figura 80, e os comandos foram baseados na documentação oficial do *site* (BERKELEYDB, 2015).

```
C:\local_instalacao\bin> dbsql nomeBanco.db
```

Figura 80 – Sintaxe de criação do banco
Fonte: Autoria própria

Para conectar e/ou criar uma base de dados é necessário acessar o local onde foi instalado o SGBD, conforme ilustrado na Figura 84. Localizada a pasta *bin* é necessário executar o comando *dbsql* seguido do nome do banco de dados, não esquecendo de sua extensão db. Caso o banco de dados seja encontrado, o comando permitirá conectá-lo, caso o banco de dados não seja encontrado o mesmo será criado.

A Figura 81 ilustra como foi feita essa implementação, no qual o nome da base tem o nome de “teste”.

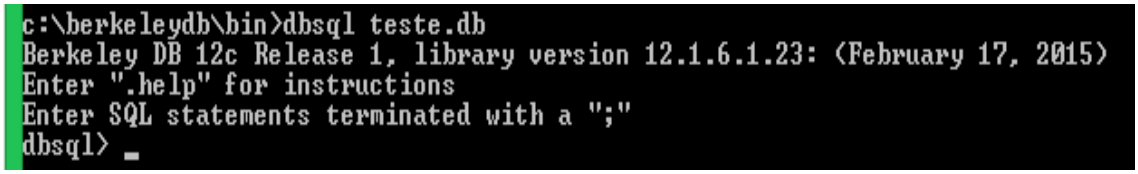

Figura 81 – Comando para criação ou alteração de um banco de dados
Fonte: Autoria própria

Com a utilização do terminal do *Windows*, a cada passo da execução, os códigos podem ou não conterem erros, e eventualmente o SGBD não mostra algumas mensagens de falhas, assim como também mensagem para confirmar se uma transação foi executada com sucesso. Para o próximo passo foi feita a criação de uma tabela, onde a sintaxe do comando pode ser observada na Figura 82.

```
CREATE TABLE <nome_tabela> (campo1 = valor1...);
```

Figura 82 – Sintaxe para criação de tabela
Fonte: Autoria própria

A tabela pode ser criada com qualquer nome e dentro da mesma podem ser inseridos um ou mais campos, sendo importante sempre utilizar o ponto e vírgula ao final do comando, para que este possa ser validado e executado pelo Berkeley. O comando de criação de uma tabela foi testando conforma apresentado na Figura 83.

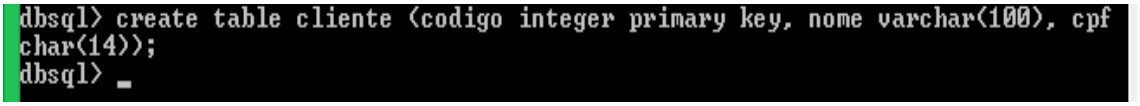

Figura 83 – Comando para criação de tabela
Fonte: Autoria própria

A tabela foi criada com apenas três campos, onde o primeiro é definido como chave primária e os outros dois como cadeia de caracteres. No BerkeleyDB quando o usuário coloca um campo como chave primária do tipo *integer*, o mesmo funcionará como auto incremento se nenhum valor for oferecido a ele. Para saber se a tabela foi mesmo criada, basta executar o comando *.tables*, assim será mostrada todas as tabelas desse banco.

Outro comando que permite a alteração na estrutura do banco é o comando *alter*, o mesmo tem funções semelhantes ao comando de mesmo

nome no modelo relacional. A sintaxe mostrada na Figura 84 é relacionada ao comando de inclusão de colunas na tabela.

```
ALTER TABLE <Nome_Tabela>  ADD COLUMN <Coluna> <Valor>...;
```

Figura 84 – Sintaxe de alteração para inclusão de coluna
Fonte: Autoria própria

Além de adicionar uma ou mais colunas, o comando de alteração de tabela pode ser usado para mudança do nome da tabela, exclusão e alguma coluna, caso seja usado o código *drop*, entre outros. Com o objetivo de testar o comando *alter*, a Figura 85 ilustra a implementação na inclusão de um campo novo na tabela “cliente”.

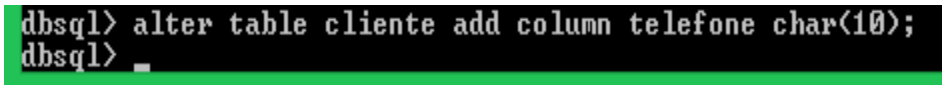

Figura 85 – Comando para alteração na tabela
Fonte: Autoria própria

O último comando que foi explorado relacionado a estrutura do banco, ou seja, a linguagem de definição de dados é o *drop*, já citado no parágrafo anterior, onde pode ser usado com mescla de outros comandos. Como o código pode ser usado para diversas funções e os mesmos são iguais aos do modelo relacional, a sintaxe visualizada na Figura 86 é usada para exclusão de uma tabela.

```
DROP TABLE <Nome_Tabela>;
```

Figura 86 – Sintaxe de exclusão de tabela
Fonte: Autoria própria

A implementação do comando não possui diferenças significativas para estudantes e profissionais que já tiveram contato com algum banco de dados relacional. Vale a ressalva que para o comando ter resultado positivo na sua execução é sempre necessário utilizar o símbolo do ponto e vírgula quando finalizado o comando, pois sem o mesmo nada será executado. A Figura 87 mostra o teste direto no banco de dados, onde foi excluída a tabela “estado” da base.

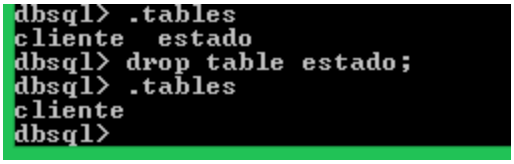

Figura 87 – Comando de exclusão de tabela
Fonte: Autoria própria

A fim de deixar clara a função do comando *drop*, usou-se o código *.tables* para demonstrar que antes do uso do *drop* na tabela “estado”, a mesma fazia parte da base de dados, e logo após a execução do comando a referida tabela foi excluída do banco.

Em seguida foram realizados testes com comandos que permitem realizar a manipulação de dados da base, sendo o primeiro comando utilizado o *insert*, onde nesta ocasião a finalidade é inserir registros nos campos da tabela. A sintaxe básica para o usuário realizar uma inserção, está descrita na Figura 88.

```
INSERT INTO <Nome_Tabela> (<Campos>) VALUES (<Valores>)
```

Figura 88 – Sintaxe de inserção de dados
Fonte: Autoria própria

Como o comando já foi visto de maneira semelhante no PostgreSQL, os atributos necessários para compor o código, que são nome da tabela, os campos e os valores que serão atribuídos a cada campo, não são necessários serem explicados com detalhes. Para a demonstração do teste foram adicionadas informações aos campos da tabela cliente, podendo o resultado ser observado na Figura 89.

Figura 89 – Comando de inserção de registros na tabela
Fonte: Autoria própria

A implementação mostra os valores inseridos nos campos “nome”, “CPF”, “RG” e “telefone”, embora haja mais um campo, o “código”. Como este foi criado sendo *integer primary key*, não há necessidade de atribuir um valor

ao mesmo, pois como descrito na demonstração do comando *create*, o “código” foi auto incrementado pelo próprio banco de dados.

O segundo comando abordado, pertencente a linguagem de manipulação, foi o *update*, mostrado na Figura 90, usado para atualização de dados em uma determinada tabela.

```
UPDATE <Nome_Tabela> SET <Campo> <Operador> <Novo_Valor>
        WHERE <Restrições>
```

Figura 90 – Sintaxe de atualização de dados
Fonte: Autoria própria

O comando *update* utiliza o nome da tabela onde o dado ou os dados serão alterados. O campo é usado para determinar com mais exatidão o local da atualização, e a *tag* do novo valor será usada para o usuário determinar o que vai ser inserido. Deve ser usado um operador de atribuição, como o sinal de igual ou o código *like*, por exemplo, e a palavra reservada *where* determina onde de fato os dados serão alterados. Essas restrições seguem o mesmo padrão do modelo relacional, no qual o teste utilizando o BerkeleyDB está sendo mostrado na Figura 91.

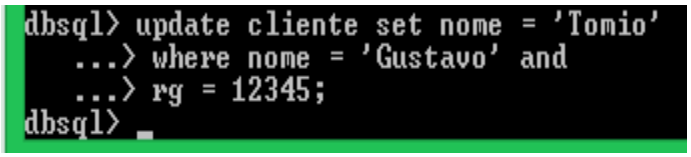

Figura 91 – Comando de atualização de registro
Fonte: Autoria própria

O exemplo mostrou a atualização do campo “nome” para “Tomio”, onde as restrições usadas foram o “nome” e o “RG”, assim como não apareceu nenhum tipo de erro, o processo foi realizado com sucesso pelo SGBD. O próximo comando experimentado foi o *delete*, o qual possui sintaxe conforme a Figura 92, servindo para a exclusão de elementos.

```
DELETE FROM <Nome_Tabela> WHERE <Restrições>
```

Figura 92 – Sintaxe de exclusão de dados
Fonte: Autoria própria

O comando *delete* possui dois parâmetros, que são descritos em outras explicações, o segundo retorna um valor booleano, para confirmar se as informações escritas realmente existem no banco e na tabela, assim a exclusão pode ser feita. Para o teste, como pode ser visto na Figura 93, a restrição diz respeito a um determinado código.

```
dbsql> delete from cliente where codigo = 4;
dbsql>
```

Figura 93 – Comando de exclusão de registro
Fonte: Autoria própria

Uma das funções mais utilizadas em um banco de dados é a de pesquisa, e no BerkeleyDB não é diferente. Como os comandos são semelhantes aos bancos do modelo relacional, apenas testes simples foram realizados. A sintaxe do comando está ilustrada na Figura 94.

```
SELECT <Instruções>

FROM <Nome_Tabelas>

WHERE <Restrições>
```

Figura 94 – Sintaxe de seleção
Fonte: Autoria própria

O primeiro teste do comando *select*, não aplicou nenhuma restrição de tupla, na qual a *tag where* não é obrigatória, porém no segundo teste exemplifica o uso de toda a estrutura do código. A Figura 95 está ilustrando o que foi escrito nos testes, onde o resultado dos mesmos aparece logo abaixo da solicitação do comando.

```
dbsql> select * from cliente;
1¦Tomio¦000.111.333-44¦12345¦4299991010
2¦Mariana¦999.111.555-00¦11112¦4299107733
3¦Marcos¦456.789.101-12¦55995¦4288553020
dbsql> select nome, cpf from cliente where telefone = '4299107733';
Mariana¦999.111.555-00
dbsql>
```

Figura 95 – Comandos de seleção
Fonte: Autoria própria

Foi visto que *triggers* e *procedures* são mais complicadas e difíceis de serem encontradas nos bancos de dados da tecnologia NoSQL. Seguindo o padrão de outros bancos, o BerkeleyDB diz estar trabalhando para tornar possível esses recursos, principalmente das *triggers*, pois no momento não há suporte para esse tipo de aplicação considerada mais complexa, porém de utilidade ímpar para bancos do modelo relacional.

3.7 CONSIDERAÇÕES DO CAPÍTULO

Após os testes nos bancos de dados escolhidos, notou-se que mesmo se tratando de uma mesma tecnologia, no caso o NoSQL, a diversas distinções e particularidades dentro de cada modelo apresentado. Onde as linguagens das ferramentas Cassandra e BerkeleyDB possuem semelhança com a linguagem SQL, usada no modelo relacional, mas não se pode afirmar que os SGBD's trabalham da mesma forma por possuírem essas similaridades.

Os bancos de dados MongoDB e principalmente o Neo4j, se distinguem de uma forma mais evidente do modelo relacional em relação a linguagem que utilizam para realizar as operações básicas de um SGBD. Outro ponto a ser observado é a forma de armazenamento de cada um dos modelos estudados, pois são diferentes entre eles, ficando difícil avaliar, por exemplo, qual seria o melhor em desempenho de escrita e leitura.

Fazendo uma diferenciação dos comandos NoSQL, o qual se baseia nas implementações dos comandos do modelo relacional, representado pelo PostgreSQL. A diferenciação foi feita levando em consideração os quatro SGBD's escolhidos de cada um dos modelos NoSQL, o Quadro 3 demonstra essa diferença entre os modelos.

NoSQL/Comandos	Neo4j	Cassandra	MongoDB	BerkeleyDB
Criação da Base	Browse (interface)	Create keyspace	Use	Dbsql
Criação de Tabela	Create ()	Create columnfamily	Db.createCollection	Create table
Alteração de Tabela	Match set	Alter columnfamily	Db.renameCollection	Alter table
Exclusão de Tabela	Match delete	Drop Columnfamily	Db.drop	Drop table

Inserção de Dados	create ({ })	Insert into	Db.save/insert	Insert into
Alteração de Dados	Match set	Update set	Db.update	Update set
Deleção de Dados	Match remove	Delete from	Db.remove	Delete from
Seleção de Dados	Match return	Select from	Db.find	Select from

Quadro 3: Diferenciação dos comandos NoSQL
Fonte: Autoria própria

Como pode ser observado no Quadro 3, os bancos Cassandra e BerkeleyDB tem comandos semelhantes aos utilizados no PostgreSQL, e nos bancos MongoDB e Neo4j pode-se perceber que são distintos de um banco de dados relacional.

4 CONCLUSÃO

Este Capítulo está divido em duas seções. A seção 4.1 explica os resultados adquiridos com a realização da pesquisa, ou seja, destaca se os objetivos traçados foram alcançados. A seção 4.2 apresenta os trabalhos futuros, em que o foco principal é mostrar que a partir da pesquisa desenvolvida, outros temas relacionados podem ser desenvolvidos.

4.1 RESULTADOS

O tema do trabalho foi desafiador, pelo fato de ser uma área relativamente nova em franco desenvolvimento, porém pouca abordada na graduação. Esses motivos levaram a um maior interesse da tecnologia, onde o principal objetivo foi estudar as ferramentas disponíveis dos quatro modelos propostos pela tecnologia NoSQL, identificar as diferenças e semelhanças com o modelo relacional e experimentar os comandos de criação dos elementos dos SGBD's assim como os comandos que permitem a manipulação de dados da base, apresentando sempre a sintaxe básica destes.

Para compreender essa tecnologia e como tem sido evoluída no mercado, foi necessário entender a divisão dos modelos existentes, ou seja, foi realizado um levantamento distinguindo os modelos de Grafos, Família de Colunas, Documentos e Chave-Valor, identificando seus propósitos e características.

De posse dessas características, iniciou-se um estudo no sentido de identificar as ferramentas disponíveis da tecnologia NoSQL. Nessa etapa foi encontrado muito material, o que possibilitou obter maior conhecimento sobre o assunto abordado.

Com os modelos devidamente compreendidos, buscou-se um SGBD de cada um dos modelos para a realização dos testes práticos, e um dos critérios principais para essa definição foi que a ferramenta fosse considerada "pura", ou seja, que implementasse somente um modelo. Depois da definição das ferramentas, as mesmas foram instaladas, porém uma delas apresentou problemas nesse processo, tanto no sistema operacional Windows quanto no

Linux, e isso se constatou pelo fato da arquitetura do computador usado para a instalação dos SGBD's, não ser compatível com os requisitos ideais para a utilização da ferramenta Riak, sendo necessária uma mudança no banco, onde o Riak foi substituído pelo BerkeleyDB, mantendo as características para escolha de ferramentas.

Na realização dos testes práticos, as documentações oficiais de cada um dos bancos de dados estudados, foram utilizadas, pois as mesmas acrescentaram positivamente no aprendizado das ferramentas. Por se tratar de modelos distintos, os testes foram realizados nos comandos mais comuns de cada SGBD, onde a intenção foi exemplificar a utilização dos principais comandos disponíveis. Decidiu-se por esse formato de experimento visto que o uso de uma única base talvez não pudesse ser implementada integralmente em todos os SGBD's pesquisados, devido a diversidade apresentada nas ferramentas.

O banco de dados Cassandra do modelo Família de Colunas se mostrou com diversas semelhanças com o modelo relacional, onde não houveram muitas dificuldades para a implementação dos testes devido a linguagem CQL ser muito similar a SQL. O SGBD possui algumas particularidades interessantes, uma delas é o fato de que em uma busca somente pode ser pesquisado valores que sejam chave primária, caso contrário um erro é exibido.

No modelo de Documentos, o MongoDB se mostrou um SGBD com muitos recursos, e entre os bancos testados, parece ser o mais flexível, sua linguagem é intuitiva para o usuário. Possui uma peculiaridade, pois caso seja pedido para se inserir um registro em uma coleção que não exista, o banco a criará automaticamente, inserindo em seguida o registro desejado.

O Neo4j do modelo de Grafos é um banco de dados com diversos recursos também, e demonstra ser o mais diferente dos quatro modelos, devido a sua estruturação estar elaborada em cima de vértices e arestas. Além de ser diferenciado dos demais, o Neo4j por fugir da experiência do usuário em bancos relacionais.

Em se tratando do banco de dados BerkeleyDB, do modelo Chave-Valor, o mesmo também, como o Cassandra, possui similaridades com o modelo relacional em relação a sua linguagem. O mesmo tem uma função

interessante quando se trata de chave primária, onde caso exista um campo na tabela declarado como *primary key* e tipo *integer*, este campo se auto-incrementará quando nenhum valor for informado para o mesmo.

Pode-se afirmar que as implementações realizadas permitiram identificar as particularidades dos bancos da tecnologia NoSQL entre si, e também quando se tratou da diferenciação dos mesmos com o PostgreSQL, do modelo relacional. Assim, podem-se observar nos experimentos algumas semelhanças entre os bancos e as particularidades de cada um, não cabendo julgar se uma característica é boa ou ruim.

Com a diferenciação das ferramentas utilizadas, foi possível mostrar algumas restrições apresentadas pelos SGBD's testados e principalmente, ao modelo relacional. Benefícios também foram identificados e levantados, pois cada SGBD possui os seus próprios. Não foi possível abordar maiores detalhes que cada ferramenta pode oferecer, pois são muitas funções, comandos e características, mas acredita-se que as que foram testadas são relevantes para o entendimento e uso da ferramenta.

De forma geral, a pesquisa levantou diversos questionamentos e um interesse maior para outros estudos nessa tecnologia de banco de dados apresentada. O conhecimento adquirido é importante para que se continue buscando novas formas de gerenciamento de banco de dados, mostrando a riqueza dessa área em termos de conteúdo.

4.2 TRABALHOS FUTUROS

Terminada a pesquisa, e analisando o que foi obtido, pode-se traçar outros objetivos relacionados à tecnologia NoSQL, no qual o trabalho pode inspirar outros docentes e discentes a continuarem pesquisando sobre o assunto. Em relação ao tema proposto, o mesmo é um início para pesquisas relacionadas a bancos de dados não convencionais no câmpus da instituição em questão.

Algumas sugestões para trabalhos futuros seriam:

- Estudar mais a fundo cada ferramenta apresentada, pesquisando as demais funcionalidades e comandos disponíveis nas ferramentas experimentadas neste trabalho;
- Pesquisar outros bancos de dados disponíveis da tecnologia NoSQL ou até fazer um estudo sobre a performance de cada banco em relação ao tempo de implementação dos comandos;
- Utilizar alguns desses SGBD's e implementar um sistema real neles, fazendo assim uma análise de comportamento quando se usa uma aplicação mais elaborada, comparando com o modelo relacional, que ainda é um dos líderes quando se trata de banco de dados, tanto na área acadêmica quanto comercialmente.

Além dos possíveis trabalhos futuros levantados anteriormente, outras pesquisas também podem ser desenvolvidas nessa área, pois a mesma é bastante abrangente, onde empresas de grande porte estão utilizando esses tipos de banco de dados, tornando-se viável a continuação dos estudos sobre a tecnologia NoSQL.

REFERÊNCIAS

ALMEIDA, A. **Neo4j na Prática:** Como Entrar no Mundo dos Bancos de Dados de Grafo. Disponível em: http://www.univale.com.br/unisite/mundo-j/artigos/51Neo4j.pdf. Acesso em: 31 out. 2014.

ALMEIDA, R. C. de; BRITO, P. F. de. **Utilização da Classe de Banco de Dados NOSQL como Solução para Manipulação de Diversas Estruturas de Dados**. Disponível em: http://www.bandalerda.com.br/wp-content/uploads/2012/10/Utilizacao_da_Classe_de_Banco_de_-Dados_NOSQL_como_Solucao_para_Manipulacao_de_Diversas_Estruturas_de_Dados.pdf. Acesso em: 03 nov. 2014.

ANICETO, R. C; XAVIER, R. F. **Um Estudo Sobre a Utilização do Banco de Dados NoSQL Cassandra em Dados Biológicos**. Disponível em: http://bdm.unb.br/bitstream/10483/7927/1/2014_RodrigoCardosoAniceto_ReneFreireXavier.pdf. Acesso em: 07 nov. 2014.

BALLEM, M. **Salvando Arquivos no MongoDB com GridFS**. Disponível em: http://www.mballem.com/post/salvando-arquivos-no-mongodb-com-gridfs/. Acesso em: 19 nov. 2014.

BASHO. **Riak NoSQL Database**. Disponível em: http://basho.com/products/#riak. Acesso em: 25 out. 2014.

BERKELEYDB. **Oracle BerkeleyDB 12c *Release* 1**. Disponível em: http://docs.oracle.com/cd/E17076_04/html/index.html. Acesso em: 04 mai. 2015.

BIANCO, G. D. **Banco de Dados em Memória sobre Clusters de Computadores**. Disponível em: http://ncc.furg.br/publi/TCC_bancos_de_dados_em_memoria_sobre_clusters_de_computadores.pdf. Acesso em: 01 dez. 2014.

BOSCARIOLI, C; SOARES, B. E. **Modelo de Banco de Dados Colunar**: Características, Aplicações e Exemplos de Sistemas. Disponível em: http://www.lbd.dcc.ufmg.br/colecoes/erbd/2013/007.pdf. Acesso em: 08 fev. 2015.

BOSTIC, K; OLSON, M. A; SELTZER, M. **BerkeleyDB**. Disponível em: http://www.eecs.harvard.edu/margo/papers/freenix99/paper.pdf. Acesso em: 02 dez. 2014.

BRITO, R. W. **Banco de Dados NoSQL x SGBDs Relacionais**: Análise Comparativa. Disponível em: http://www.infobrasil.inf.br/userfiles/27-05-S4-1-68840-Bancos%20de%20Dados%20NoSQL.pdf. Acesso em: 03 fev. 2015.

BURD, G. **Oracle Berkeley DB – *Data Storage***. Disponível em: http://pt.slideshare.net/gregburd/oracle-berkeley-db-data-storage-ds-tutorial. Acesso em: 09 mar. 2015.

CASSANDRA. ***Cassandra Query Language* (CQL) v3.2.0**. Disponível em: https://cassandra.apache.org/doc/cql3/CQL.html. Acesso em: 18 mar. 2015.

DA SILVA, D. A. P; ETSCHEID, D. M. **Um Estudo Sobre a Persistência Poliglota e sua aplicação em uma plataforma *web***. Disponível em: http://www.fateclins.edu.br/site/trabalhoGraduacao/mXNlfr3uCOo7R3PFwnQzyvyQkx0os4rTIV5CC.pdf. Acesso em: 01 nov. 2014.

DATASTAX. **What's New in Cassandra 2.0: Prototype Triggers Support**. Disponível em: ttp://www.datastax.com/dev/whats-new-in-cassandra-2-0-prototype-triggers-support. Acesso em: 20 mar. 2015.

DIANA, M. de; GEROSA, M. A. **NOSQL na Web 2.0**: Um Estudo Comparativo de Bancos Não-Relacionais para Armazenamento de Dados na Web 2.0. Disponível em: http://www.academia.edu/957469/NOSQL_na_Web_2.0_Um_Estudo_Comparativo_de_Bancos_N%C3%A3o-

Relacionais_para_Armazenamento_de_Dados_na_Web_2.0. Acesso em: 17 nov. 2014.

DIEGUES, N; ORAZOV, M; PAIVA, J; RODRIGUES, L; ROMANO, P. **Auto-Configuração de Bases de dados NoSQL Multi-Dimensionais**. Disponível em: http://www.gsd.inesc-id.pt/~jgpaiva/pubs/inforum13.pdf. Acesso em: 01 dez. 2014.

FAGUNDES, P. **MongoDB Versus Cassandra**. Disponível em: http://mongodbwise.wordpress.com/2014/05/06/mongodb-vs-cassandra/. Acesso em: 17 nov. 2014.

FANTINATO, M; PRADO, E. P. V; SUN, V; SOUZA, A. M. de. **Critérios para Seleção de SGBD NoSQL:** o Ponto de Vista de Especialistas com Base na Literatura. Disponível em: http://www.lbd.dcc.ufmg.br/colecoes/sbsi/2014/0012.pdf. Acesso em: 28 nov. 2014.

FARIA, A. de O. **Apache Cassandra NoSQL, uma tecnologia emergente**. Disponível em: http://www.linhadecodigo.com.br/artigo/2840/apache-cassandra-nosql-uma-tecnologia-emergente.aspx. Acesso em: 03 nov. 2014.

FOWLER, M; SALADALE, P. J. **NoSQL Um Guia Conciso para o Mundo Emergente de Persistência Poliglota Essencial**. São Paulo: Novatec, 2013.

FRIESS, I, I. **Análise de Bancos de Dados NoSQL e Desenvolvimento de uma Aplicação**. Disponível em: www-app.inf.ufsm.br/bdtg/arquivo.php?id=171&download=1. Acesso em: 01 nov. 2014.

HYPERGRAPHDB. **HyperGraphDB *Documentation***. Disponível em: http://www.hypergraphdb.org/learn. Acesso em: 30 out. 2014.

INFOGRID. ***The Web Graph Database***. Disponível em: http://infogrid.org/trac/. Acesso em: 30 out. 2014.

IANNI, V. **Introdução aos Bancos de Dados NoSQL**. Disponível em: - http://www.devmedia.com.br/introducao-aos-bancos-de-dados-nosql/26044. Acesso em: 25 out. 2014.

JUNIOR, N. C. de L. **Banco de Dados NoSQL**. Disponível em: http://www.codate.com.br/2014/03/18/banco-de-dados-nosql/. Acesso em: 30 nov. 2014.

LENNON, J. **Explore o MongoDB**. Disponível em: http://www.ibm.com/developerworks/br/library/os-mongodb4/. Acesso em: 20 nov. 2014.

LINUX MAGAZINE. **Base de Dados Para Estatística Vai Parar na Nuvem**. Disponível em: http://ns1.linuxmag.com.br/lm/noticia/base_de_dados_para_estatistica_vai_para_a_nuvem. Acesso em: 25 out. 2014.

LÓSCIO, B. F; OLIVEIRA, H. R de; PONTES, J. C. de S. **NoSQL no Desenvolvimento de Aplicações Web colaborativas**. Disponível em: http://www.addlabs.uff.br/sbsc_site/SBSC2011_NoSQL.pdf. Acesso em: 20 nov. 2014.

MACEDO, J. A. **Graph Database**: Soluções na Literatura e Implementações Disponíveis. Disponível em: http://jamacedo.com/2011/07/graph-database/. Acesso em: 30 out. 2014.

MACÊDO, J. A. F. de; MACHADO, J. C; MOREIRA, L. O; SOUSA, F. R. C. **Gerenciamento de Dados em Nuvem:** Conceitos, Sistemas e Desafios. Disponível em: http://200.17.137.109:8081/novobsi/Members/josino/fundamentos-de-banco-de-dados/2012.1/Gerenciamento_Dados_Nuvem.pdf. Acesso em: 10 nov. 2014.

MONGODB. ***The MongoDB 3.0 Manual***. Disponível em: http://docs.mongodb.org/manual/. Acesso em: 15 abr. 2015.

MULLER, T. A. ***Neo4j Cloud Deployment com Spring Data***. Disponível em: http://www.univale.com.br/unisite/mundo-j/artigos/55_Neo4j.pdf. Acesso em: 27 out. 2014.

NEO4J. ***Basic friend finding based on social neighborhood***. Disponível em: http://neo4j.com/docs/stable/cypher-cookbook-friend-finding.html. Acesso em: 09 mar. 2015.

NEO4J. ***How*** **Neo4j** ***HA Operates***. Disponível em: http://neo4j.com/docs/stable/ha-how.html. Acesso em: 31 out. 2014.

NEO4J. ***The*** **Neo4j** ***Manual v2.3.0-M01:*** *Cypher Query Language.* Disponível em: http://neo4j.com/docs/milestone/cypher-query-lang.html. Acesso em: 28 mar. 2015.

NEO4J. ***Transaction***. Disponível em: http://neo4j.com/docs/stable/query-transactions.html. Acesso em: 31 out. 2014.

OBJECTIVITY. **InfiniteGraph**. Disponível em: http://www.objectivity.com/products/infinitegraph/. Acesso em: 30 out. 2014.

ORACLE. **Oracle BerkeleyDB:** *Programmer's Reference Guide*. Disponível em: http://docs.oracle.com/cd/E17076_04/html/programmer_reference/BDB_Prog_Reference.pdf. Acesso em: 01 dez. 2014.

PASQUALINI. T. da S. **Cassandra:** Um sistema de armazenamento NoSQL altamente escalável. Disponível em: http://www.dcomp.sor.ufscar.br/verdi/topicosCloud/ApresCassandra.pdf. Acesso em: 02 nov. 2014.

PCMAG. ***A Definition of*:** BerkeleyDB. Disponível em: http://www.pcmag.com/encyclopedia/term/38551/berkeley-db. Acesso em: 02 dez. 2014.

PEREIRA, V. H. P. **Segurança e Privacidade de Dados em Nuvens de Armazenamento**. Disponível em: http://run.unl.pt/bitstream/10362/13104/1/Pereira_2014.pdf. Acesso em: 03 nov. 2014

PIGA, L. **Capítulo 5:** Quando Usar MongoDB. Disponível em: http://leandropiga.nothus.com.br/2013/09/capitulo-5-quando-usar-mongodb.html. Acesso em: 20 nov. 2014.

POSTGRESQL. **PostgreSQL 9.4.1 *Documentation*:** *The PostgreSQL Global Development Group.* Disponível em: http://www.postgresql.org/docs/9.4/static/. Acesso em: 15 fev. 2015.

SANTANA, O. G. de. **Desenvolvendo com NOSQL**. Disponível em: http://www.devmedia.com.br/desenvolvendo-com-nosql-cassandra-em-java-conhecendo-cassandra/23203. Acesso em: 02 nov. 2014.

SQLITE. ***About the* SQLITE**. Disponível em: https://www.sqlite.org/about.html. Acesso em: 05 mai. 2015.

WEISSMANN, H. L. **Coisas que Não Te Contam Sobre MongoDB**. Disponível em: http://www.itexto.net/devkico/?p=1621. Acesso em: 17 nov. 2014.

YADAVA, H. ***The BerkeleyDB Book***. New York: Apress, 2007.

APÊNDICE A - Instalação do Neo4j

O *software* é desenvolvido pela empresa *Neo Technology*, a qual possui os direitos de distribuição do mesmo, onde há uma versão baseada nos termos GNU e outra versão chamada *Enterprise* que é comercializada pelo próprio desenvolvedor. Para se obter o programa, basta se conectar à *internet* e visitar a página oficial do Neo4j[1].

No canto superior da página principal há um *link* em azul denominado *Download* Neo4j, o qual deverá ser acessado para entrar na área de *downloads*, onde haverá duas opções de versões, a *Community* e a *Enterprise*. Deve-se optar pelo *link Download Community Edition*, que ao ser clicado iniciará automaticamente o *download*.

Após concluída a etapa anterior, será obtido o produto Neo4j *Community* com a versão 2.1.5 para *Windows*, onde esses passos a seguir podem variar dependendo da versão do programa e da plataforma. O próximo passo é fazer a instalação desse arquivo. Com um duplo clique no instalador, começará a segunda etapa, a qual abrirá a seguinte imagem, mostrada na Figura 96 e deve-se clicar o botão *next*.

Figura 96 - Tela inicial de instalação
Fonte: Autoria própria

[1] A ferramenta é encontrada no endereço neo4j.com.

Na Figura 97, aparecerá o termo de licença do produto, onde para se continuar é necessário preencher o campo que aceita o contrato descrito para habilitar o botão *next*, clicar no mesmo e seguir para a próxima tela.

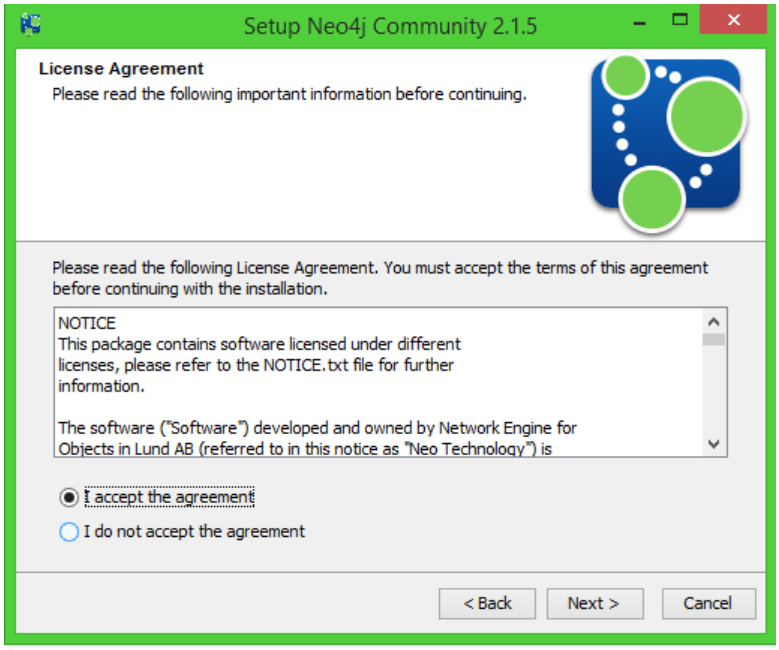

Figura 97 - Termo de Licença
Fonte: Autoria própria

A próxima indagação será a respeito de onde o usuário deseja fixar a instalação do aplicativo, ficando a seu critério o local. A Figura 98 mostra um exemplo de caminho escolhido. Em seguida deve-se pressionar novamente o botão *next*.

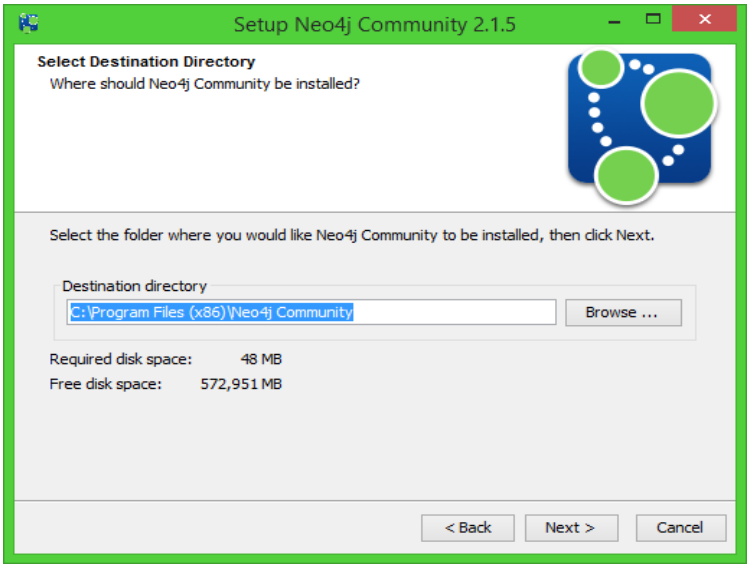

Figura 98 - Escolha do local de instalação
Fonte: Autoria própria

A seguir, o usuário terá a opção de escolher o nome que irá compor a pasta onde ficarão os arquivos instalados pelo programa. Também há um campo para criar uma pasta no menu inicial e outro para criação de atalho, sendo opcional, o qual pode ser visto na Figura 99. Decidido essas informações, clica-se para prosseguir.

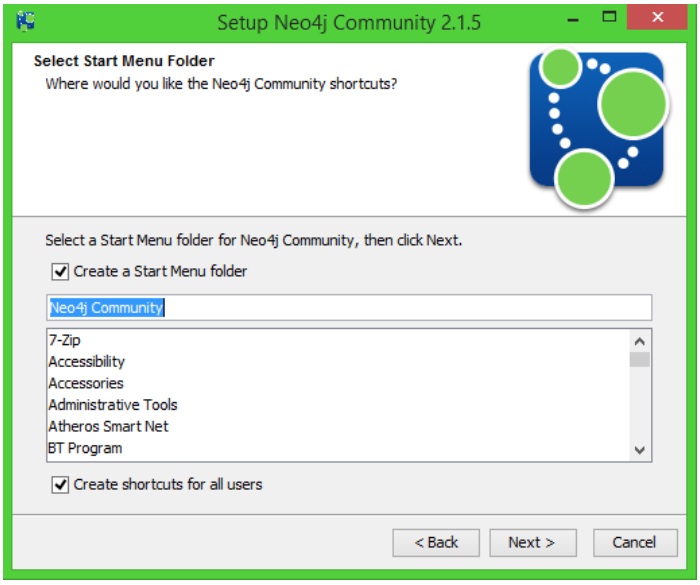

Figura 99 - Escolha do nome do programa
Fonte: Autoria própria

Finalizada essas etapas, o usuário deve aguardar enquanto os arquivos são instalados automaticamente, o que pode ser visualizado na Figura 100.

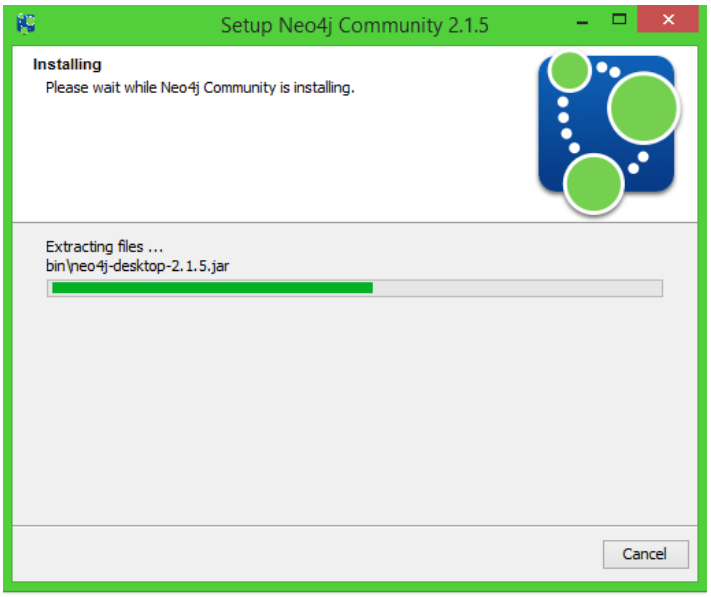

Figura 100 - Processo de instalação
Fonte: Autoria própria

Depois de instalado, a última tela a ser vista pelo usuário é a Figura 101, onde será finalizada a instalação, tendo a opção de finalizar e rodar o programa ou somente terminar.

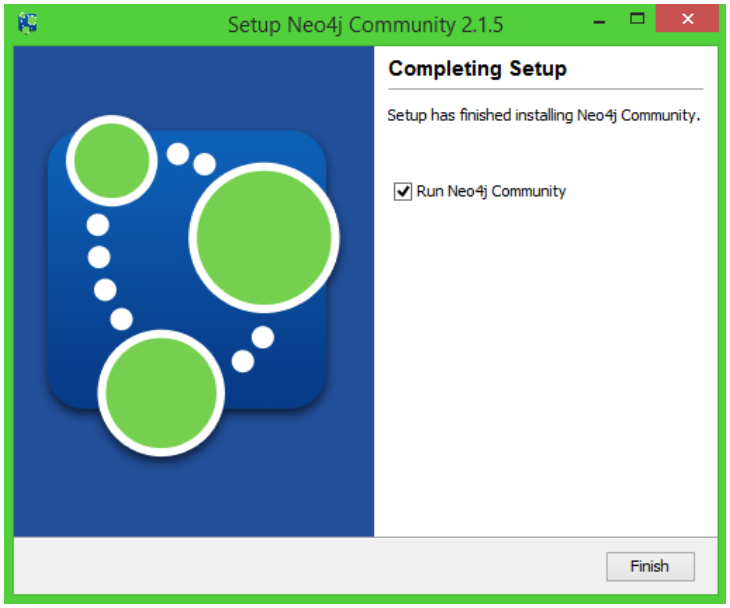

Figura 101 - Término da instalação
Fonte: Autoria própria

APÊNDICE B - Instalação do Cassandra

O Cassandra[2] é um gerenciador de banco de dados criado pelo Facebook e um ano depois passado a empresa *Apache Software Foundation*, porém possui colaboração de outras empresas. A partir de 2008 os criadores resolveram liberar seu código fonte, sendo atualmente um *software* livre.

O *software* será instalado no sistema operacional *Windows.* No *site* para *download* há vários tipos de arquivos, foi escolhida a versão recomendada pelo *site* devido sua estabilidade de acordo com as informações fornecidas no *site*. A versão do programa escolhida para o processo de instalação é a 2.0.11, onde o modo de instalação pode variar dependendo da versão do *software* e a plataforma escolhida. Assim é necessário clicar no *link* para o *download* começar, levando em consideração a arquitetura do sistema operacional do usuário.

Após baixado o arquivo, o próximo passo é seguir com a efetiva instalação do programa, o qual deve-se clicar duas vezes para abrir o mesmo. A Figura 102 representa a imagem que deverá aparecer na tela, iniciando o processo clicando no botão *next*.

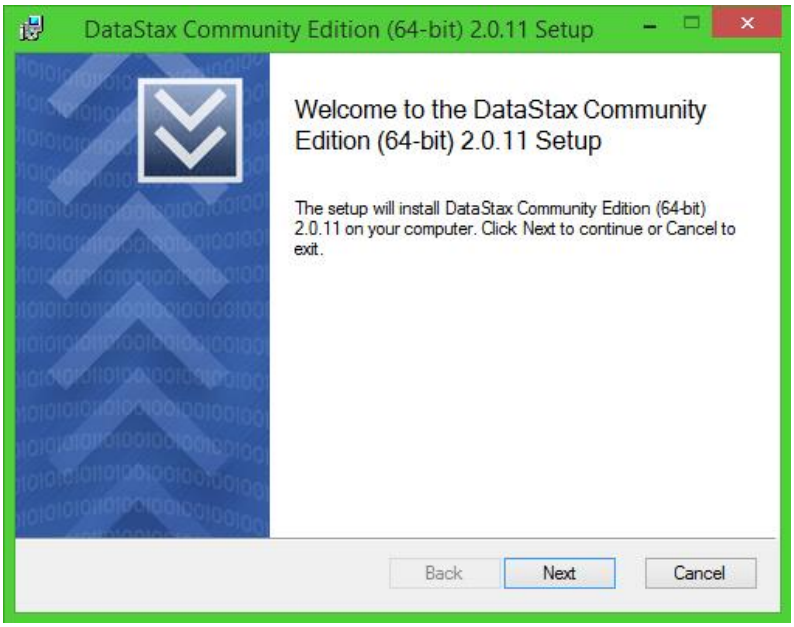

Figura 102 - Tela inicial de instalação
Fonte: Autoria própria

A página a seguir mostrará o termo de licença do *software*, o qual para se dar prosseguimento é necessário aceitar esse termo preenchendo a caixa

[2] O *site* oficial para *download* é planetcassandra.org/cassandra.

de diálogo, vista na Figura 103, assim o botão *next* será habilitado e poderá ser pressionado, dando continuidade ao processo de instalação.

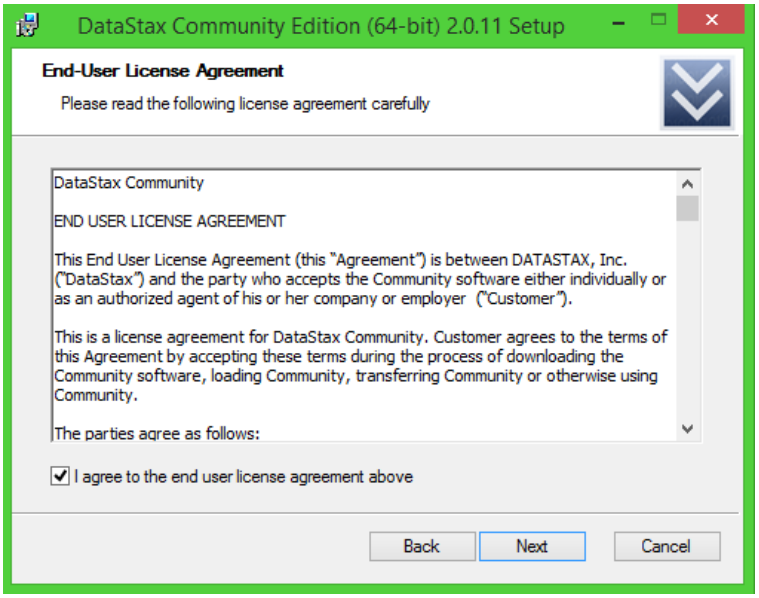

Figura 103 - Termo de Licença
Fonte: Autoria própria

Na Figura 104, é possível escolher onde em seu microcomputador ficará os arquivos instalados, ficando a critério do usuário. Decidido o local, pressiona-se *next* novamente.

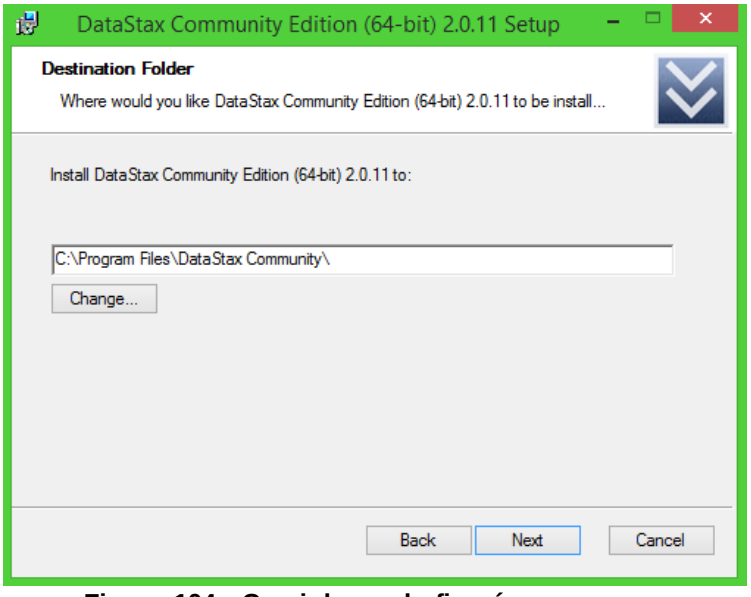

Figura 104 - Caminho onde ficará o programa
Fonte: Autoria própria

A próxima tela, ilustrada na Figura 105, tem por objetivo representar duas opções de cunho opcional, onde as caixas de diálogo devem ser preenchidas caso se deseje, iniciar os dois serviços acoplados ao Cassandra automaticamente. Para prosseguir, clique *next*.

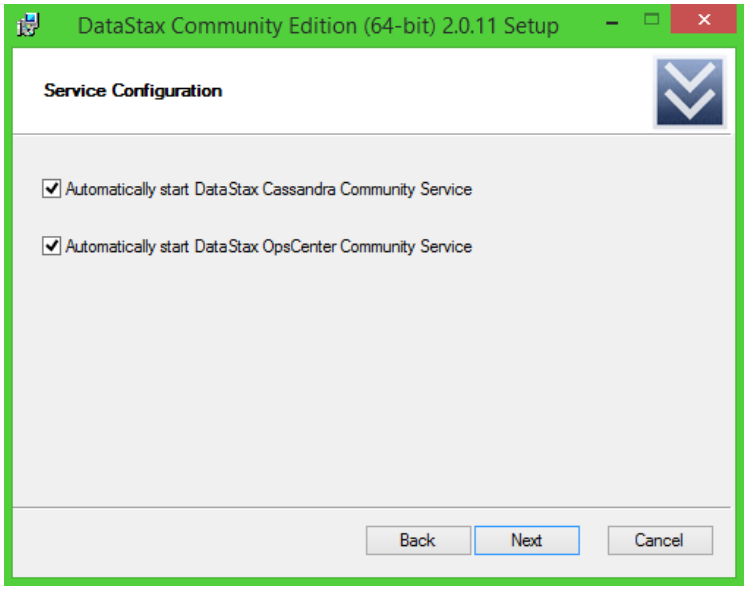

Figura 105 - Serviços a serem executados automaticamente
Fonte: Autoria própria

A Figura 106 é o último passo antes da instalação automática se iniciar, e para continuar, basta clicar no botão *Install*.

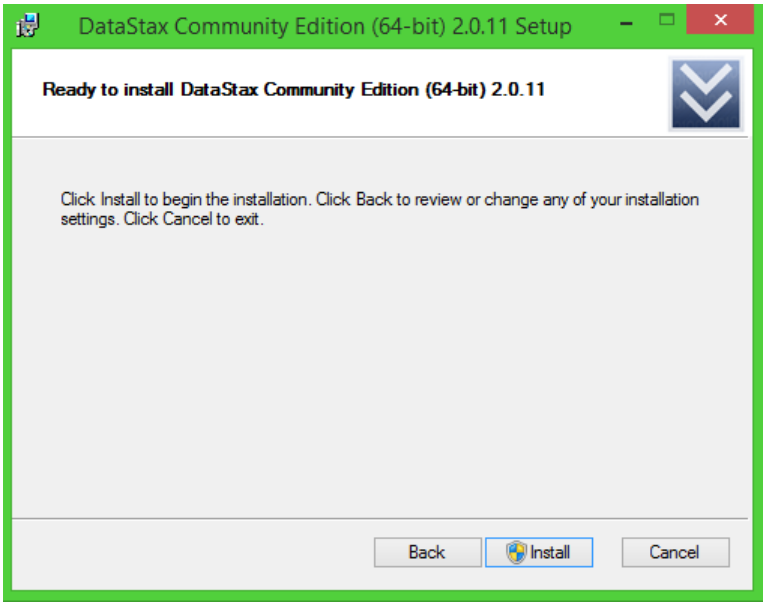

Figura 106 - Iniciar a efetivamente a instalação
Fonte: Autoria própria

Logo em seguida, deve-se aguardar enquanto o programa é instalado, conforme mostra a Figura 107.

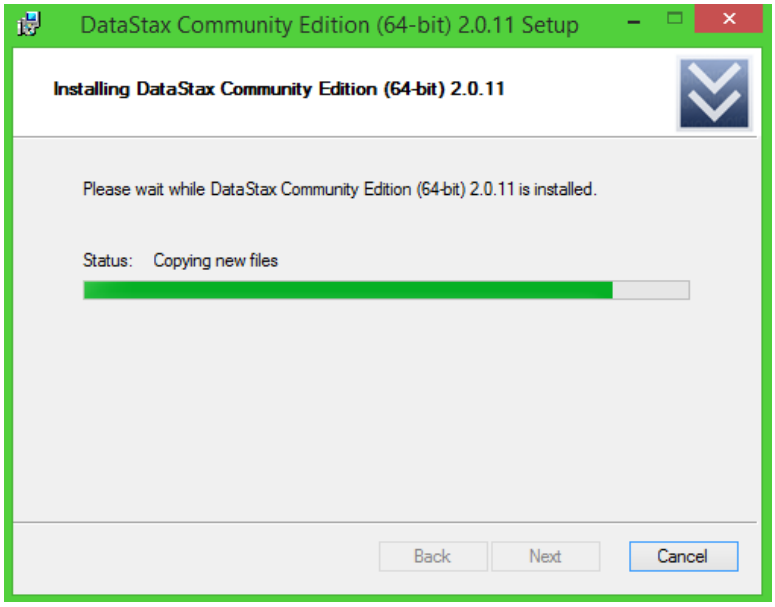

Figura 107 - Acompanhamento da instalação
Fonte: Autoria própria

Para terminar o processo e concluir com sucesso, a última tela a ser vista é retratada na Figura 108, a qual há duas caixas de diálogo, onde o usuário poderá escolher em iniciar a parte gráfica do programa e a outra diz respeito do registro do banco de dados. Para terminar deve-se clicar no *Finish*.

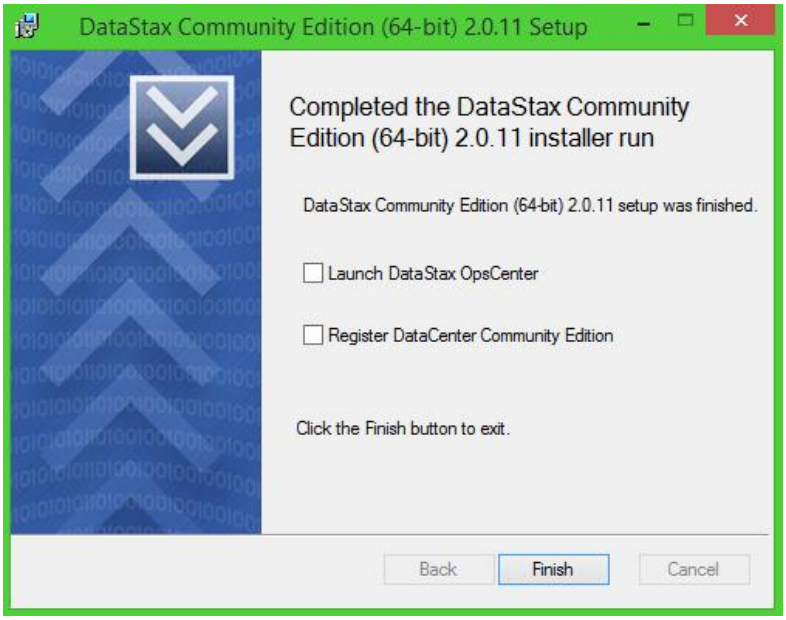

Figura 108 - Finalizando o processo
Fonte: Autoria própria

APÊNDICE C - Instalação do MongoDB

O banco de dados não convencional MongoDB[3] é desenvolvido pela empresa formalmente chamada de 10gen, também conhecida como MongoDB Inc. Segue os conceitos GNU, dando mais liberdade a seus utilizadores. A versão que será feito o processo de instalação será a 2.6.5 para *Windows*, onde poderá os passos poderão variar dependendo da versão ou plataforma escolhida.

Acessada a página de *downloads* do MongoDB, o usuário terá várias opções e versões para baixar, levando em conta a versão do *software* e o sistema operacional que o mesmo utiliza, no qual o *link* escolhido dará início automático do *download*.

Terminado o processo da obtenção do arquivo, deve-se dar um duplo clique no mesmo para se iniciar os procedimentos de instalação. Após isso, aparecerá na tela uma imagem semelhante à Figura 109, representando o começo do processo, sendo necessário clicar no botão *next* para prosseguir.

Figura 109 - Tela de início do programa
Fonte: Autoria própria

Com relação a Figura 110, ela é a próxima etapa, a qual mostra o termo de licença do *software* e sendo assim, deve ser aceito selecionando a caixa de diálogo para desbloquear o botão *next* e poder dar prosseguimento a instalação da forma correta.

[3] Site para *download* do MongoDB é o www.mongodb.org/downloads.

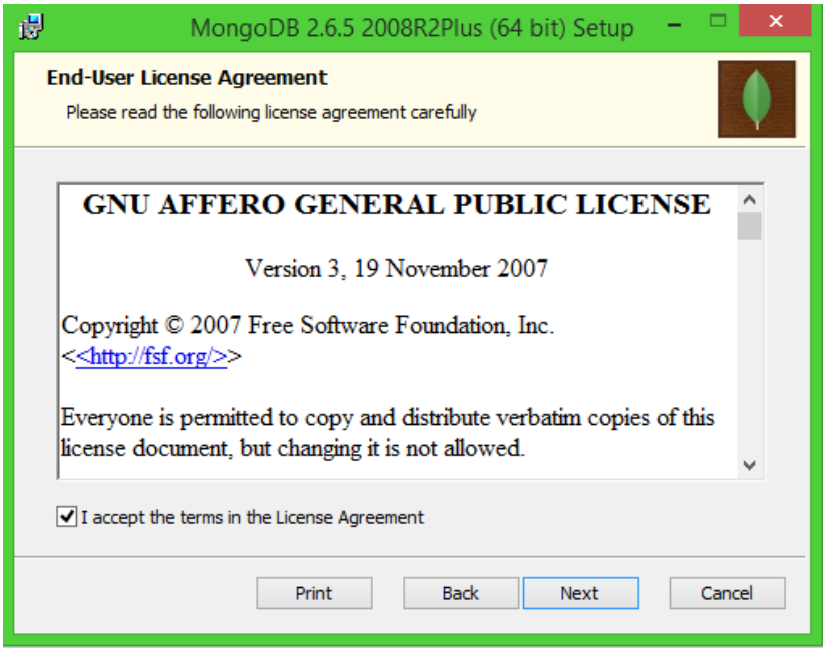

Figura 110 - Termo de Licença
Fonte: Autoria própria

A tela exibida a seguir pela Figura 111 irá designar como o usuário quer prosseguir, escolhendo uma das três opções de *setup*, customizado, típico ou completo. Isso irá depender da necessidade de quem irá utilizar o programa. Para facilitar o processo será utilizado o botão *Typical*.

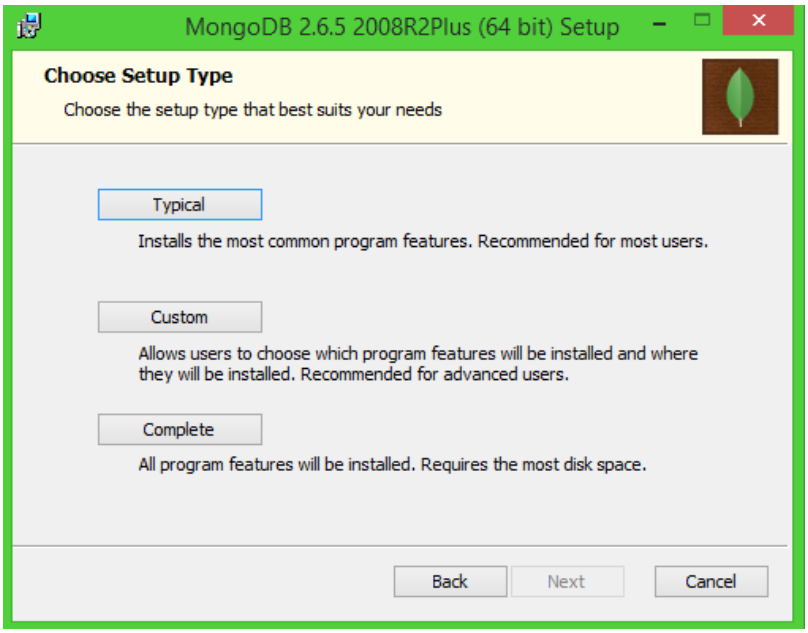

Figura 111 - Escolha do tipo de instalação
Fonte: Autoria própria

Escolhida a opção, pode-se ver que este é o último passo antes de se iniciar a instalação automática, sabendo que é fundamental clicar sobre o botão *Install*, como visto na Figura 112.

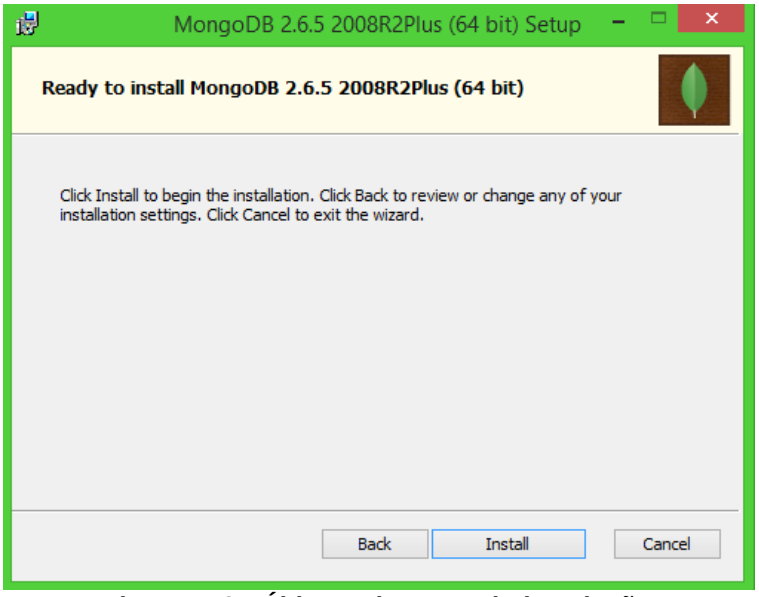

Figura 112 - Última tela antes da instalação
Fonte: Autoria própria

Depois disso, o que se tem a fazer é esperar a conclusão do procedimento, ilustrado na Figura 113.

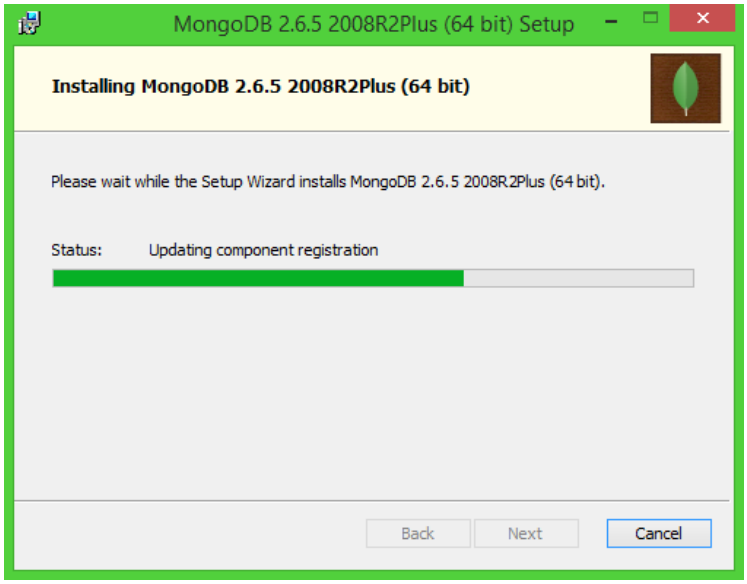

Figura 113 - Barra de progressão do processo de instalação
Fonte: Autoria própria

Finalizada todas as etapas, a Figura 114 mostra a exibição da última tela, onde o usuário terá somente que clicar no *Finish* e o programa estará pronto para ser usado.

Figura 114 - Processo finalizado
Fonte: Autoria própria

APÊNDICE D - Instalação do BerkeleyDB

O sistema gerenciador de banco de dados BerkeleyDB[4] atualmente é desenvolvido pela *Oracle Corporation*, rodando em diversas plataformas, o mesmo possui três produtos, o BerkeleyDB, o BerkeleyDB *Java Edition* e o Berkeley DB XML. Os dois últimos são para serviços mais específicos, por isso será instalado o primeiro, considerado de uso geral.

O programa segue as regras GNU, tendo assim uma licença pública, ou seja, gratuita. Para obtenção do mesmo, deverá ser acessado o endereço principal do desenvolvedor, o oracle.com, onde neste deve-se passar o cursor do mouse sobre o campo *Downloads* e acessar no quadro *Database* o *link* Oracle BerkeleyDB. Assim escolher a versão geral e de acordo com seu sistema operacional, neste caso será a versão 6.1.19 para *Windows*.

O processo de instalação seguido a partir de agora somente é válido para a versão do programa citada no parágrafo anterior, podendo variar caso o usuário escolha outra versão ou outra plataforma. Logo após finalizar o *download*, a próxima etapa começar a instalação do aplicativo em questão, dando um duplo clique no arquivo. Aparecerá a tela inicial, mostrada na Figura 115, onde deve-se clicar em *next*.

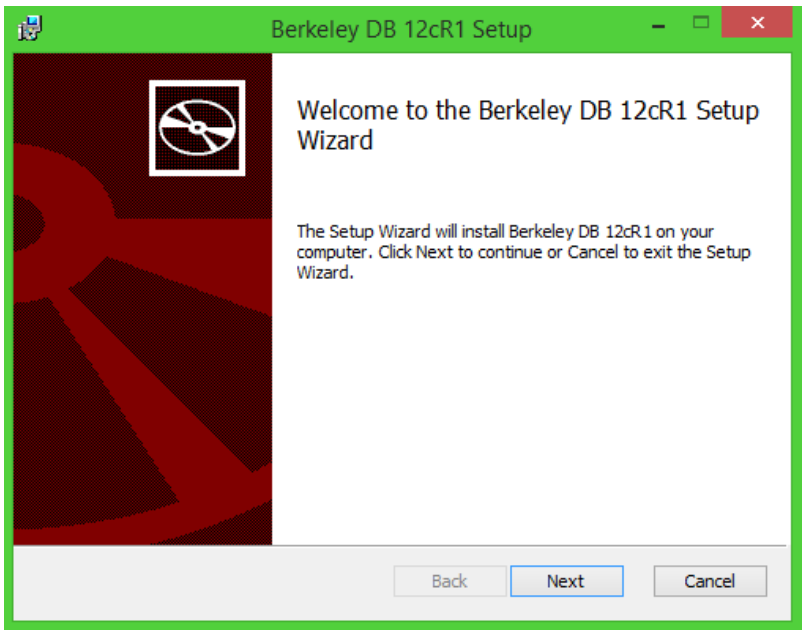

Figura 115 - Tela inicial de instalação
Fonte: Autoria própria

[4] O *site* para *download* do BerkeleyDB é o www.oracle.com.

A próxima interface leva o acordo de licença, descrevendo informações sobre a versão do produto e os termos de uso, como visto na Figura 116. Para habilitar o botão *next* o usuário precisa preencher a caixa de diálogo que aceita os termos descritos.

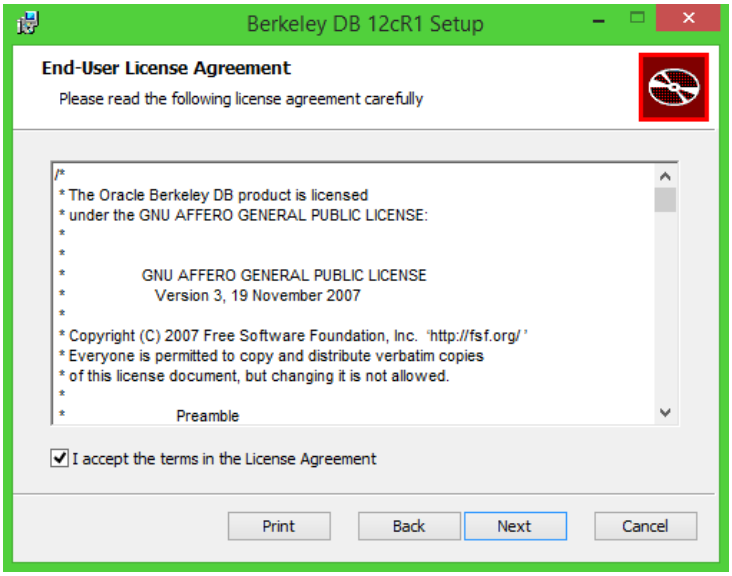

Figura 116 - Termo de licença
Fonte: Autoria própria

A Figura 117 ilustra a parte na qual será escolhido o local de armazenamento do programa, através do botão *Browse*. Feito isso, clica-se novamente no botão *next*.

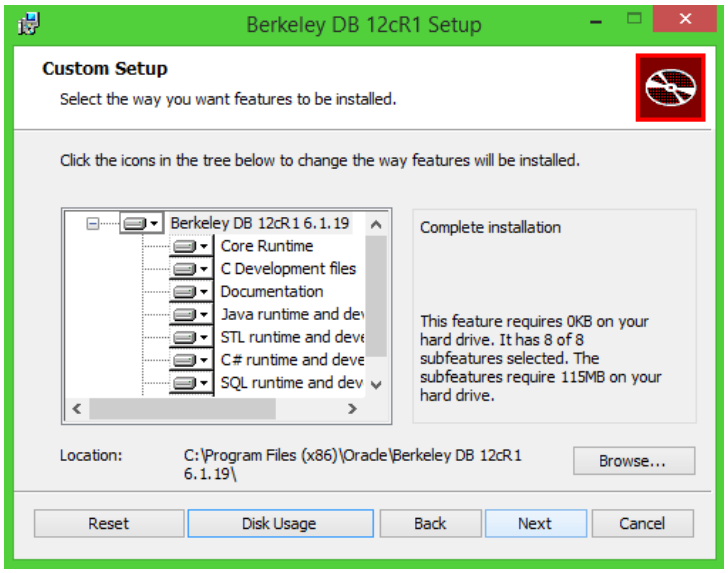

Figura 117 - Caminho para armazenamento do programa
Fonte: Autoria própria

Antes de efetivamente começar a instalação automática há ainda uma tela, descrita na Figura 118, onde o usuário deve clicar no *Install* para dar início ao processo.

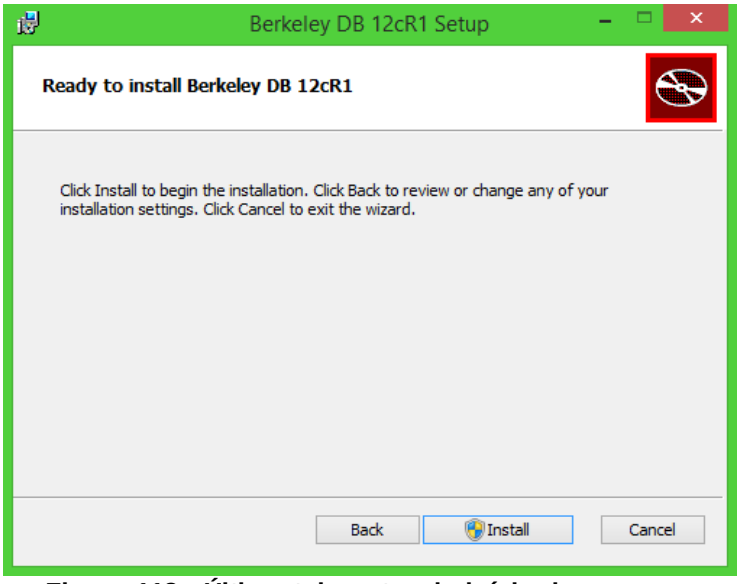

Figura 118 - Última tela antes do início do processo
Fonte: Autoria própria

Agora é necessário aguardar enquanto o computador instala o *software*, como pode ser visto na Figura 119 e o tempo de duração irá depender da configuração da máquina.

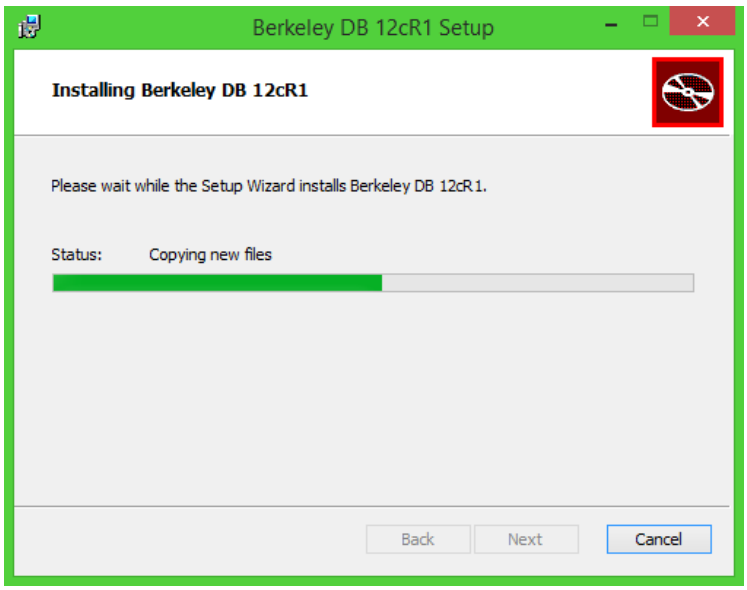

Figura 119 - Instalação em andamento
Fonte: Autoria própria

Terminado o processo, uma tela irá informar que o mesmo foi completado e o usuário deve apenas clicar no botão *Finish* para encerrar, como visualizado na Figura 120, estando assim o programa pronto para uso.

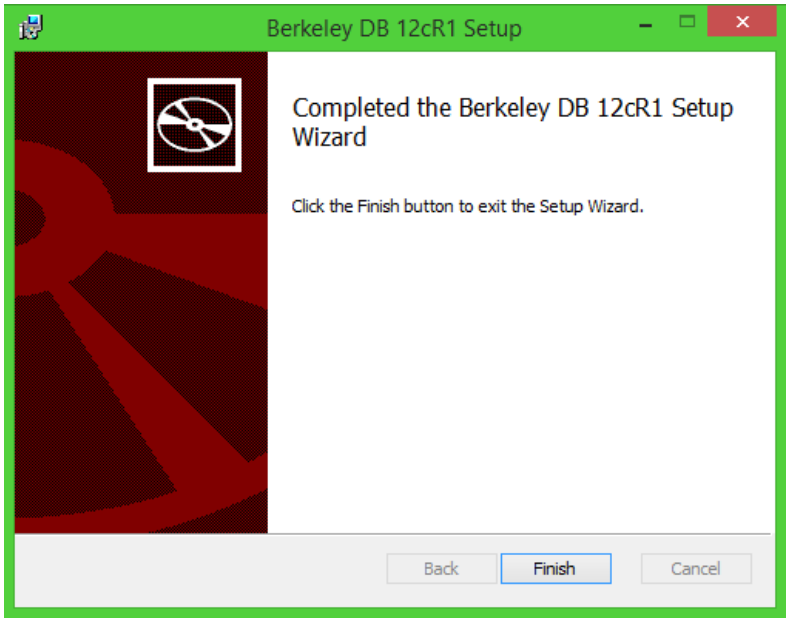

Figura 120 - Finalizando a instalação
Fonte: Autoria própria