Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

BANCO DE DADOS CASSANDRA:

Uma opção escalável e eficiente para o armazenamento de dados na *Web*.

CARLOS EDUARDO BIAGIO

Americana, SP
2016

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

BANCO DE DADOS CASSANDRA:

Uma opção escalável e eficiente para o armazenamento de dados na *Web*.

CARLOS EDUARDO BIAGIO

edubiagio@gmail.com

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Fatec-Americana, sob orientação do Doutor José Alberto Florentino Rodrigues Filho.

Área: Banco de dados.

Americana, SP

2016

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

471b	Biagio, Carlos Eduardo Banco de dados Cassandra: uma opção escalável e eficiente para o armazenamento de dados na WEB. / Carlos Eduardo Biagio. – Americana: 2016. 65f. Monografia (Graduação em Tecnologia em Análise e Desenvolvimento de Sistemas). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza. Orientador: Prof. Dr. José Alberto Florentino Rodrigues Filho 1. Banco de dados 2. WEB – rede de computadores I. Rodrigues Filho, José Alberto Florentino II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana. CDU: 681.3.07 681.519

CARLOS EDUARDO BIAGIO

BANCO DE DADOS CASSANDRA:

Uma opção escalável e eficiente para o armazenamento de dados na *web*.

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Fatec-Americana

Área: Banco de dados.

Americana, 20 de junho de 2016.

José Alberto Florentino Rodrigues Filho. (Presidente)
Doutor
Fatec – Americana

Antônio Alfredo Lacerda. (Membro)
Especialista
Fatec – Americana

Kleber de Oliveira Andrade. (Membro)
Mestre
Fatec - Americana

AGRADECIMENTOS

Agradeço ao meu orientador José Alberto que me instruiu durante o desenvolvimento dessa monografia, ao professor Clerivaldo que me auxiliou na escolha do tema e nas revisões. Agradeço também a todos os professores do curso, que foram muito importantes na minha vida acadêmica. E a todos os profissionais que trabalham na FATEC.

DEDICATÓRIA

Dedico primeiramente a Deus que me iluminou nos momentos de dificuldade, e colocou pessoas de bom coração no meu caminho. Aos meus pais João e Lourdes, e as minhas irmãs Liliane e Elaine que sempre estiveram me auxiliando e me apoiando incondicionalmente. A Marili que esteve sempre presente e me ajudando. Aos amigos que conheci durante o curso, em especial a Maria, o Bruno e a Denise.

.

RESUMO

Nos últimos trinta anos o modelo relacional dominou o mercado de banco de dados, porém com o aumento na quantidade de dados trafegando pela internet, o modelo relacional apresentou algumas limitações em relação a escalabilidade e a disponibilidade. E assim surgiu a necessidade da criação de modelos alternativos ao modelo relacional, essa fatia de mercado recebeu o nome de NoSQL - *Not only* (Não somente SQL). Dentre os bancos de dados não relacionais o Cassandra vem se mostrando uma opção muito eficaz e confiável, e objetivo dessa monografia é descrever as principais funções presentes no Cassandra que lhe permite gerenciar com eficiência grandes quantidades de dados, e garantir características como disponibilidade, escalabilidade, trabalhar de forma distribuída e possuir tolerância a falhas. A metodologia utilizada foi a pesquisa bibliografia baseando-se em obras de autores renomados na área de bancos de dados. E através de um exemplo para o qual foi implementando o Cassandra como banco de dados, já que um banco de dados relacional se mostraria ineficiente no cenário exposto.

Palavras Chave: Bancos de dados; NoSQL, Cassandra.

ABSTRACT

In the last thirty years the relational model dominated the database market, but with the increase in the amount of data traveling over the Internet, the relational model was presented some limitations on scalability and availability. And so one came the need to create alternative models to the relational model, this market share was called NoSQL (not only SQL). Among the non-relational databases Cassandra has proved a very effective and reliable option, and purpose of this paper is to describe the main functions present in Cassandra that allows it to manage large amounts of data, and ensure characteristics such as availability, scalability, work in a distributed manner and have fault tolerance. The methodology used was the research literature based on works by renowned authors in databases. And through implementing the Cassandra database on an example, which the relational database would prove ineffective in the presented scenario.

Keywords: Database; NoSQL; Cassandra.

Sumário

Lista de figuras

Lista de tabelas

Lista de siglas

SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
SGBD	Sistema Gerenciador de Banco de Dados
NoSQL	*Not only* (Não somente SQL)
CQL	*Cassandra Query Language* (Linguagem de Consulta Cassandra)
IMS	*Information Magagement System* (Sistema de Gerenciamento de Informações)
IDMS	*Integrated Database Management System* (Sistema Integrado de Gerenciamento de Banco de Dados)
OO	Orientado a objetos
RO	Relacional objetos
ACID	Atomicidade, Consistência, Isolação, Durabilidade.
ANSI	*American National Standards Institute* (Instituto Nacional Americano de Padrões)
ISO	*International Organization for Standardization* (Organização Internacional para Padronização)
BASE	*Basically Available, Soft state, and Eventual consistency*. (Basicamente disponível, Estado leve e Eventualmente consistente)
IP	Internet Protocol (Protocolo de internet)
SSL	Secure Socket Layer
UUID	*Universally Unique Identifies* (Identificador universal único)
UTF-8	*8-bit Unicode Transformation Format*

1. Introdução

O armazenamento de dados é uma necessidade presente no cotidiano da sociedade. Antigamente as empresas faziam uso de fichas de papel organizadas em pastas que eram guardadas em arquivos físicos, reaver informações e manter esses registros eram tarefas demoradas, onerosas, e tais atividades ficavam restritas a localização dos arquivos.

Com a evolução dos computadores na década de 60, os arquivos físicos começaram a migrar para arquivos digitais. Foi então que surgiram os primeiros bancos de dados, onde cada entidade era um arquivo, e um software era responsável por cadastrar, alterar, excluir e recuperar as informações, nesse modelo não era possível que uma entidade se relacionasse com outras.

Diante da necessidade de relacionamento entre as entidades, os softwares precisavam ficar cada vez mais complexos para permitir que o registro de um arquivo pudesse estar interligado a outro. Com essa dificuldade foram realizadas pesquisas buscando outras opções, e surgiram alguns outros modelos como o hierárquico e o de redes. Mas nenhum conseguiu solucionar de maneira eficaz a necessidade de existir relacionamento entre as entidades.

Foi então que no inicio da década de 70, Edgar Frank Codd propôs em seu artigo *Relational Model of Data for Large Shared Data Banks* (Modelo de dados relacional para grandes bancos de dados compartilhados), a possibilidade da criação de um banco de dados com base em um modelo matemático, que possibilitaria que os usuários inserissem e consultassem grandes quantidades de informações. Posteriormente uma equipe de pesquisadores da IBM, desenvolveu um protótipo chamado de *System/R*, usando as ideias propostas por Codd. Nessa mesma pesquisa foi criada a linguagem de consulta SQL - *Structured Query Language* (Linguagem de Consulta Estruturada), que se tornou a linguagem padrão para os bancos de dados relacionais.

Os SGBDs (Sistema gerenciador de banco de dados) que surgiram em seguida foram baseados no Modelo Relacional, como o *Oracle*, o *DB2*, e posteriormente o *MySQL* e *SQL Server*. No atual mercado de banco de dados o Modelo relacional é dominante. Entretanto, em algumas situações um banco de dados relacional apresenta alguns fatores limitantes. Portanto, esse trabalho visa

apontar os cenários onde o gerenciamento de quantidades massivas de dados tem se mostrado um problema para utilização de bancos de dados relacionais, ou seja, quando o Modelo Relacional não apresenta a eficiência esperada.

A solução encontrada pelos desenvolvedores para tornar o gerenciamento de grandes quantidades de dados eficiente. Foi a criação de modelos alternativos que flexibilizavam as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) fortemente presente no modelo relacional, o que permitiu ganho em características como: alta disponibilidade, escalabilidade horizontal e descentralização. Em 1998 foi utilizado o termo NoSQL - *Not only* (Não somente SQL) para intitular o movimento de criação de bancos de dados não relacionais.

Esse trabalho possui o objetivo geral de Identificar os principais motivos que levaram os desenvolvedores a criar um modelo alternativo para solucionar a dificuldade do modelo relacional em trabalhar com grandes volumes de dados. E objetivo específico de descrever as principais funções que o Cassandra possui para garantir a disponibilidade, escalabilidade, trabalhar de forma distribuída e possuir tolerância a falhas.

O Cassandra é um poderoso banco de dados NoSQL com capacidade de armazenar centenas de *terabytes*, é altamente escalável, com suporte para replicação em vários *data centers*, possui arquitetura descentralizada sem nenhum ponto de falha e suporta escrita de grandes volumes de dados com alto desempenho. Foi criada uma linguagem de consulta exclusiva chamada CQL - *Cassandra Query Language*, a qual possui algumas semelhanças com a SQL. Cassandra foi desenvolvido pelo Facebook, e foi lançado como um projeto *open-source* em 2008. Atualmente faz parte da fundação Apache como um projeto de alto nível. O CERN (Organização Europeia para a Pesquisa Nuclear) usa o Cassandra desde 2012 para armazenar dados de pesquisas realizadas no colisor de partículas ATLAS. (CASSANDRA, 2013).

A Metodologia utilizada para a elaboração deste trabalho foi a pesquisa bibliográfica que Severino (2014) descreve da seguinte forma “A *pesquisa bibliográfica* é aquela que se realiza a partir do registro disponível, decorrente de pesquisas anteriores, em documentos impressos, como livros, artigos, teses etc.”. Baseando-se nas obras de autores renomados no assunto de banco de dados para explorar o modelo relacional. E utilizando obras sobre bancos de dados NoSQL que

abordam fatores que limitantes que os bancos de dados com base no modelo relacional apresentam. Por fim explorando as principais propriedades e características presentes no banco de dados Cassandra que lhe garante escalabilidade elástica e possibilidade de trabalhar de forma distribuída sem ponto de falha.

Este trabalho justifica-se pelo aumento substancial do volume de dados provenientes de aplicações *web,* conhecido como *Big Data*, onde é necessário um gerenciamento eficiente e escalável, e o modelo relacional não é capaz de atender essa demanda. O referencial teórico baseia-se no livro Sistemas de Bancos de Dados, de NAVATHE e ELMASRI, 2011, na obra de BRADBERRY e LUBOW *Practical Cassandra: A Developer's* Approach e no artigo dos desenvolvedores do Cassandra LAKSHMAN e MALIK *Cassandra - A Decentralized Structured Storage System*.

2. Banco de Dados

De acordo com Silberschatz, Korth e Sudarshan (2012), banco de dados “é uma coleção de dados inter-relacionados, representando informações sobre um domínio específico”, ou seja, sempre que tivermos um conjunto de informações relacionadas referente ao mesmo assunto, podemos definir como um banco de dados. Porém essa é uma definição muito abrangente, assim Elmasri e Navathe (2011) complementam que “um banco de dados representa algum aspecto do mundo real, às vezes chamado de minimundo ou de universo de discurso (*UoD – Universe of Discourse*). As mudanças no minimundo são refletidas no Banco de Dados”. Portanto, um banco de dados possui alguma fonte derivada da fração mundo real que esta sendo representada. Para que um banco de dados seja uma representação autêntica, ele deve refletir todas as mudanças ocorridas em seu minimundo. Ou seja, precisa ser constantemente atualizado.

Em um banco de dados computadorizado tarefas como inserção e atualização de dados podem ser feitas através de programas chamados de sistema gerenciador de banco de dados (SGBD). Elmasri e Navathe (2011) definem da seguinte forma “O SGBD é um sistema de software de uso geral que facilita o processo de definição, construção, manipulação e compartilhamento de banco de dados entre diversos usuários e aplicações”. Além de definir as estruturas de armazenamento dos dados e manipular as informações. Um SGBD é responsável por outras funções importantes como garantir a segurança das informações, e não permitir acessos não autorizados ou maliciosos. A união do banco de dados com o software SGBD pode ser chamada de sistema de banco de dados.

Neste sentido Silberschatz, Korth e Sudarshan (2012), explicam que “O maior benefício de um banco de dados é proporcionar ao usuário uma visão abstrata dos dados. Isto é, o sistema acaba por ocultar determinados detalhes sobre a forma de armazenamento e manutenção desses dados”. Como muitos usuários dos sistemas não possuem conhecimento avançados em informática é função dos desenvolvedores ocultar essa complexidade através de vários níveis de abstração, e com isso simplificar a interação do usuário com o sistema.

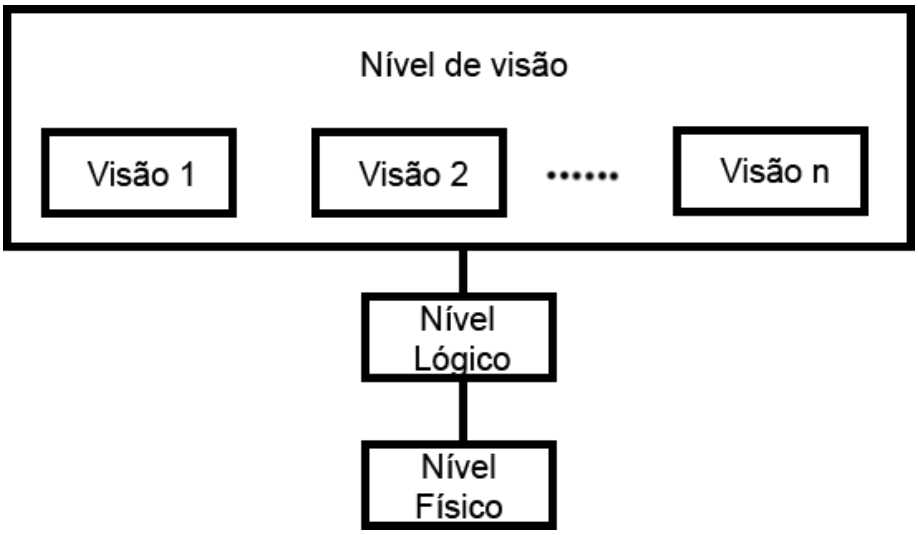
Pode-se citar três níveis ou camadas de abstração:

Nível Físico. Neste nível os dados são descritos da maneira que realmente estão armazenados. Ou seja, em bits. Por isso, é a camada mais baixa de abstração.

Nível Lógico. É o segundo nível e descreve todos os dados que estão armazenados e as maneiras que se relacionam no banco de dados. Esta camada é usada pelos administradores para definir quais informações pertenceram ao banco de dados.

Nível de visão. É o nível mais alto de abstração, a maioria dos usuários que tem acesso não precisam ter conhecimentos de todas as informações do banco de dados. Por isso, para simplificar a interação, apenas parte do banco de dados é apresentada. E o sistema é capaz de oferecer varias visões do mesmo banco de dados.

Figura 1 - Representação de níveis de abstração



.

Silberschatz, Korth e Sudarshan (2012), (Adaptado).

2.1. Independência de dados

Independência de dados é a capacidade de alterar a definição de um esquema em uma camada de abstração sem afetar a definição dos outros esquemas. A independência física diz respeito à habilidade de fazer alterações no nível físico sem que essas mudanças surtam efeito em outros níveis, ou seja, podemos mover o arquivo do banco de dados de um dispositivo para outro, ou

renomear o arquivo e a camada lógica permanecera inalterada. Já a independência lógica é a habilidade de fazer alterações na camada lógica sem a necessidade de alterar os esquemas externos, ou seja, após mudanças na camada lógica os esquemas externos ou programas de aplicação devem trabalhar como antes. (Elmasri; Navathe, 2011).

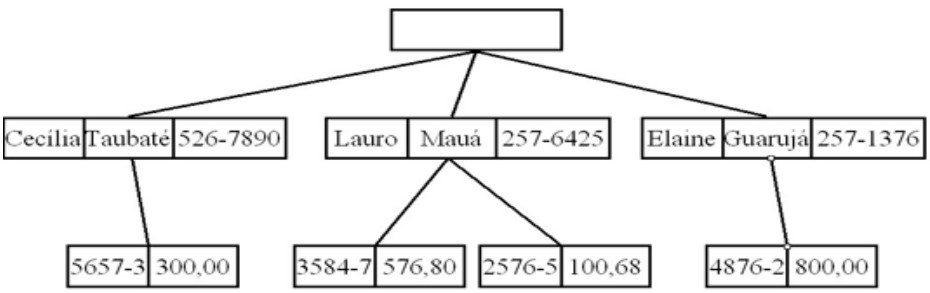
2.2. Modelos de banco de dados

Os projetos de exploração espacial trazem desenvolvimentos significativos para ciência e indústrias de tecnologia. Como parte do projeto Apollo da NASA (Agencia Espacial Americana), foi criado um dos primeiros bancos de dados baseados no sistema hierárquico, o GUAM - *Generalized Update Access Method*. Posteriormente foram desenvolvidos sistemas fundamentados no modelo de redes. Os bancos de dados desenvolvidos em ambos os modelos eram em sua maioria implantados em mainframes. (Oppel, 2011).

2.2.1. Modelo Hierárquico

Em um banco de dados hierárquico os registros são conectados uns aos outros através de links. Cada registro contém apenas um valor e possui uma coleção de atributos. Um link pode interligar somente dois registros. Neste modelo os registros são organizados na forma de uma árvore com raiz, que agrupada em um conjunto recebe o nome de árvore de banco de dados, que formam uma floresta. A representação de modelo é feita através do diagrama de estrutura de árvore, que pode ser visto na figura 2.

Figura 2 - Representação do modelo hierárquico.



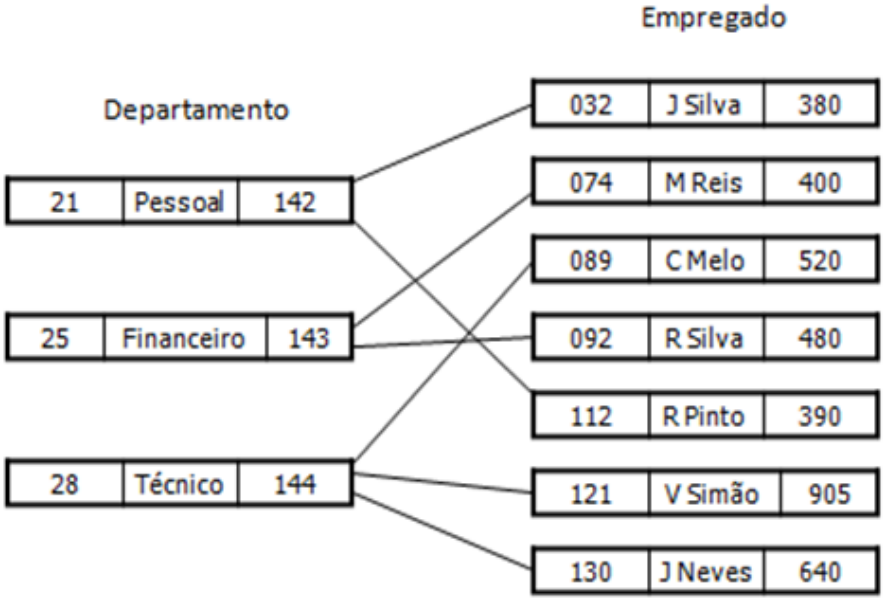
Silberschatz, Korth e Sudarshan (1999), (Adaptado).

O principal banco de dados que usava estruturas hierárquicas foi desenvolvido pela IBM nos anos 60, recebeu o nome de IMS - *Information Magagement System*. Era usado principalmente em mainframes, foi o banco de dados mais usado durante as décadas seguintes. O IMS possui extensos recursos de processamento de transações. Ainda é usado nos dias atuais e após atualizações oferece suporte para linguagens como Java e XML. A linguagem de manipulação de dados do IMS é chamada de DL/I. (Silberschatz; Korth; Sudarshan, 2012).

2.2.2. Modelo de redes

O modelo de redes foi criado como uma extensão ao modelo hierárquico. Sendo que cada registro possui uma coleção de atributos, e uma associação é feita através de um link entre dois registros. A diferença é que cada registro filho pode ser associado a mais de um registro pai, eliminando assim o conceito de hierarquia. O modelo de redes pode ser representado através de um diagrama de estrutura de dados como visto na Figura 3.

Figura 3 - Representação do modelo de redes.



Silberschatz, Korth e Sudarshan (2012), (Adaptado).

O mais popular banco de dados com base na arquitetura de redes é o IDMS - *Integrated Database Management System*, desenvolvido pela Goodcich Corporation, foi lançado no ano de 73 e em 89 foi adquirido pela CA Technologies. O IDMS ainda está em operação atualmente principalmente em mainframes.

Porém, esses sistemas apresentavam alguns problemas descritos da seguinte forma por Elmasri e Navathe (2011):

> Um dos principais problemas com os sistema de bancos de dados antigos era a mistura de relacionamentos conceituais com o armazenamento e posicionamento físico dos registros no disco. Logo esses sistemas não ofereciam capacidade suficiente para abstração de dados e independência entre dados e programas [..].

Assim, era muito grande a dificuldade de programar de forma eficaz as novas consultas que precisavam de uma organização diferente. Outro problema era que novas transações podiam ser feitas apenas através de linguagem de programação, o que as tornavam demoradas e onerosas.

2.2.3. Modelo Relacional

Diante da complexidade do modelo de redes e do hierárquico, e por ambos apresentarem problemas de flexibilidade, muitos pesquisadores começaram a procurar outras soluções. Apenas alguns eventos nas historia da computação foram realmente revolucionários, mas o artigo publicado por E. F. (Ted) Codd que levou a criação do modelo relacional foi sem dúvidas um desses momentos.

Elmasri e Navathe (2011) descrevem o trabalho de Codd da seguinte maneira:

> O modelo de dados relacional foi introduzido inicialmente por Ted Codd, da IBM Research, em 1970, em um artigo clássico, que atraiu atenção imediata devido sua simplicidade e base matemática. O modelo usa o conceito de relação matemática – que se parece com uma tabela de valores – como seu bloco de montagem básico, e sua base teórica reside em uma teoria de conjunto e lógica de predicado de primeira ordem [...].

Os primeiros bancos de dados relacionais começaram a ser desenvolvidos no fim dos anos 70, como o *System/R* da IBM. Junto com ele, os pesquisadores criaram a linguagem SQL - *Structured Query Langague*, posteriormente os sistemas criados passaram a utilizar SQL como linguagem de

consulta. Por esse motivo ela evoluiu e ser tornou a linguagem padrão para bancos de dados relacionais. O modelo relacional veio a ser o primeiro modelo de dados para aplicações comerciais. Atualmente é o principal modelo no mercado de banco de dados.

2.2.4. Modelo orientado a objetos

O modelo orientado a objetos OO foi criado inicialmente nos anos 70, mas não teve significante uso comercial até o inicio dos anos 90. Essa necessidade repentina surgiu devido ao aumento da utilização de linguagens baseadas no paradigma de orientação a objetos como C++, Java e C#, e o objeto precisa ser transformado em dados para ser gravado em um banco de dados relacional, tarefa que consome tempo e recursos computacionais. E também pela incapacidade dos SGBDs relacionais em trabalhar com tipos de dados complexos como objetos, imagens e arquivos de vídeos. A popularização da internet criou uma enorme demanda por esses dados complexos.

Como objeto podemos entender um agrupamento lógico de dados relacionados e lógica de programação que representam algo do mundo real. Geralmente um objeto tem associado a ele variáveis, mensagens que podem conter zero, um ou mais parâmetro e um conjunto de métodos. A principal diferença entre o modelo OO e os apresentados anteriormente é que as variáveis só podem ser acessadas através de seus métodos. Essa propriedade é chamada de encapsulação.

O fato dos dados ficarem encapsulados ou escondidos e estarem acessíveis somente através de métodos compromete uma importante característica dos bancos de dados, de permitir consultas baseadas no conteúdo. Outro problema se da pela dificuldade das linguagens de pesquisa de objeto, e por não serem padronizadas. (Silberschatz; Korth; Sudarshan, 2012),

2.2.5. Modelo Relacional-Objeto

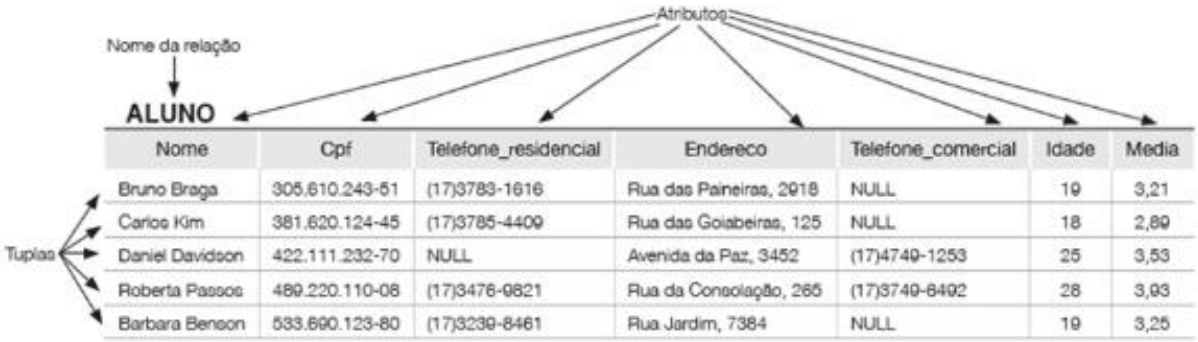
Para compreender melhor o modelo relacional-objeto RO, podemos defini-lo como uma extensão ao modelo relacional que incorporou algumas características do modelo orientado a objetos. Em outras palavras esse modelo é

capaz de lidar diretamente com objetos através de linguagens de consulta relacionais, como a SQL. Alguns exemplos de bancos de dados RO são: *Oracle*, *DB2* e *Infomix*.

3. Bancos de dados Relacionais

O modelo relacional possui um sólido fundamento teórico baseado na teoria matemática dos conjuntos e na álgebra relacional. Com isso, um banco de dados é representado como um conjunto de relações, que podem ser visualizadas como tabelas, que por sua vez possui linhas e colunas. “Na termologia do modelo relacional, uma linha é chamada de tupla, um cabeçalho da coluna é chamado de atributo e a tabela é chamada de relação.” (Elmasri; Navathe, 2011). Uma tupla representa uma entidade ou relacionamento do mundo real, e cada coluna divide os valores de cada linha e definem seu tipo, portanto todos os valores de uma coluna possuem uma mesma classificação. Na figura 4 é possível observar uma relação.

Figura 4 - Representação do modelo relacional.



Nome	Cpf	Telefone_residencial	Endereco	Telefone_comercial	Idade	Media
Bruno Braga	305.610.243-61	(17)3783-1616	Rua das Paineiras, 2918	NULL	19	3,21
Carlos Kim	381.620.124-45	(17)3785-4409	Rua das Goiabeiras, 125	NULL	18	2,89
Daniel Davidson	422.111.232-70	NULL	Avenida da Paz, 3452	(17)4749-1253	25	3,53
Roberta Passos	489.220.110-08	(17)3476-9821	Rua da Consolação, 265	(17)3749-8492	28	3,93
Barbara Benson	533.690.123-80	(17)3239-8461	Rua Jardim, 7384	NULL	19	3,25

Silberschatz, Korth e Sudarshan (2012), (Adaptado).

3.1. Chaves

Chaves são um conceito fundamental do modelo relacional, pois através delas é possível estabelecer relacionamentos entre tuplas de relações de uma forma simples e eficaz. Existem três tipos de chaves:

Chave primária: É responsável por identificar de forma exclusiva cada tupla de uma relação. Ou seja, em uma tabela jamais poderá existir linhas com a mesma chave primária. Podemos utilizar um atributo ou uma combinação deles para formar uma chave primária, que nesse caso recebe o nome de chave primária composta.

Chave estrangeira: Possui a finalidade de estabelecer o relacionamento entre relações referenciando o valor de um atributo que foi usado como chave primária. É importante frisar que a referência não diz respeito somente à outra relação. Ou seja, uma chave estrangeira pode fazer menção à chave primária da relação que pertence. Quando isso ocorre recebe o nome de relação recursiva.

Chave alternativa: Ocorre nas situações que uma tabela contém mais que uma combinação de atributos com valores únicos. Ou seja, por possuir a propriedade de unicidade ela poderia ser uma chave primária. A chave candidata não é aplicada na pratica.

3.2. Domínio

Outro importante conceito do modelo relacional são os domínios, que após a definição de uma relação, devemos estabelecer quais os valores que cada atributo pode assumir. Esse conjunto de valores recebe o nome de domínio. É fundamental especificar também quais são os atributos que o valor pode ser vazio ou nulo. Atributos que não permitem valores vazios são denominados atributos obrigatórios. Geralmente chaves estrangeiras e chaves primárias são atributos obrigatórios.

3.3. Forma Normal

Normalização é uma técnica criada por Codd no ano de 1972, composta por condições que devem ser seguidas durante o projeto de um banco de dados. Permitindo assim o armazenamento consistente dos dados, acesso eficiente e redução da redundância. Essas condições receberam o nome Forma Normal.

Originalmente Codd propôs três dessas formas, porém atualmente existem outras comumente aceitas.

Primeira Forma Normal: Uma relação deve conter apenas atributos com valores atômicos e o valor de todos os atributos em uma tupla precisa ter um único valor no domínio daquele atributo. Ou seja, não pode existir uma tupla com atributos multivalorados ou valores repetidos. Caso isso ocorra é necessário criar uma nova relação para cada atributo não atômico.

Segunda Forma Normal: As relações não podem conter atributos não chaves que não dependam da chave primária da relação, isto é, um atributo que não é chave primária não pode depender de parte da chave primária. E também de satisfazer a primeira forma normal.

Terceira Forma Normal: Uma relação não pode conter atributos não chaves com valores dependentes de outros atributos, assim, os atributos não chaves devem ser independentes mutuamente. Além de cumprir a segunda forma normal.

Terceira Forma Normal de Boyce-Codd: Posteriormente Boyce e Codd acrescentaram mais um requisito a terceira forma normal e que ficou conhecida como 3.5, e diz que não pode existir dependência funcional dentro da chave primária.

Quarta Forma Normal: Em uma relação não pode existir dependências multivaloradas. Ou seja, a presença de uma tupla não deve implicar na existência de outra tupla na relação. E cumpri o proposto na terceira forma normal.

Quinta Forma Normal: Se uma relação for dividida, e não puder ser reconstruída através da junção das outras relações. E atender a quarta forma normal ela cumpriu o proposta na quinta forma normal.

3.4. Transações

Uma transação é uma série de ações que devem ser processadas completamente processadas, ou não deve ser processada nenhuma parte. Nas palavras de Silberschatz, Korth e Sudarshan (2012):

> Uma transação é uma unidade de execução de programa que acesse e, possivelmente, atualiza vários itens de dados. Uma transação, geralmente, é o resultado da execução de um programa de usuário escrito em uma linguagem de manipulação de dados de alto nível ou em uma linguagem de programação (por exemplo, SQL, COBOL, C ou Pascal), e é delimitada por declamações (ou chamadas de função) da forma begin trasaction e end trasaction. A transação consiste em todas as operações ali executadas, entre o começo e o fim da transação.

É importante assegurar a execução apropriada do conjunto de operação transações, e administrar a execução de transações simultâneas, mantendo assim a integridade dos dados. Para isso, uma transação deve manter determinadas propriedades, que podem lembradas usando o acrônimo ACID (Atomicidade, Consistência, Isolação, Durabilidade).

Atomicidade: Uma transação deve conservar a sua integridade, ou seja, deve ser completamente bem sucedida ou falhar. Quando for bem sucedida, todas as mudanças que foram feitas pela transação devem ser preservadas pelo sistema. Caso uma transação falhe, todas as mudanças que seriam feitas, devem ser completamente abortadas.

Consistência: Uma transação precisa manter a consistência, em outras palavras, quando uma transação for bem sucedida, deve transformar o estado do banco de dados de um estado de consistência para outro.

Isolação: Cada transação deve ser executada de forma isolada, independente de ter outras transações ocorrendo ao mesmo tempo.

Durabilidade: Todas as mudanças realizadas com sucesso devem persistir no banco de dados, mesmo que ocorram falhas no sistema. Em outras

palavras, todas as mudanças devem ser armazenadas de forma que não desapareçam caso o banco de dados falhe.

3.5. SQL

SQL se tornou a linguagem universal para bancos de dados relacionais, praticamente todos SGBDs atuais tem suporte para ela. Os motivos por essa grande aceitação é o tempo e o esforço que foram dedicados para o desenvolvimento de suas características linguísticas e suas normas, por isso a SQL se tornou portável entre vários SGBDs diferentes.

O percursor da SQL era chamado de QUEL, que foi desenvolvida junto com o System/R, banco de dados relacional experimental da IBM. Porém, outros fabricantes criaram seus SGBDs comerciais, como a *Oracle* e a *Ingres*. Com isso, a IBM lançou em 1982 o SQL/DS, com a linguagem de consulta SEQUEL - *Structured English Query Language*, mas esse nome já era patenteado por uma empresa de aviação inglesa. O nome foi trocado para SQL.

Comitês de padrão da SQL foram formados em 1986 pelo ANSI - *American National Standards Institute,* em 1987 pelo ISO - *International Organization for Standardization*. Então dois anos depois foi publicado o primeiro padrão, chamado de SQL-89, e após três anos foi ampliada com a SQL-92. A terceira geração, publicada em 1999, chamada de SQL-99 ou SQL3, incluiu revisões que permitiram a SQL funcionar em um banco de dados Objeto-Relacional. Revisões adicionais aconteceram em 2003 e 2006, atualmente continuam ocorrendo os trabalhos de padronização.

Elmasri e Navathe (2011) enfatizam uma importante característica da SQL da seguinte maneira: ”SQL é uma linguagem de banco de dados abrangente: tem instruções para definição de dados, consultas e atualizações. Logo ela é uma DDL e uma DML”. Assim sendo, a SQL é divida em categorias de acordo com as operações que se deseja efetuar.

3.5.1. DQL – *Data Query Language* (Linguagem de consulta de dados)

Responsável por realizar consultas nas relações sem alterar nenhum dado ou objeto do banco de dados. Essa categoria contém um único comando, que alguns autores o enquadram como fazendo parte da DML, mas nesse caso vamos fazer a distinção, já que através dele não é possível manipular dados. (Oppel, 2011).

SELECT: Através desse comando, é possível buscar informações no banco de dados. Devemos especificar quais atributos retornaram no resultado. Para refinar nossa busca, existem as seguintes cláusulas que podemos usar:

FROM: Aponta quais relações contém os atributos que serão selecionados.

WHERE: Especificar quais condições uma tupla devem atender para estar na seleção.

ORDER BY: Define a ordem pela qual as tuplas serão organizadas.

GROUP BY: Separa as tuplas selecionadas em grupos específicos.

DISTINCT: Exibe os dados selecionados sem repetição.

UNION: Combina o resultado de duas consultas em uma única relação com tuplas correspondentes.

JOIN: Usando em conjunto com a cláusula **FROM**, esse comando é usado para juntar relações.

3.5.2. DML – *Data Manipulation Language* (Linguagem de manipulação de dados)

Realiza consultas baseadas tanto na álgebra relacional quanto no cálculo relacional de tuplas, além de possuir comandos para inserção, exclusão e modificação de tuplas no banco de dados. Essa categoria contem os comandos:

INSERT: Adiciona novas tuplas a uma relação, isso pode ser feito de duas formas, usando a cláusulas **VALUES**, seguida dos valores que serão inseridos, dessa maneira será criada apenas uma tupla de cada vez. Outra maneira de usar o comando **INSERT** é usando uma *subquery*, usando o comando **SELECT** par

retornar as informações que serão inseridas, será crida uma linha para cada registro que o **SELECT** retornar.

UPDATE: Atualiza o conteúdo de um ou mais atributos de uma relação, descritos no comando. A cláusula **SET** é usada para definir quais atributos serão alterados e o novo valor que o atributo deve assumir. Já a cláusula **WHERE** tem a função de estabelecer o limite de ação do comando, ou seja, para alterar somente as tuplas que se enquadram dentro das condições especificadas. Caso a cláusula **WHERE** for omitida serão atualizados todas as tuplas da relação.

DELETE: Remove uma ou mais tuplas de uma relação, este comando não pode ser aplicado em atributos, ou seja, só pode apagar a tupla por completo. Uma cláusula **WHERE** pode ser usada para limitar as tuplas afetadas, no caso de ser omitida, todo o conteúdo de uma relação será apagado.

3.5.3. DDL – *Data Definition Language* (Linguagem de definição de dados)

Oferece comandos para definição de esquemas de relações, exclusão de relações, criação de índices e modificação nos esquemas de relações. Essa categoria inclui três comandos básicos:

CREATE: Cria em um banco de dados novos objetos de um tipo especificado no comando.

ALTER: Troca as definições de um objeto existente no banco de dados.

DROP: Apaga um objeto existente no banco de dados.

Dentro deste contexto podemos considerar um objeto, os elementos associados ao banco de dados, como por exemplo, tabelas, view, *index* etc.

3.5.4. DCL - *Data Control Language* (Linguagem de controle de dados)

Gerencia privilégios ou autorizações para que usuários possam realizar ações no banco de dados. Os privilégios são controlados por dois comandos:

GRANT: Define quais operações serão permitidas a um determinado usuário, por exemplo, quais serão as relações que ele pode realizar consultas, ou se ele pode manipular dados no banco de dados.

REVOKE: Remove ou restringes as ações que determinado usuário pode realizar no banco de dados.

3.5.5. DTL - *Data Transaction Language* (Linguagem de transação de dados)

Oferece suporte para garantir a atomicidade de uma transação, ou seja, se todas as instruções de uma transação estão corretas o comando **COMMIT**, permite finalizar a execução da transação, e no caso de ocorrer algum erro ou falha, o comando **ROLLBACK** faz com que as mudanças sejam revertidas até a última vez que o **COMMIT** foi executado. (Oppel, 2011).

3.6. Escalabilidade

Escalabilidade diz respeito à maneira que um sistema pode suportar ou ser modificado para suprir um aumento na demanda por seus serviços. Bondi, 2000 descreve com as seguintes palavras: “O conceito conota a habilidade de um sistema acomodar um número crescente de elementos ou objetos, para processar crescentes cargas de trabalhos graciosamente e/ou ser suscetível a ser ampliado”. No caso de um banco de dados, a escalabilidade tem a ver com a facilidade em que o banco pode gerenciar uma crescente quantidade de dados e ou transações, sem perder desempenho. É importante não confundir escalabilidade com desempenho, pois escalabilidade não tem nada a ver com ser rápido, tem foco somente em

quantidade. Mas também não quer dizer que um sistema com bom desempenho precisa ser escalonado.

O principal problema com o modelo relacional, é que ele é muito difícil de ser escalável. Isso ocorre devido a sua estruturação pouco flexível e menos adequada para situações que o sistema precisa crescer. Esse problema tem se tornado cada vez maior principalmente em aplicações *web*, onde a quantidade de dados armazenados cresce exponencialmente.

3.6.1. Escalabilidade Vertical

A opção encontrada para continuar usando o modelo relacional nesse tipo de situação, foi usar a escalabilidade vertical, que consiste em atualizar o servidor. Ou seja, atualizar componentes, ou ainda substituir o servidor por um melhor. Essa opção pode funcionar de inicialmente, porém no caso do sistema continuar crescendo ela se torna inviável devido seu alto custo.

3.6.2. Escalabilidade Horizontal

Consiste em adicionar nós físicos em um sistema, como por exemplo, adicionar mais computadores em um sistema distribuído. Uma das vantagens da possibilidade de escalar um sistema horizontalmente ao invés de verticalmente, se dá pela grande oferta de hardware, tornando muitas vezes mais barato a escalabilidade horizontal. Pois escalonando um sistema verticalmente o valor de um computador com grande capacidade de processamento acaba sendo superior ao de vários computadores de menor porte, que juntos atingem a demanda necessária.

Para usar a escalabilidade horizontal em um banco de dados, o sistema precisa ter sido projetado desde sua concepção. Fato esse que não ocorre no modelo relacional. Porém, esta é uma característica fortemente presente nos banco de dados não relacionais, que foram criados para trabalhar de forma distribuída.

4. Bancos de dados não relacionais

Conhecidos também por NoSQL - *Not Only SQL* (Não somente SQL), termo que foi criado em 1998 por Carlo Strozzi, para nomear sua aplicação Strozzi NoSQL, que apesar de fazer uso do modelo relacional, as consultas não usavam a linguagem SQL. Em 2009 o termo voltou à tona em um evento organizado em San Francisco. Califórnia. Para discutir sobre o surgimento de bancos de dados não relacionais distribuídos e com código aberto. A partir de então o NoSQL tornou-se um termo genérico para definir os diversos bancos de dados que não seguem o modelo relacional, trabalham de forma distribuída, possuem escalabilidade horizontal e possuem código aberto. (Tiwari, 2001)

Nos últimos seis anos muitos bancos de dados foram desenvolvidos, atualmente existem mais de 220 diferentes tipos de NoSQL de acordo com o site NoSQL Databases[1]. O motivo desse movimento estar evoluindo aceleradamente é devido ao enorme fluxo de informação que requer armazenamento de alta velocidade e disponibilidade elevada. Como essa demanda só tende a crescer, os bancos de dados não relacionais serão uma realidade cada vez mais presente no cotidiano dos desenvolvedores.

As transações no modelo relacional respeitam as propriedades ACID, e os modelos não relacionais encontraram uma maneira de flexibilizar essas características para obter melhores resultados em disponibilidade e desempenho. Como podemos ver a seguir.

4.1. Teorema de CAP

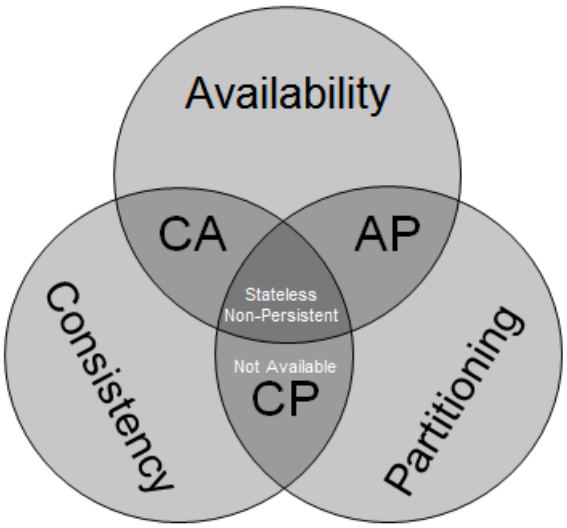
Também conhecido como teorema de Brewer, afirma que para um sistema distribuído é impossível satisfazer simultaneamente as três propriedades: Consistência, Disponibilidade e Tolerância de partição. (Mishra 2014).

Consistência: possui significado diferente da definição usada nas propriedades ACID. Nesse caso, diz respeito que todos nós, em um grupo, veem os mesmos dados ao mesmo tempo. Ou seja, uma consulta realizada em qualquer nó,

[1] Disponível em http://nosql-database.org/.

terá sempre o mesmo resultado, independente do nó que atender. Também implica que quando um comando de atualização for executado, o valor trocado em um nó, será replicado antes que seja realizada a próxima consulta. A Figura 5 mostra uma representação do Teorema de CAP.

Figura 5 - Representação do Teorema de CAP.



(Mishra 2014), (Adaptado).

“A” vem do inglês *Availability* (Disponibilidade) e se refere à garantia de que independente do sucesso ou falha da solicitação será retornada uma resposta. Isto é, as operações do sistema continuarão independentemente de um possível problema no sistema. Disponibilidade permite que o sistema tente lidar ou compensar problemas externos como falhas de hardware, falhas de rede e quedas de energia etc.

Tolerância de partição, diz respeito à capacidade de um sistema distribuir de forma eficaz a carga entre vários nós. A carga pode ser de dados ou consultas. Com isso, mesmo que alguns nós tenha caído o sistema continuará funcionando. A técnica usada para gerenciar a distribuição de cargas entre os nós recebe o nome de *Sharding*, que consiste em dividir os dados e move-los para outro nó ou local, o que melhora o desempenho.

Existem muitas razões para que todas as partes do teorema não possam ser atendidas em um sistema distribuído. A maioria tem a ver com o volume de dados e o tempo que demora em mover os dados de verificação. CAP geralmente é

usada para justificar o uso de modelos de fraca consistência. Muitas ideais do CAP evoluíram para ideia de BASE.

4.2. BASE

Como na química, base é considerada o oposto do ácido, em inglês *ACID,* BASE *Basically Available, Soft state, and Eventual consistency*. (Basicamente disponível, Estado leve e Eventualmente consistente), foi usado para quando se trabalha com grandes sistemas distribuídos que mantém os princípios do teorema de CAP. (Mishra 2014).

Um sistema estar basicamente disponível significa que ele responderá qualquer requisição, com exceção quando ocorre um erro para obter resposta ou os dados podem estar em estado inconsistente ou de mudança.

A ideia de estado leve significa que o sistema esta em mudança constante. Isso ocorre devido à consistência eventual, onde é normal o sistema sofrer alterações mesmo após o fim das inserções.

Eventual consistência diz respeito ao conceito de que quando o sistema para de receber inserções, os dados serão distribuídos para os locais que pertencem no sistema. A vantagem disso é não precisar verificar a consistência em cada transação, como ocorre nos sistemas ACID.

4.3. Modelos de dados

Os bancos de dados NoSQL são divididos em quatro categorias básicas:

Baseados em chave-valor: estes sistemas armazenam valores e um índice para encontrá-los numa chave programada. Temos o *Redis* e o *Couch DB* baseados nesse modelo.

Orientandos a documentos: armazenam documentos. Estes são indexados e é fornecido um sistema de pesquisa simples. *MongoDB* e o *Couchbase* seguem esse modelo.

Orientados a colunas: Armazena cada registro em uma coluna, e cada registro possui um identificador virtual para estabelecer relacionamento com outras colunas. Temos o *Cassandra* e o *HBase*, que são baseados nesse modelo.

Baseados em grafos: As entidades recebem o nome de vértices ou nós, que são ligadas por arestas, cada uma podendo guardar dados entre os relacionamentos e cada relacionamento pode ter uma direção. *Neo4j* e o *Titan Graph DB* seguem esse modelo.

Com tantas opções e categorias, a questão mais importante é qual, como, e porque escolher um. Cada categoria de banco de dados NoSQL foi criada para solucionar uma gama de problemas específicos. Assim, não é possível que um único banco de dados consiga solucionar todos os problemas do mercado. Foi esse cenário que favoreceu o surgimento dos bancos de dados NoSQL. Portanto, antes de escolher um banco de dados, é importante primeiramente entender as necessidades.

5. Apache Cassandra

Foi desenvolvido pelo Facebook em 2008 por Avinash Lakshman e Prashant Malik, como parte da ferramenta de busca de mensagens da rede social. Ainda nesse mesmo ano foi lançado como um projeto de código aberto, e no ano seguinte passou a fazer parte da Fundação Apache como um projeto da incubadora. Em fevereiro de 2010, se tornou um projeto de nível superior. Atualmente o Cassandra está em sua versão 3.0.

Seu nome vem da mitologia Grega, Cassandra era uma linda mulher pela qual o deus Apollo se apaixonou, e lhe concedeu o dom da profecia em troca de um beijo, porém como Cassandra sabia o que aconteceria, quando Apollo foi beija-la ela cuspiu na boca dele. E esse nome foi escolhido, pois Cassandra era capaz de saber o que ia acontecer. Que de certa forma tem a ver com a maneira de poder usar o Cassandra para construir um produto melhor por ter uma compreensão acurada do que esta acontecendo ao seu redor. (Bradbarry; Lubow, 2014).

Lakshman e Malik (2008) descrevem seu sistema com as seguintes palavras:

> Cassandra é um sistema distribuído de armazenamento para gerenciar grandes quantidades de dados estruturados espalhados por muitos servidores, oferecendo alta disponibilidade sem um ponto de falha. Cassandra tem o objetivo de rodar em uma infraestrutura de centenas de nós (com a possibilidade de se espalhar por diferentes data centers). Nesta escala, pequenos e grandes componentes falham continuamente. A confiabilidade e escalabilidade dos sistemas de software dependem da maneira que o Cassandra gerencia o estado de persistência em face dessas falhas de unidades. Cassandra não tem suporte para um sistema com modelo de dados relacional, Embora em muitos aspectos o Cassandra se assemelhe com um banco de dados e compartilha muito o projeto e estratégias com os mesmos, em vez disso, oferece aos clientes um modelo de dados simples que suporta um controle dinâmico sobre o layout e formato dos dados. O sistema Cassandra foi projetado para rodar em uma simples infraestrutura de hardware e lidar com gravações de alto rendimento enquanto não sacrifica a eficiência de leitura.

O Cassandra é um poderoso banco de dados NoSQL de código aberto, e pertence ao modelo orientado a coluna. É altamente escalável e foi projetado para gerenciar sem pontos de falhas quantidades massivas de dados em tempo real distribuídos entre vários servidores. Possui a finalidade de manusear grandes quantidades de dados distribuídos entres vários servidores, sem perder disponibilidade.

A natureza da arquitetura descentralizada garante que não existam pontos de falhas e permite que todos nós de um *cluster* desempenhem o mesmo papel. Isto é, todos nós de um *cluster* pode responder a qualquer pedido. Como os dados são replicados automaticamente entre os nós, caso ocorra uma falha em um dos nós, ele pode ser substituído. Além disso, novos nós podem ser adicionados sem que o sistema pare de trabalhar. Sua estratégia de replicação pode ser configurada de acordo com a necessidade, podendo funcionar com uma arquitetura centralizada ou distribuída, ser flexível quanto a redundância ou se necessário fazer um controle rígido.

O Cassandra possui suporte para a maioria das linguagens de programação, além de oferecer uma linguagem própria de modelagem e consulta

chamada CQL - *Cassandra Query Language* (Linguagem de Consulta Cassandra), a qual possui algumas semelhanças com a SQL, para contornar o problema dos bancos NoSQL, de não possuir uma linguagem padronizada. O uso do Cassandra ao redor do mundo está aumentando com o passar dos anos. Empresas como *Netflix*, *eBay*, *Twitter* e *Reddit* usam o Cassandra em seus sistemas.. O maior *cluster* de conhecimento público possui 300 TB de dados distribuídos em 400 máquinas. (Bradbarry; Lubow, 2014).

5.1. *Schema-less* (Sem esquema)

Cassandra faz parte do modelo baseado em colunas que é considerado sem esquema. Ou seja, que não é necessário criar um esquema com antecedência, Quando desejar acrescentar uma coluna, só precisa estabelecer o nome da coluna e gravar, e a coluna será criada caso o nome não tenho sido usado anteriormente. Assim linhas bem longas podem ser criadas, com capacidade para milhões de colunas. Além disso, as linhas não são obrigadas a ter dados em todas as colunas que a mesma tabela possui. (Bradbarry; Lubow, 2014).

Apesar de não dar opção para criar esquemas, no caso de sabermos como será a estrutura de dados, podemos estabelecer nomes das colunas e o tipo específico que ela vai conter. Isso também permite que possamos atribuir índices secundários paras as colunas que foram criadas.

5.2. Componentes

Antes de aprofundar em detalhes específicos de como o Cassandra funciona é importante definir alguns componentes e termos frequentemente mencionados.

Nós: Local onde os dados permanecem armazenados é o componente básico da infraestrutura do Cassandra.

***Data Center*:** Uma coleção de nós relacionados dentro de um mesmo cluster. Um *data center* pode estar em uma máquina física ou em uma máquina virtual.

***Racks*:** São agrupamentos de nós dentro de um *data center*. Durante a replicação dos dados, as cópias devem ser armazenadas em *racks* diferentes.

***Cluster*:** Um *cluster* é um conjunto que pode ter um ou mais *data centers* distribuídos por vários locais do mundo.

Replicação: É o processo que copia os dados e armazena em vários nós com intuito de garantir a disponibilidades e a tolerância a falhas. A replicação é configurada através do fator de replicação que determina a quantidade de cópias que serão realizadas no *cluster*.

***Keyspace*:** É um conjunto de dados semelhante ao esquema do modelo relacional, com a diferença que o *keyspace* possui informações da forma que os dados serão replicados pelos *clusters*. De modo geral é criado um *keyspace* para cada aplicação.

Família de colunas: São tabelas que agrupam colunas de forma ordenada através do nome da coluna. As famílias de coluna não possuem uma estrutura pré-definida.

Colunas: Cada coluna é identificada por um nome, e contém um valor e um *timestamp* que são gravados através da aplicação. Uma coluna pode ser inserida em tempo de execução, só precisa informar um valor para essa nova coluna em uma linha. A coluna é a menor unidade para armazenar dados no Cassandra.

Linhas: São colunas que possuem uma mesma chave primária. Diferente do modelo relacional que aloca espaço para todas as colunas de uma linha o valor armazenado é *NULL*, no Cassandra só é alocado espaço para colunas presentes em cada linha.

***Token*:** Cada *Token* determina a posição dos nós, e sua porção de dados de acordo com um valor *Hash*. .

***Peer-to-peer* (ponto-a-ponto):** É um modelo de arquitetura de redes onde todos os pontos ou nós funcionam ao mesmo tempo como cliente e como servidor, tornando possível o compartilhamento de serviços e dados sem a necessidade de um servidor central.

Protocolo *Gossip*: É um protocolo de comunicação *peer-to-peer*, para descobrir e partilhar a localização e informações sobre o estado de outros nós do *cluster*. O protocolo também é mantido localmente por cada nó para usar quando um nó e reiniciado. Os nós trocam informações sobre eles e sobre os outros nós, assim todos nós descobrem rapidamente sobre todos nós do cluster. Cada informação compartilhada tem uma versão associada, quando ocorre uma mudança de estado à versão é atualizada.

5.3. Modelo de dados do Cassandra

O Modelo de dados do Cassandra é similar ao *Google Bigtable*[2], que os dados são gravados de forma orientada a colunas. Isso vai contra o modelo relacional que o armazenamento é realizado de maneira orientada a linhas. O modelo orientado a colunas permite que os dados sejam salvos de maneira eficaz. Nele a gravação de valores nulos é dispensada, portanto, quando não existir valor para uma coluna a mesma não é gravada. Assim evita o desperdício de espaço com o armazenamento de valores nulos.

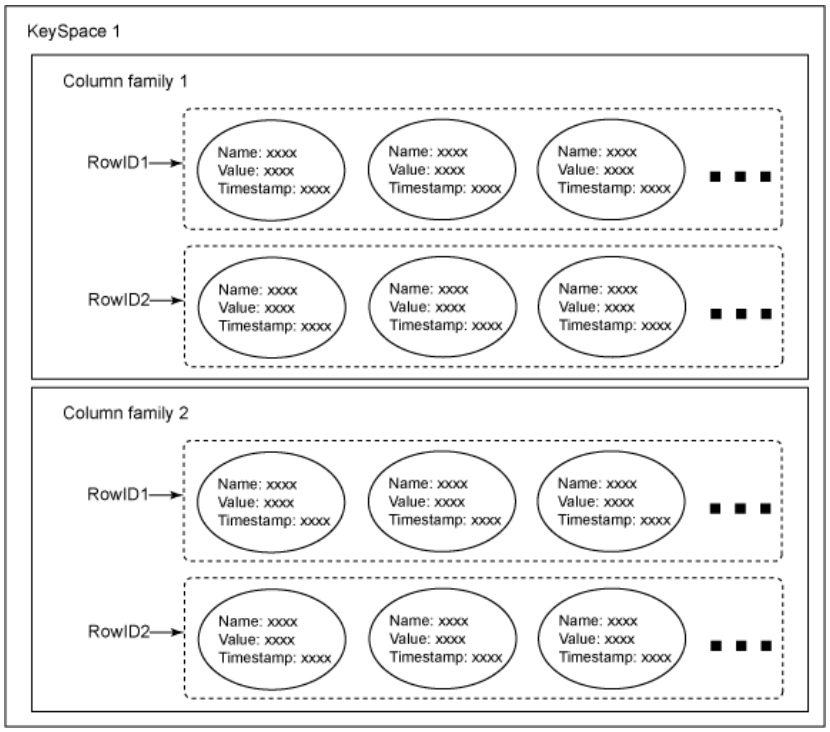
Cada registro pode ser considerado um conjunto de chave/valor, que o registro é encontrado através de sua chave, que pode ser chamada de chave primária, ou *row-key* (chave de linha). Cada registro é ordenado e armazenado com base em sua *row-key*. Lakshman e Malik (2008) descrevem o modelo com as seguintes palavras em seu artigo:

> Uma tabela no Cassandra é um mapa distribuído multidimensional indexado por uma chave. O valor é um objeto que é altamente estruturado. A row-key em uma tabela é uma sequencia sem restrições de tamanho, embora tipicamente com tamanho de 16 a 32 bytes. Todas as operações com uma única row-key são atômicas não importando quantas colunas são lidas ou escritas. As colunas são agrupadas em conjuntos chamados de famílias de

[2] O BigTable é um banco de dados orientado a colunas criado pelo Google

colunas, semelhante ao que ocorre no sistema Bigtable. Cassandra apresenta dois tipos de família de colunas, família Simples e Família de super colunas. Famílias de Super colunas podem ser visualizadas como uma coluna de famílias dentro de uma coluna de família.

Figura 6 - Representação do modelo orientado a colunas.



(Mishra 2014), (Adaptado).

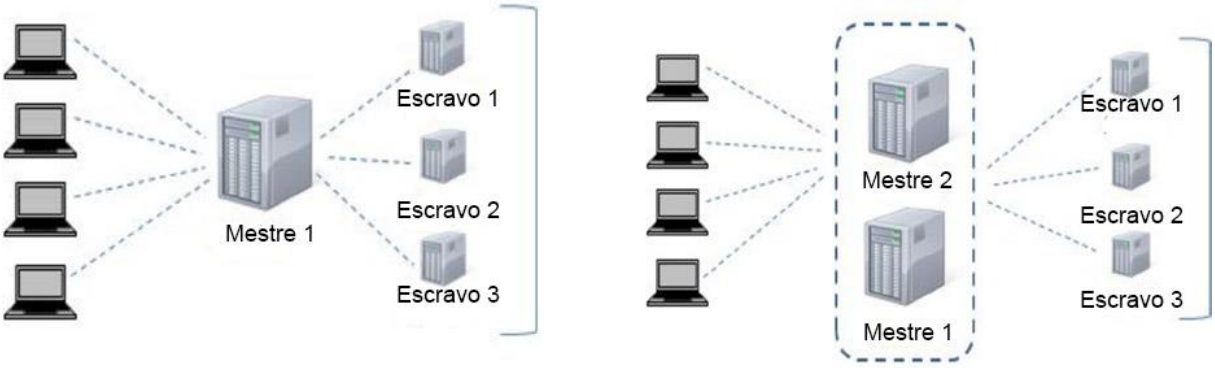
Como pode ser observado na figura 6, além do valor, cada registro possui um campo *timestamp* (carimbo de tempo). É através desse campo que o Cassandra escolhe qual é a versão mais recente do dado. Esse valor é atribuído pelo próprio sistema e possui a precisão de milissegundos, ou seja, um segundo dividido por mil.

5.4. Principais características

Distribuído e descentralizado

Os SGBDs baseados no modelo relacional armazenam os dados em sistemas centralizados ou em uma arquitetura mestre/escravo, ou seja, os recursos ou o mestre estão disponíveis em uma única máquina. Como demonstrado na figura 7, todo sistema está centralizado no mestre.

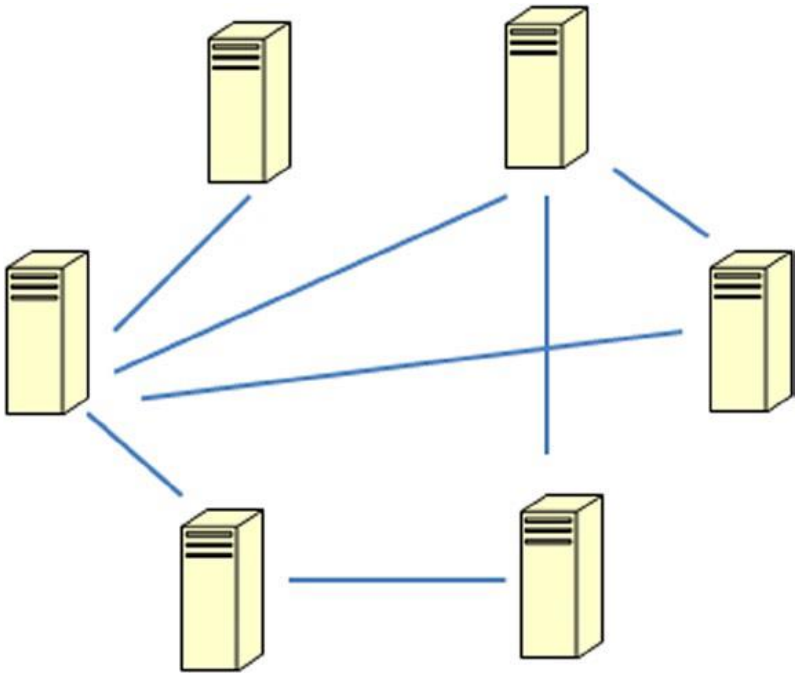
Figura 7 - Representação arquitetura Mestre/Escravo.



(Mishra 2014), (Adaptado).

A arquitetura do Cassandra é do tipo *peer-to-peer* (ponto para ponto), onde cada nó em um cluster possui a mesma importância. Cada nó permanece independente do outro e os dados são replicados em um ou mais nós. Tais características fazem com que o Cassandra seja um banco de dados descentralizado e distribuído. A figura 8 apresenta a maneira com que a arquitetura *peer-to-peer* funciona. (Mishra 2014).

Figura 8 - Representação arquitetura *peer-to-peer*.



(Mishra 2014), (Adaptado)

Escalabilidade elástica

O Cassandra possui escalabilidade horizontal, ou seja, oferece a possibilidade do sistema crescer de acordo com o aumento da demanda. Esse processo é realizado através do aumento da quantidade de nós em um *cluster*, em outras palavras acrescentam-se máquinas no sistema para aumentar o poder de processamento com a vantagem de ser um procedimento relativamente simples e com custos reduzidos. (Mishra 2014).

Disponibilidade e tolerância a falhas

Em um sistema centralizado ou com a arquitetura mestre/escravo, todos os recursos se tornam dependente de uma única máquina. Portanto no caso dessa máquina parar todo acesso ao serviço de banco de dados será comprometido.

Na arquitetura descentralizada e distribuída usada pelo Cassandra os dados são replicados pelos nós de um *cluster*. E no caso de falha em um dos nós, a requisição é completada por outro nó onde os dados foram replicados. Garantido assim que o sistema permaneça disponível e possua tolerância a falhas. (Mishra 2014).

Consistência

Diferente dos SGBDs baseados no modelo relacional o Cassandra não segue as propriedades “ACID”, ele possui apenas as características “AID”, ou seja, os dados armazenados são atômicos, isolados e duráveis. O “C” de “ACID” não está presente no Cassandra, conceitos como chave-estrangeira não existem. A consistência dos dados é configurável em *cluster*, isto significa que o administrador que escolhe se a consistência será mais forte, onde todos os nós respondem, ou eventual, quando apenas um nó responde, e os outros são atualizados eventualmente. Dessa forma o administrador configura a consistência de cada transação, podendo fazer com que todos os nós de um *cluster* respondam a uma transação, e em outras a consistência seja espalhada eventualmente.

Alta performance

O banco de dados Cassandra faz uso do multiprocessamento para ter um alto desempenho. Assim é capaz de suportar centenas de *terabytes* mantendo altas velocidades de escrita e leitura.

5.5. Replicação de dados

Como forma de garantir a disponibilidade e a tolerância a falhas, o Cassandra precisa saber a forma que os dados serão replicados e distribuídos através dos nós do *cluster*, onde serão armazenados e quantas cópias serão feitas. A maneira que a replicação de dados será realizada é determinada por configurações como o fator de replicação, estratégia de replicação, o *snitch* e o particionador. (Mishra 2014).

5.5.1. Fator de replicação

O fator de replicação diz respeito ao número de cópias que será feita de cada linha. Por exemplo, se o fator de replicação é três, serão gravadas três cópias idênticas de cada linha em nó diferente do *cluster*. E cada cópia possui a mesma importância que as outras. Uma regra básica, mas muito importante ao estabelecer o fator de replicação é de que o valor nunca pode ser maior que o número de nós. Caso isso ocorra as escritas serão rejeitadas, portando não adianta acreditar que usando um fator de replicação alto, os dados estarão seguros. (Mishra 2014).

5.5.2. Estratégia de replicação

A estratégia de replicação define a maneira que o Cassandra se comporta durante a escrita e leitura de dados. Existem duas opções de estratégia *SimpleStrategy* e *NetworkTopologyStrategy*.

SimpleStrategy

É usada quando o Cassandra é em apenas uma máquina, ou em um *cluster* com um único *data center*. Essa é a estratégia padrão do Cassandra. Ela

trabalha armazenando a primeira cópia em um nós escolhido pelo particionador e as cópias seguintes são armazenadas no próximo nó seguindo o sentido horário. (Mishra 2014).

NetworkTopologyStrategy

É usada em situações que o Cassandra está implantado em vários *data centers*, essa estratégia determina o número de cópias que serão realizadas em cada *data center*. Esse número é definido pelo fator de replicação estabelecido momento de criação do *keyspace*. (Mishra 2014).

5.5.3. *Snitchs*

Snitch é um protocolo para mapear os endereços IPs de *racks* e *data centers*. Ele cria um mapeamento dos agrupamentos de nós para ajudar a localizar o lugar que os dados serão lidos. Em função de escrita não é necessário usar *snitch*, pois os dados são enviados para um nó e depois replicados para outros nós. Quando uma requisição de leitura é feita, a aplicação encaminha esse pedido para um único nó, e o *snitch* define qual nó irá responder a requisição, baseando em um histórico de desempenho dos nós. (Brown 2015).

Existem algumas variedades de *snitchs* que podem ser usados no Cassandra.

SimpleSnitch

É o tipo de *snitch* padrão e também o mais simples, pois ele não reconhece *data centers* ou *rack*, portanto não são necessárias muitas configurações. Esse tipo de *snitch* é usado em aplicação com um único *data center*. É importante lembra que quando esse modelo é usado a fator de replicação deve ser sempre um.

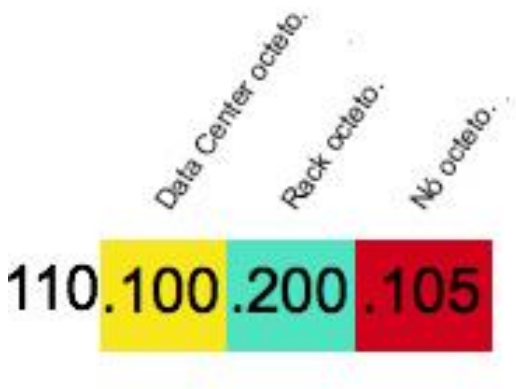
DynamicSnitch

É uma camada que existe em todos outros modelos de protocolo *snitch*, exceto no *simplesnitch*. Ele oferece ao *snitch* escolhido uma camada adicional para determinar o melhor local de leitura possível. O *dynamicsnitch* determina a melhor abordagem através de cálculos de latência. Esse modelo é o recomendado na maioria dos casos com múltiplos *data centers*.

RackInferringSnitch

Funciona nos casos em que é conhecida a topologia da rede, onde cada nó é identificado por um IP. Os endereços de IP dos nós são compostos por quatro octetos como exposto na figura 9.

Figura 9 - Composição do IP de um nó.



(Mishra 2014), (Adaptado)

O primeiro octeto 110 não tem significado especial.
O segundo octeto 100 representa o octeto do *data center*.
O terceiro octeto 200 representa o octeto do rack.
O último octeto 105 representa o octeto do nó.

Ec2Snitch

É usado somente para aplicação baseadas em AWS – *Amazon Web Services,* onde o *cluster* está inteiro em uma única região. Cada região é considerada um *data center* e as áreas disponíveis são considerados rack do *data center*.

Ec2MultiRegionSnitch

Também é usado somente em implementações *Amazon Web Service,* porém nesse caso o *cluster* está espalhado por múltiplas regiões. Os endereços privados de cada nó são usados para ter comunicação com todas as regiões.

PropertyFileSnitch

Assim como os modelos anteriores, o *PropertyFileSnitch* ajuda a determinar a localização dos nós através de racks e *data center*. A diferença é que a configuração é determinada pelo usuário através do arquivo chamado *cassandra.topology.proprietes*. Porém em casos de um *cluster* muito grande a configuração pode ser trabalhosa.

GossipingPropertyFileSnitch

Define o *data center* e o rack de um nó local e usa o protocolo *grossiping* para distribuir essas informações para os outros nós.

5.5.4. Particionador

Determina como os dados são distribuídos pelos nós do *cluster*. O particionador é uma função que atribui um *Token* que representa cada linha com uma chave de partição, então cada linha é distribuída no *cluster* pelo valor do *Token*, é importante que cada linha possa caber em um único nó independe do fator de replicação escolhido. Existem três opções de particionador que podem ser usadas:

ByteOrderedPartitioner

Foi um dos primeiros particionador disponível no Cassandra e atualmente não é recomendado usa-lo. Ele é usado para particionar os dados solicitados. Sua principal vantagem é que pode fazer buscar pela chave primaria. Uma grande desvantagem ao usar esse particionador é que ele não realiza o balanceamento de carga de forma eficiente. (Brown 2015).

RandomPartitioner

Era o particionador padrão até a versão 1.2 do Cassandra. Ele usa o valor em MD5 *Hash* da linha para distribuir os dados pelo *cluster*. Se estiver usando nós virtuais não é necessário calcular o *token*. Outro beneficio é que esse tipo de particionador realiza a paginação das linhas usando a função de *token*. (Brown 2015).

Murmur3Partitioner

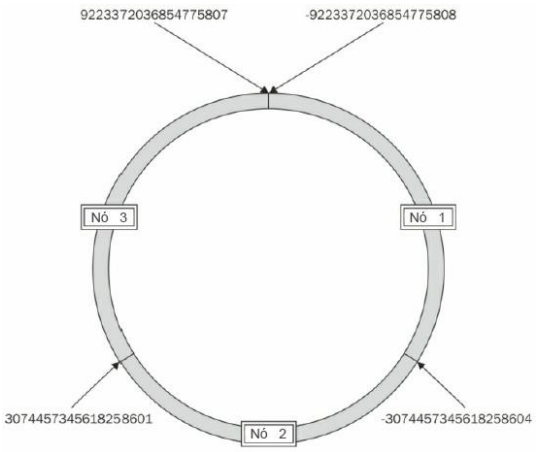
É o particionador padrão quando se cria um *cluster* Cassandra. Ele oferece a função de *hashing* chamada *MurmurHash,* que possui alta velocidade de resposta. Essa função cria um *Hash* de 64 bits da chave da linha. Assim como o *RandonPartitioner* é possível realizar a paginação das linhas. (Brown 2015).

Uma vez que um particionador for escolhido, é necessário usar esse mesmo particionador.

5.6. Nós Virtuais – *Vnodes*

Em versões anteriores ao Cassandra 1.2 era permitido atribuir apenas um *Token* para cada nó de um *cluster*. Isso mudou a partir da versão 1.2 em que permite que cada nó possua muitos *tokens*. Esse paradigma é chamado de nó virtual ou *Vnode* que permitem que cada nó tenha uma grande quantidade de pequenas partições distribuída pelo *cluster*. Os *tokens* no Cassandra são valores inteiros de 64 bits, então o menor *hash* possível será de -9.223.372.036.854.775.808 e o maior será de 9.223.372.036.854.775.808. É comum representar um *cluster* como um anel, e cada intervalo de *tokens* corresponde a uma parte da circunferência (Brown 2015). A figura 10 exibe um *cluster* com três nós sem o uso de *Vnodes*.

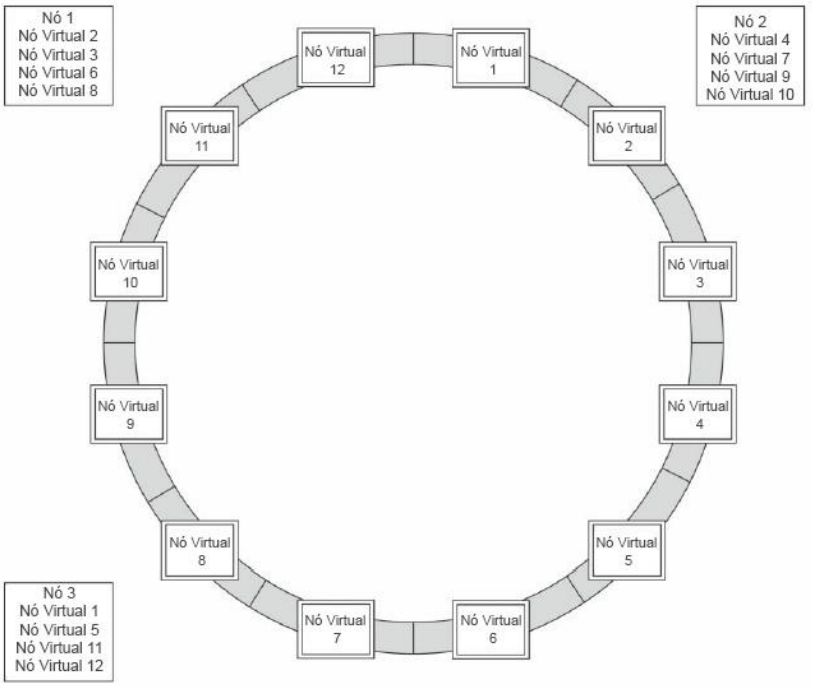
Figura 10 - Representação de um *cluster*.



(Brown 2015), (Adaptado)

A quantidade de *Vnodes* dentro de um nós é definido pelo desenvolvedor. Os nós virtuais são considerados um nó real pelo particionador, assim a adição e remoção de nós não requer o reequilíbrio do *cluster*. Quando um novo nó é adicionado, fica encarregado por uma parte dos dados armazenados nos outros nós. Ao remover um nó de um *cluster*, seus dados serão redistribuídos em partes iguais entre os nós que permanecerem. A figura 11 exibe um *cluster* de três nós físicos e a distribuição de doze nós virtuais.

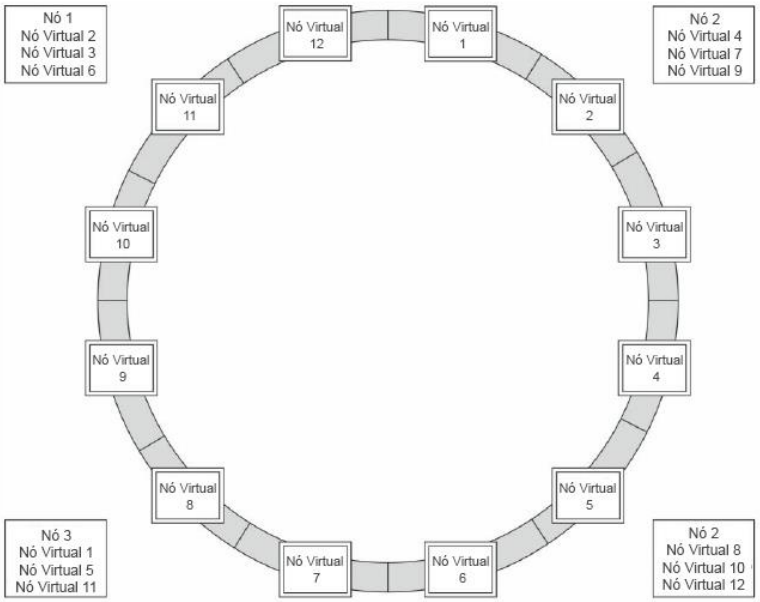
Figura 11 - Representação de um *cluster* com três nós físicos.



(Brown 2015), (Adaptado)

A figura 12 exibe o mesmo *cluster* e a maneira que os nós virtuais são redistribuídos quando se adiciona mais um nó físico. Podemos observar que cada nó possuía quatro nós virtuais e após a criação de um novo nó físico cada nó passou a ter três nós virtuais.

Figura 12 - Representação de um *cluster* com quatro nós físicos.

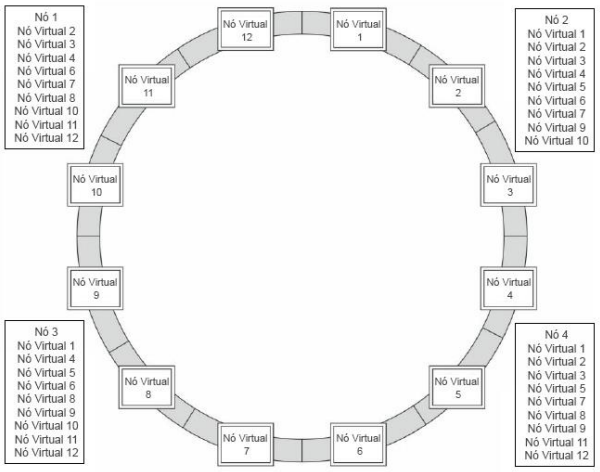


.

(Brown 2015), (Adaptado)

O uso de *Vnodes* aumenta desempenho no armazenamento de linhas com menor comprimento, e permite estabelecer um número de *Vnodes* proporcional a capacidade de cada máquina, com isso é possível usar máquinas com configurações diferentes sem perder eficiência. Caso um nó fique sem conexão, sua reconstrução fica mais rápida graças ao outros nós do *cluster*. A figura 13 demonstra como é feita a replicação dos dados com o uso de *Vnodes*.

Figura 13 - Representação da replicação de nós virtuais em um *cluster*.



(Brown 2015), (Adaptado)

Cada nó virtual é mantido em três nós físicos diferentes garantindo assim que no caso de falha em um dos nós físicos, outros dois possuem os dados. Até mesmo no caso de falha em dois nós físicos simultaneamente a distribuição dos nós físicos garante que todos dados permaneceram disponíveis.

5.7. CQL

Toda arquitetura do Cassandra o torna uma ferramenta muito eficiente e confiável para armazenar dados, e para utilizar todos esses recursos sem perder desempenho, é necessário uma maneira simples de manipular esses dados armazenados. Devido às diferenças entre o Cassandra e os bancos de dados relacionais, não era possível usar a SQL.

Antes o Cassandra utilizava o *Thrift*, que é uma IDL – *Interface Description Language* (Linguagem de descrição de interface), que possibilita a comunicação entre componentes escritos em linguagens diferentes. Porém o uso dessa ferramenta tornava difícil o uso do o Cassandra, e surgiu a necessidade de uma solução que fosse mais simples de usar e não precisasse de tantas atualizações. Então a solução encontrada foi criar uma linguagem mais parecida possível com a SQL. Essa nova linguagem foi chamada de CQL – *Cassandra Query Language* (Linguagem de consulta Cassandra). Atualmente a CQL está em sua versão 3.3.

Além de o nome ter sido inspirado na SQL, muitos comandos existentes na CQL são equivalentes com comandos SQL. Com tudo, CQL e SQL não são completamente compatíveis. Como por exemplo, CQL não possui recursos como instruções **JOIN**, uma vez que não é possível no Cassandra. Com tudo CQL não é apenas uma parte da SQL, instruções para atualizar o registro de tempo de uma coluna, estão disponíveis na CQL, e não possui uma função equivalente na SQL.

5.8. Segurança

O Cassandra utiliza três recursos de segurança, a encriptação na transmissão de dados, autenticação de usuários e gerenciamento de permissões

5.8.1. Criptografia

Para garantir a segurança dos dados enquanto trafegam de um nó para outro, ou do usuário para o nó o Cassandra usa o protocolo de criptografia SSL – *Secure Socket Layer.* Ele assegura que os dados não sejam comprometidos e que a transferência seja segura. É gerado um certificado SSL para cada nó em um *cluster*, e cada nó deve possuir os certificados dos outros nós. Isso evita que ocorram acessos indevidos e que os dados sejam interceptados durante trafego.

5.8.2. Autenticação

A segurança dos acessos é feita através de um controle interno de contas de acesso. O administrador cria contas para os usuários que são autenticados em cada acesso ao *cluster*. Contas de usuário são criadas com CQL através do comando **CREATE USER**.

5.8.3. Permissão

Após ter o acesso autenticado, existe a gestão de permissões que define quais as funções que cada usuário tem permissão para executar. O gerenciamento de permissões é feito com os comandos **GRANT/REVOKE**, similar aos comandos SQL.

5.9. Tipos de dados

Na criação de tabelas é preciso definir qual o tipo de dados de cada coluna. A CQL suporta todos os tipos de dados suportados pelo Cassandra, a tabela 1 mostra os tipos dados e suas descrições:

Tabela 1 - Tipos de dados suportados pelo Cassandra

Tipo	Descrição
ascii	Conjunto de caracteres tipo ASCII.
bigint	Números inteiros entre -2^{63} e 2^{63}, utiliza 4 bytes.

blob	Valores expressos em hexadecimais, podendo ser textos ou pequenas imagens.
boolean	Indica true (verdadeiro) ou false (falso), utiliza 1 byte.
decimal	Números decimais com precisão variável.
double	Reais entre (aproximadamente) 10^{-4932} e 10^{4932}, utiliza 8 bytes, precisão de 15 dígitos.
float	Reais entre (aproximadamente) 10^{-38} e 10^{38}, utiliza 4 bytes, precisão de 7 dígitos.
inet	Endereço de IP. Podendo ser IPv4 com 4 bytes, ou IPv6 com 16 bytes.
int	Números inteiros entre -2147483648 e 2147483647, utiliza 4 bytes.
text	Textos no forma UTF-8.
timeuuid	Identificador UUID tipo 1 em conjunto com um *timestamp*.
timestamp	Valor de 64 bits que representa um número em milissegundos, apresentado como uma data.
uuid	Identificador UUID tipo 1 ou tipo 4.
varchar	Textos no forma UTF-8.

Mishra 2014, Adaptado.

5.10. Problema exemplo

Para apresentar os conceitos do Cassandra citados anteriormente, esse capítulo será dedicado a exemplificar, baseando-se na obra de Brown, 2015, a criação de um banco de dados com a finalidade de gerenciar os dados de uma aplicação com características de uma rede social. Em que usuários se cadastram, e publicam mensagens, e essas mensagens podem ser visualizadas por outros usuários, e os usuários podem seguir uns aos outros.

Por se tratar de uma rede social existe a necessidade de velocidades de escritas e leitura elevadas, além disso, é importante que tenha alta disponibilidade e tolerâncias a falhas para que o conteúdo esteja sempre disponível para acesso dos usuários. Também se faz necessário a capacidade de atender o crescimento, ou seja, possuir escalabilidade horizontal. Contando com esse crescimento e precisando atender milhões de acessos outra característica imprescindível é ter alta

performance. O Cassandra se mostra a melhor opção de banco já que apresenta todos os recursos para suprir a demanda do cenário exposto. (Brown 2015).

O primeiro passo é a criação de um *keyspace*, como visto anteriormente é similar ao esquema do modelo relacional, a principal diferença fica na configuração do fator de replicação e da definição do tipo de estratégia de replicação deverá ser adotada.

Figura 14 - Criação do *keyspace*

```
CREATE KEYSPACE “my_status”
WITH REPLICATION = {
    ‘class’ : ‘SimpleStrategy’, ‘replication_factor’ : 2
};
```

(Brown 2015), (Adaptado).

No exemplo acima o *keyspace* foi nomeado “my_status”, utilizando a *SimpleStrategy* para replicação de dados, já que todos nós encontram-se em um mesmo *data center*. Considerando que o *cluster* inicialmente conta com um *data center* com três nós, podemos definir o fator de replicação em dois. Assim, os dados armazenados estarão em dois nós diferentes.

Com a criação de *keyspace* tudo que for inserido dentro dele serão aplicadas as configurações definidas em sua criação. No caso de necessidade de alteração dessas configurações é possível alterar, usando o comando **ALTER KEYSPACE**. Como o *keyspace* definido é necessário selecionar qual *keyspace* será usado.

Figura 15 - Seleção do *keyspace*

```
USE “my_status”;
```

(Brown 2015), (Adaptado).

O comando **USE** diz para a Cassandra que todos os comandos seguintes serão realizados dentro do “my_space”.

Após a seleção do *keyspace* o próximo passo é definir tudo que será inserido e dividir em famílias de colunas, que são similares as tabelas do modelo relacional. Inclusive comando na CQL para criação de uma família de colunas pode ser **CREATE COLUMN FAMILY** ou simplesmente **CREATE TABLE**. Porém existem algumas diferenças:

- O Cassandra não tem o conceito de *NULL*, ou seja, se uma coluna não tem dados em uma linha, ela não existe naquela linha.
- No Cassandra não é possível estabelecer um valor *default* para uma coluna, se uma linha é inserida sem um valor para alguma coluna, não tem como estabelecer um valor padrão para uma coluna que seja inserida sem valor.
- O Cassandra não oferece validação de dados como comprimento máximo ou outro tipo de restrições. O valor máximo é o valor limite que cada tipo suporta.

Figura 16 - Criação da tabela “users”

```
CREATE TABLE “users” (
    “username” text,
    “email” text,
    “encrypted_password” blob,
    PRIMARY KEY (username)
);
```

(Brown 2015), (Adaptado).

No exemplo acima foi criada a tabela “users“, que possui três colunas, “username“ com tipo *text*; “email“ com tipo *text* e “encrypted_password“ com o tipo *blob*. a coluna “username“ foi definida com chave primária da coluna. Assim como nos SGBDs relacionais, a Cassandra não permite a inserção de novas colunas a partir da aplicação, porém é permitido atualizar o esquema de uma tabela através do comando **ALTER TABLE**.

Para o exemplo usado a primeira ação a ser realizada por um usuário é criar uma conta. No caso, ele será responsável por escolher um *username* (Nome de usuário), fornecer seu e-mail e uma senha. Caberá a aplicação validar os dados e

criptografar a senha. Assim que os dados estiverem prontos podem ser inseridos em uma nova linha da tabela “users“.

Figura 17 - Inserção de dados tabela “users”

```
INSERT INTO “users”
   (“username”, ”email”, ”encryppted_password”)
   VALUES (
         ‘alice’,
         ‘alice@gmail.com’,
         0x8914977ed729792e403da53024c6069a9158b8c4
);
```

(Brown 2015), (Adaptado).

O comando acima demonstra a inserção de um usuário no banco, a ordem dos valores deve ser a mesma em que as colunas foram especificadas, assim o *username* será “alice”, o e-mail “alice@gmail.com” e a senha é um valor hexadecimal que equivale a senha do usuário. Lembrando que o como o campo foi definido como *blob* só aceita valores hexadecimais.

Uma inserção também pode ser realizada parcialmente desde que a coluna definhada como chave primaria não esteja em branco, como no exemplo seguinte:

Figura 18 - Inserção de dados tabela na “users”

```
INSERT INTO “users”
   (“username”,  ”encryppted_password”)
   VALUES (
         ‘bob’,
         0x10920941a69549d33aaee6116ed1f47e19b8e713
);
```

(Brown 2015), (Adaptado).

Na inserção acima a coluna “email“ foi suprimida, portanto, esse registro existirá apenas a coluna “username“ com valor “bob” e a coluna com o valor em hexadecimal equivalente a senha. (Brown 2015).

Após a inserção de dados na tabela para dar sequencia ao exemplo proposto é necessário recuperar esses dados que foram gravados. Para isso o comando usado é o **SELECT**, muito conhecido por todos que já usaram SQL. O exemplo abaixo demonstra como selecionar os dados de um usuário.

Figura 19 - Seleção de dados da tabela "users"

```
SELECT * FROM “users”
WHERE ”username”  = ‘alice’;
username        |        email          |        encrypted_password
-----------------------------------------------------------------------------------------
alice               | alice@gmail.com  | 0x8914977ed729792e403da5...
```

(Brown 2015), (Adaptado).

O comando acima diz para o Cassandra retornar a linha na qual a coluna “username“ que também é chave primária possui o valor “alice“. O ‘ * ‘ diz que a resposta deve conter todas as colunas dessa linha. No exemplo seguinte é demonstrado como podemos escolher as colunas que serão exibidas.

Figura 20 - Seleção de dados da tabela "users"

```
SELECT “username”,”enctypted_password” FROM “users”
WHERE ”username”  = ‘alice’;

username        |        encrypted_password
-----------------------------------------------------------------------------------------
alice               | 0x8914977ed729792e403da53024c6069a9158b8c4
```

(Brown 2015), (Adaptado).

Para exibir todas as colunas de todos os usuários, basta retirar a clausura **WHERE**. Dessa maneira o Cassandra retornara todos os registros armazenados na

tabela “users”. No entanto se o banco possuísse mais de dez mil registros, seriam exibidos apenas dez mil. Esse limite existe apenas em comandos enviados a partir do terminal. Em consultas realizadas a partir de aplicação o limite pode ser estabelecido de acordo com o uso da aplicação e a linguagem utilizada.

Figura 21 - Seleção de dados da tabela “users”

```
SELECT * FROM  “users”;
username        |        email          |        encrypted_password
-----------------------------------------------------------------------------------------
bob               | NULL                | 0x10920941a69549d33aaee6...
alice             | alice@gmail.com  | 0x8914977ed729792e403da5...
```

(Brown 2015), (Adaptado).

Observando o resultado retornado pela consulta podemos notar duas situações, uma que aparece *NULL* na coluna “email“ do *username* “bob“, porém na verdade na estrutura do Cassandra essa coluna não existe para essa linha, e o resultado aparece como *NULL*, somente para um entendimento melhor do usuário. Já que o campo em branco poderia ser confuso. Outra ocorrência é que o registro “bob” aparece antes de “alice”. Isso ocorre porque o Cassandra retorna as linhas na ordem que estão armazenadas, essa ordem é determinada pelo *token* da chave primária de cada linha que é atribuído pelo particionador. No exemplo a seguir são exibidos os valores de cada *token*. (Brown 2015).

Figura 22 - Seleção de dados da tabela “users”

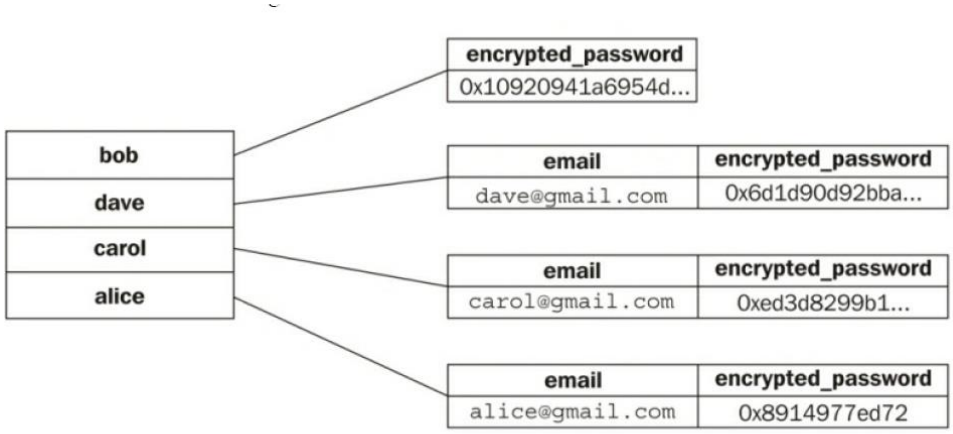
```
SELECT  “username” , token(“username”)
FROM  “users”

 username        |  token(username)
----------------------+---------------------
 bob               | -4224119183329226894
 alice             |  8347135061062854629
```

(Brown 2015), (Adaptado).

Para entender melhor o comportamento das colunas dentro de uma tabela. Podemos pensar na tabela como um conjunto de chaves em que cada chave aponta para uma linha, e cada uma dessas linhas contêm dados em um conjunto de colunas. Para facilitar o entendimento foram adicionados novos usuários na tabela “users”. A figura 14 ilustra a representação da tabela “users”.

Figura 23 - Representação dos dados da tabela "users"



(Brown 2015), (Adaptado)

A próxima etapa para dar sequencia no desenvolvimento do exemplo proposto é a criação de uma tabela com a finalidade de armazenar as mensagens publicadas pelos usuários. É importante que esses dados estejam vinculados ao usuário que escreveu, e que essas mensagens possam ser ordenadas em uma sequencia.

Figura 24 - Criação da tabela “user_status_updates”

```
CREATE TABLE “user_status_updates” (
    “username” text,
    “id” timeuuid,
    “body” text,
    PRIMARY KEY (“username”, ”id”)
);
```

(Brown 2015), (Adaptado).

A criação dessa tabela apresenta algumas diferenças com relação à tabela “users” que foi criada anteriormente. Como cada mensagem está vinculada a um usuário é imprescindível que a coluna “username” seja chave, porém se somente essa coluna fosse chave cada usuário poderia ter apenas uma mensagem, por isso essa tabela conta com uma chave composta, garantindo que cada usuário possa escrever varias mensagens. Essa chave composta identifica unicamente cada linha a partir da combinação das colunas “username” e “id”. A coluna “id” é do tipo *timeuuid,* que se trata de um UUID[3], que pode ser convertido para um tipo *timestamp.* Através dessa coluna o Cassandra é capaz de ordenar as linhas de acordo com a data que elas foram geradas. (Brown 2015).

Com a tabela de mensagens pronta já é possível que os usuários insiram mensagens com identificador único. E que essas mensagens sejam recuperadas e atribuídas corretamente ao seu criador, e ordenadas em uma linha temporal. O exemplo a seguir demonstra a inserção de mensagens de usuários diferentes.

Figura 25 - Inserção de dados na tabela “user_status_updates”

```
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
    ‘alice’, NOW(), ‘Learning Cassandra! ’ );
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
    ‘alice’, NOW(), ‘Alice Update 1’ );
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
    ‘alice’, NOW(), ‘Alice Update 2’ );
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
    ‘alice’, NOW(), ‘Alice Update 3’ );
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
    ‘bob’, NOW(), ‘Eating a taste sandwich’);
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
    ‘bob’, NOW(), ‘Bob Update 1’ );
INSERT INTO “user_status_updates”  (“username”, “id”, “body”) VALUES (
```

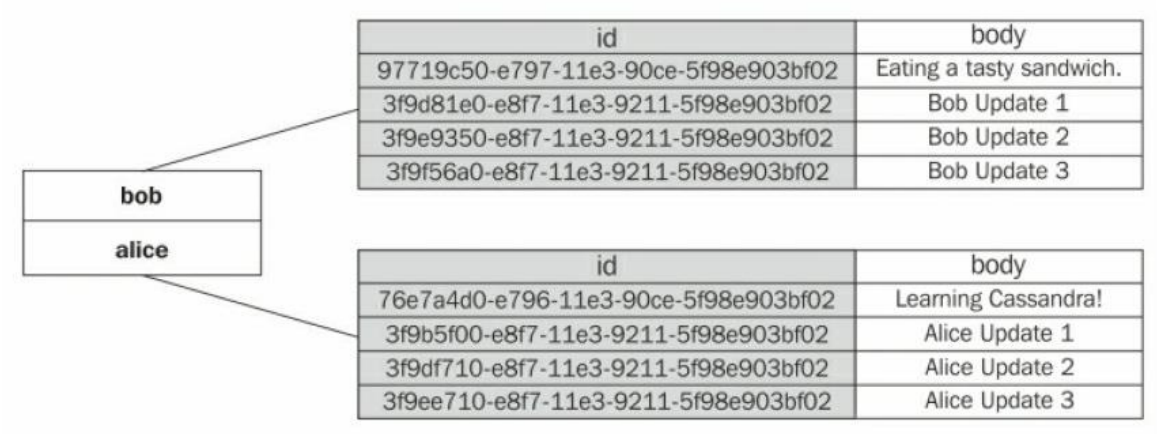
[3] UUID - *Universally Unique Identifies* (Identificador universal único) é um número extenso gerado de um jeito que garante que ele nunca se repita no mundo.

```
    'bob', NOW(), 'Bob Update 2' );
INSERT INTO "user_status_updates"  ("username", "id", "body") VALUES (
    'bob', NOW(), 'Bob Update 3' );
```

(Brown 2015), (Adaptado).

No exemplo acima são inseridas três mensagens de cada usuário, a diferença fica por conta da função "NOW()" responsável por atribuir o momento exato da inserção. Essa função é usada em casos que não é necessário saber qual UUID foi gerado. Caso a aplicação necessite saber o UUID gerado é melhor fazer uso de bibliotecas para gerar os UUID em nível de aplicação. Na figura 15 pode-se observar a representação da tabela com as mensagens.

Figura 26 - Representação dos dados da tabela "user_status_updates".



(Brown 2015), (Adaptado).

Agora com a possibilidade dos usuários se cadastrarem e postarem suas mensagens, outra função a ser inserida para seguir com o exemplo proposto é a possibilidade de um usuário poder seguir outros. Para isso é necessário criar uma tabela responsável por armazenar o relacionamento entre os usuários.

Figura 27 - Criação da tabela “user_follows”

```
CREATE TABLE “user_follows” (
    “followed_username” text,
    “follower_username” text,
    PRIMARY KEY (“followed_username”,  “follower_username”)
);
```

(Brown 2015), (Adaptado).

No exemplo apresentado é criada uma tabela “user_follows” que contêm uma coluna para o usuário que está sendo seguido “followed_username” e uma coluna para o seguidor “follower_username”, uma característica importante dessa tabela é que ela só tem duas colunas ambas compõem a chave primária, portanto nenhuma pode ficar sem dados. O exemplo seguinte demonstra a inserção de relacionamentos na tabela.

Figura 28 - Inserção de dados na tabela “user_follows”

```
INSERT INTO “user_follows”
    (“followed_username”, “follower_username”)
VALUES (‘alice’, ‘bob’);
INSERT INTO “user_follows”
    (“followed_username”, “follower_username”)
VALUES (‘bob’, ‘alice’);
INSERT INTO “user_follows”
    (“followed_username”, “follower_username”)
VALUES (‘alice’, ‘dave’);
INSERT INTO “user_follows”
    (“followed_username”, “follower_username”)
VALUES (‘dave’, ‘alice’);
```

(Brown 2015), (Adaptado).

A partir desses dados já é possível saber quem segue determinado usuário como no exemplo abaixo.

Figura 29 - Seleção de dados da tabela “follower_username”

```
SELECT “follower_username”
 FROM “user_follows”
WHERE “followed_username” = ‘alice’;

follower_username
-------------------------
bob
dave
```

(Brown 2015), (Adaptado).

O exemplo acima busca o conteúdo das colunas “follower_username”, onde a coluna “followed_username” tem o usuário “alice”. O resultado demonstra que “bob” e ”dave” seguem “alice”. Porém dessa maneira não é possível descobrir quais são os usuários que um usuário segue, pois a estrutura da chave primaria do Cassandra permite apenas buscas a partir da primeira coluna, ou com os valores de todas as colunas que compõem a chave. Por exemplo, não podemos realizar a seguinte busca, para saber que “alice” está seguindo.

Figura 30 - Seleção de dados da tabela “followed_username”

```
SELECT “followed_username”
 FROM “user_follows”
WHERE “follower_username” = ‘alice’;
```

(Brown 2015), (Adaptado).

Nessa busca não serão obtidos resultados, a solução para isso é a criação de um índice secundário, esse conceito é familiar para quem usa o modelo relacional, ou seja, um índice permite realizar buscas em colunas além da primeira.

Abaixo um exemplo da criação de índice para a coluna “follower_username”, e os resultado proveniente de uma busca pelo índice.

Figura 31 - Criação de índice pra a coluna “follower_username”

```
CREATE INDEX ON “user_follows” (“follower_username”);

SELECT “follower_username”
 FROM “user_follows”
WHERE “followed_username” = ‘alice’;

follower_username
--------------------------
bob
dave
```

(Brown 2015), (Adaptado).

Após a criação do índice a buscas passa a ter resultado e exibe que “alice” segue os usuário “bob” e ”dave”.

6. Conclusão

A evolução dos bancos de dados foi conduzida pelas necessidades de cada época. Uma característica dessa evolução era que as tecnologias que surgiam, substituíam quase que por completo as existentes, como foi o caso do modelo relacional que surgiu nos anos 70 e alguns anos depois se tornou o modelo predominante no mercado. Recentemente presenciamos uma nova etapa na evolução dos bancos de dados, com a demanda por armazenar de forma eficiente a enorme quantidade de dados provenientes de aplicações *web,* intitulada *Big Data,* o modelo relacional mostrou-se ineficiente, com isso novos modelos e paradigmas de armazenamento ganharam espaço para atender essa nova fatia de mercado, esses novos modelos ficaram conhecidos como NoSQL. Porém essa etapa não vem para substituir o modelo relacional, mas para coexistirem como completares, cada uma atendendo aos seus propósitos específicos.

Nesse novo mercado de banco de dados, o Cassandra vem ganhando destaque. Ele foi desenvolvido para atender de forma eficiente os requisitos para se trabalhar com o *Big Data*. Suas principais características garantem que ele possa crescer de acordo com a necessidade da aplicação, trabalhar de forma descentralizada, possuir tolerância a falhas e alta velocidade de escrita e leitura.

Para demonstrar de maneira pratica o funcionamento do Cassandra foi exposto nesse trabalho um cenário onde era necessário armazenar dados de uma rede social. E utilizando conceitos básicos foram construídas tabelas para armazenar usuários e mensagens além de estabelecer relacionamento entre os usuários. Por se tratar de uma aplicação com potencial de atingir milhares de usuários e milhões de mensagens, e com a necessidade de garantir que os dados estejam disponíveis e possam ser recuperados com alta velocidade; os bancos de dados relacionais não seriam capazes de atender esses requisitos de uma forma efetiva.

Há alguns anos atrás, na década de 90 principalmente, ao desenvolver uma aplicação não era necessário pensar muito, podia escolher os bancos de dados relacionais sem maiores dúvidas. Mas recentemente é imprescindível uma análise profunda dos requisitos antes de escolher um modelo de banco de dados, principalmente quando se tratar de uma aplicação *web.* Deve-se olhar atentamente

para essa nova fatia de mercado que vem crescendo os bancos de dados NoSQL. Outro motivo para se atentar aos requisitos é para não utilizar um modelo de forma inadequada, como por exemplo: implementar o Cassandra e tentar usar com um banco de dados relacional, pois pode trazer inúmeros problemas. Afinal se trata de uma alternativa e não um substituto ao modelo relacional.

Seguindo o estudo sobre modelos de bancos, uma opção interessante para trabalhos futuros seria a exploração de outros modelos de bancos de dados NoSQL.

7. Referências

BRADBERRY, Russell; LUBOW, Eric. **Practical Cassandra:** A Developer's Approach. New Jersey: Pearson Education, 2014. 168 p.

BROWN, Mat. **Learning Apache Cassandra.** Birmingham: Packt Publishing Ltd., 2015.

CASSANDRA, Planet. **Cassandra at CERN (Large Hadron Collider).** , 2013. Disponível em: <http://www.planetcassandra.org/blog/cassandra-at-cern-large-hadron-collider>. Acesso em: 18 mai. 2016.

LAKSHMAN, Avinash; MALIK, Prashant. **Cassandra - A Decentralized Structured Storage System.** , 2009. Disponível em: <http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf>. Acesso em: 12 out. 2015.

MISHRA, Vivek. **Beginning Apache Cassandra Development.** Nova Iorque: Apress, 2014.

NAVATHE, Shamkant B; ELMASRI, Ramez. **SISTEMAS DE BANCO DE DADOS.** 6. ed. São Paulo: Pearson Brasil, 2013.

OPPEL, Andy. **Databases A Beginner's Guide**. Nova Iorque: The Mcgraw-hill Education, 2009.

SEVERINO, Antônio Joaquim. **Metodologia do trabalho científico.** 23. ed. São Paulo: Cortez, 2014.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S.. **SISTEMA DE BANCO DE DADOS**. 6. ed. São Paulo: Elsevier, 2012.