FIAP ON

ARQUITETURA NOSQL

COLUMNAR-ORIENTED DATABASES (CASSANDRA)

MARCELO MANZANO E REGINA C. CANTELE

3

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

3 COLUMNAR-ORIENTED DATABASES (CASSANDRA)

3.1 Características

Os sistemas NoSQL do tipo família de colunas ou Wide Column Store são conhecidos por possuírem escalabilidade horizontal, de modo a conseguirem armazenar grandes quantidades de dados. Quase todos os sistemas deste tipo são altamente influenciados pela solução da Google denominada BigTabl – um sistema de armazenamento distribuído para gerir dados estruturados, desenhado com o intuito de conseguir escalar horizontalmente de forma natural, suportando quantidades muito elevadas de dados. Estes sistemas também são conhecidos por estarem intimamente relacionados com vários sistemas que recorrem a funções MapReduce.

Este tipo de sistemas NoSQL é muitas vezes considerado como um tipo específico do modelo chave-valor. Os sistemas de bases de dados que recorrem a este modelo definem a estrutura do valor (no par chave-valor) como um conjunto predefinido de colunas. De um modo pragmático, podemos encarar o modelo de dados destes sistemas como sendo uma tabela de baixa densidade, cujas linhas podem conter um número arbitrário de colunas. As chaves de cada linha (ou par chave-valor) fornecem um meio de indexação natural. As estruturas de armazenamento do tipo família de colunas possuem 4 componentes principais utilizados na sua estrutura: colunas, supercolunas, família de colunas e superfamília de colunas.

Figura 3.1 – Exemplo Família de colunas
Fonte: Adaptado por FIAP (2020)

A coluna representa a estrutura mais elementar deste modelo e é constituído por um par nome-valor. Qualquer número de colunas pode ser combinado para gerar uma supercoluna. Colunas e supercolunas são por sua vez armazenadas em filas. Quando uma fila possui apenas colunas, obtém a designação de família de colunas; se possuir na sua estrutura supercolunas, então a designação tomada é a de superfamília de colunas. Em um banco de dados orientado à coluna, todos os valores da coluna são armazenados juntos, como os valores da primeira coluna serão armazenados juntos, então os valores da segunda coluna serão armazenados juntos e os dados em outras colunas serão armazenados de maneira semelhante. Essa categoria de bancos de dados NoSQL tem:

- Uma chave (RowId) é uma identificação que tem um valor único usado para identificar um registro específico, semelhante à chave primária no modelo relacional.
- Um carimbo de data/hora (abreviado como timestamp) é um número inteiro usado para identificar uma versão específica de um valor de dados.
- Pelo menos um grupo de colunas com o formato “Família: Qualificador = Valor”, onde “Família” é o nome de um grupo de colunas, “Qualificador” é o nome de um qualificador de coluna e “Valor” é um nome real valor de um qualificador de coluna armazenado em texto.
- O nome de um grupo de colunas precisa ser definido quando a tabela é criada, mas o nome de um qualificador de coluna não.
- Os usuários podem encontrar o valor real dos dados por meio do valor de uma chave de linha específica, o nome de uma determinada família de colunas, o nome de um qualificador de coluna específico e o valor de um carimbo de data/hora específico.

3.2 Tecnologias

Muitas tecnologias estão associadas ao NoSQL colunar ou família de colunas, e o site Db-Engines classifica-os por sua popularidade atual. A popularidade usa parâmetros como número de menções do sistema em sites, interesse geral no sistema no Google Trends, frequência de discussões técnicas sobre o sistema, número de ofertas de emprego nas quais o sistema é mencionado em sites como Indeed e Simply Hired, número de perfis em redes profissionais mais populares como LinkedIn e Upwork, e relevância nas redes sociais como o número de tweets do Twitter.

11 systems in ranking, July 2020

Rank Jul 2020	Rank Jun 2020	Rank Jul 2019	DBMS	Database Model	Score Jul 2020	Score Jun 2020	Score Jul 2019
1.	1.	1.	Cassandra	Wide column	121.09	+2.08	-5.91
2.	2.	2.	HBase	Wide column	48.66	-0.07	-8.88
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	30.40	-0.40	+1.32
4.	4.	4.	Datastax Enterprise	Wide column, Multi-model	8.67	+0.32	-0.47
5.	5.	5.	Microsoft Azure Table Storage	Wide column	5.51	+0.24	+1.08
6.	6.	6.	Accumulo	Wide column	3.97	+0.12	-0.17
7.	7.	7.	Google Cloud Bigtable	Wide column	3.16	+0.22	+1.15
8.	8.	8.	ScyllaDB	Multi-model	2.56	-0.15	+0.87
9.	9.	9.	MapR-DB	Multi-model	0.68	+0.05	+0.04
10.	↑ 11.		Elassandra	Wide column, Multi-model	0.36	+0.08	
11.	↓ 10.	11.	Alibaba Cloud Table Store	Wide column	0.35	-0.01	+0.12

Figura 3.2 – Db-Engines NoSQL colunar
Fonte: DB-Engines (2020)

Agora, uma breve descrição do HBase e Microsoft Azure Cosmos DB.

3.2.1 HBase

O sistema HBase foi criado em 2007 por uma empresa de San Francisco denominada de Powerset. Este sistema, desde a sua criação, foi encarado como uma contribuição para o ecossistema Hadoop, geralmente abordado no Apache Framework Hadoop.

O Apache HBase é um projeto de fonte aberta que consiste numa base de dados NoSQL orientada para o armazenamento do tipo família de colunas, distribuída, tolerante a falhas e altamente escalável. Este sistema pode ser encarado como uma base de dados que está construída em cima do sistema HDFS, pertencente ao ecossistema Hadoop. Esta base de dados foi criada à semelhança do sistema BigTable apresentado pela Google. Desta forma, grande parte das suas funcionalidades e implementações apresentam uma enorme semelhança com aquelas dispostas pelo BigTable.

Este sistema é especialmente indicado para situações em que é necessário armazenar tabelas com volume elevados, esparsas e distribuídas por um grande número de servidores. Estas tabelas podem conter bilhões de linhas e milhões de colunas. Nesse sistema, os dados são armazenados em tabelas; cada tabela contém múltiplas linhas, mas um número fixo de colunas. Para cada linha podem existir vários “qualificadores” dentro de uma mesma família de colunas. Na

interseção entre “qualificadores” e linhas encontram-se as células das tabelas. Linhas são ordenadas por chaves de linha.

Figura 3.3 – Hbase Table
Fonte: Sinha (2019)

O sistema não possui nos seus elementos principais uma linguagem declarativa para realização de queries. O Hbase lida com falhas de forma completamente transparente para o utilizador. A escalabilidade é totalmente integrada no sistema. Novos nós podem ser adicionados aos recursos disponíveis em tempo real, sem necessidade de o administrador realizar processos de redistribuição ou de rebalanceamento. O sistema realiza essas tarefas automaticamente. Este sistema é ativamente utilizado e desenvolvido por várias organizações importantes com necessidades amplas de gestão de dados Big Data, como o Pinterest, Facebook, eBay e Yahoo!

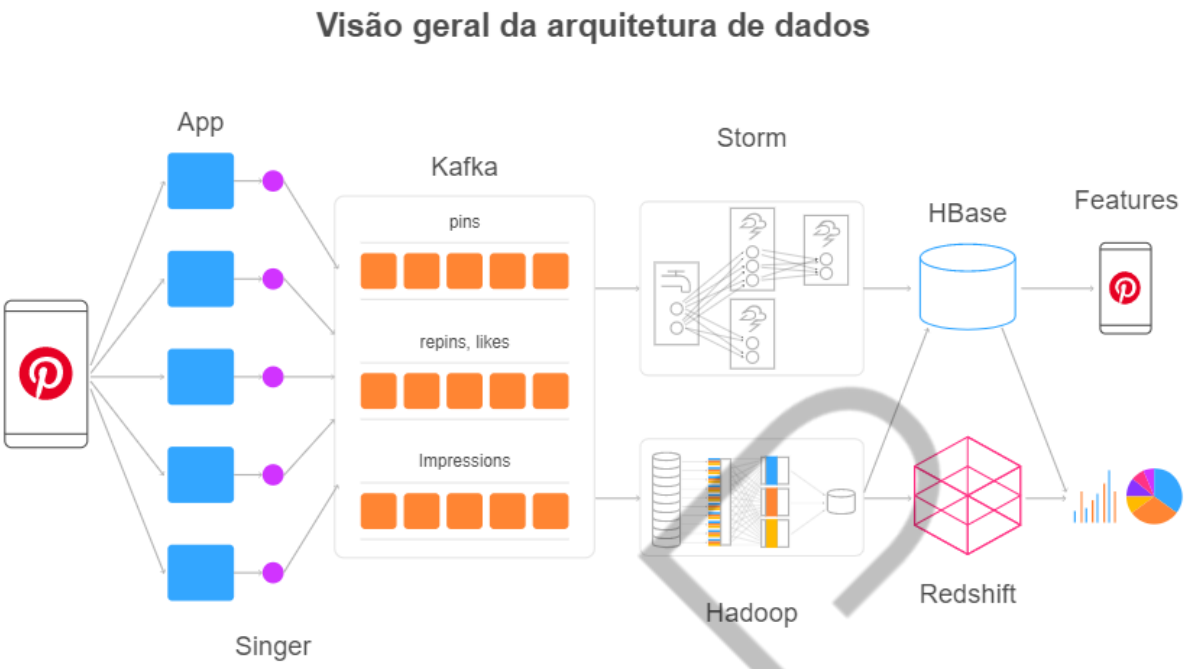

Figura 3.4 – Pinterest Data Architecture
Fonte: HBASE (2020)

3.2.2 Microsoft Azure Cosmos DB

Azure Cosmos DB é um serviço de banco de dados do Microsoft Azure projetado para qualquer aplicativo Web, aplicativo móvel, jogo ou aplicativo de IoT que exija processamento, leitura e gravação de uma grande quantidade de dados. Fornece aos aplicativos:

- Latência baixa, menos de 10 ms para cargas de trabalho de leitura e gravação devido ao uso de armazenamento com suporte SSD e replicação multimestre para seus dados onde quer que estejam os usuários, permitindo que eles se conectem à réplica mais próxima deles.
- Alta disponibilidade até 99,999% dos dados armazenados no Cosmos DB devido à capacidade de executar failover regional nos bancos de dados da sua conta Cosmos DB.
- Opções variadas de consistência – cinco opções de consistência bem definidas, incluem robustez forte e limitada, sessão, prefixo consistente e eventual que fornece flexibilidade total e baixa relação custo-desempenho.
- Opções avançadas de segurança para criptografia de dados em repouso e em movimento, além da autorização no nível da linha.

- Suporte para várias APIs para trabalhar com seus dados armazenados no banco de dados Cosmos, como a API do SQL Core, Cassandra, mongodb, Gremlin e Armazenamento de Tabelas do Azure, oferecendo uma maneira fácil de migrar o aplicativo para o Cosmos DB sem a necessidade para realizar mudanças significativas. Ele não requer gerenciamento de esquema ou índice, pois todos os dados serão indexados automaticamente; e não requer gerenciamento complexo de vários centros de dados ou implantações ou atualização de software de banco de dados.

3.2.3 Cassandra

O banco de dados Apache Cassandra é um banco de dados NoSQL distribuído, tolerante a falhas, escalabilidade linear, alta disponibilidade – sem nenhum ponto único de falha – e orientado a colunas. O Apache Cassandra é excelente para lidar com grandes quantidades de dados estruturados (a tabela tem colunas definidas) e semiestruturados (a linha da tabela não precisa preencher todas as colunas).

Pasmem, porém o Cassandra é escrito em Java e é usado principalmente para dados de séries temporais, como métricas, IoT (Internet of Things), logs, mensagens de bate-papo, e assim por diante. Ele é capaz de lidar com uma enorme quantidade de gravações e lê e escala para milhares de nós. Algumas características:

- **Open Source:** é um projeto de código aberto da Apache; o Cassandra pode ser integrado a outros projetos de origem, como Hadoop, Apache Pig, Apache Hive, entre outros.
- **Arquitetura ponto a ponto:** todos os nós no cluster e Cassandra se comunicam entre si; não há paradigma mestre-escravo.
- **Nenhum ponto único de falha:** em um cluster, todos os nós são criados iguais; os dados são distribuídos entre os nós do cluster e cada nó é capaz de lidar com solicitações de leitura e gravação.

- **Altamente disponível e tolerante a falhas:** devido à replicação de dados nos nós do cluster, o Cassandra é altamente disponível e tolerante a falhas.
- **Modelo de dados flexível:** os conceitos do DynamoDB e BigTable são integrados ao Cassandra para permitir estruturas de dados complexas.
- **Escalabilidade linear:** quando novos nós são adicionados, os dados são distribuídos de maneira mais uniforme entre os nós, o que reduz a carga de cada nó.
- **Consistência:** nível de consistência é o número de nós necessários para chegar a um acordo sobre os dados para leituras e gravações; o nível de consistência controla o comportamento de leitura e gravação com base em seu fator de replicação; o nível de consistência pode ser zero, um, quorum, local quorum ou todos.

Para garantir alta disponibilidade e replicação dos dados, o Cassandra usa um processo que se chama Gossip (fofoca, em tradução literal).

3.2.3.1 Protocolo Gossip

Os nós de computador (ligados em cluster) não estão muito atrás dos comportamentos dos humanos e dos vírus quando se trata de fofocar. Os fundamentos do protocolo gossip são os mesmos.

É a forma de espalhar uma informação para as pessoas do mesmo mundo. Quando você fica sabendo que seu gerente se demitiu, você repassa essa informação para o seu colega rapidamente. Seu colega se interessa pela informação e a repassa para outro colega, enquanto você divide essas informações com outro colega seu. Em nenhum momento, cada pessoa na empresa estaria ciente da "fofoca" sobre o seu gerente deixando o emprego. Na verdade, "em nenhum momento" está incorreto; o tempo gasto para que todos saibam seria da ordem logarítmica do número de funcionários da empresa. É exatamente como os nós de computador fofocam uns com os outros.

Considere uma rede de nós de computador. Digamos que o nó N1 receba uma nova informação. N1 selecionaria aleatoriamente um peer (digamos, nó N2) e compartilharia a informação. N1 e N2 escolheriam um peer cada (digamos, nó N3 e N4) aleatoriamente e compartilhariam a informação. O processo continua dessa maneira até que a informação seja passada para todos os nós conectados. Normalmente, os nós também armazenam o tempo de troca de informações. No exemplo acima, na primeira troca, o N2 armazenaria os detalhes de N1, as informações e a hora em que N2 ficou sabendo sobre as informações.

3.2.3.2 Visão Geral de Arquitetura do Cassandra

Em um ambiente Cassandra, seguimos com os conceitos aprendidos anteriormente sobre clusters, nós, servidores etc., contudo, o significado de alguns componentes varia, dentro do seu ecossistema; veja, para o Cassandra, as definições seguem desta forma:

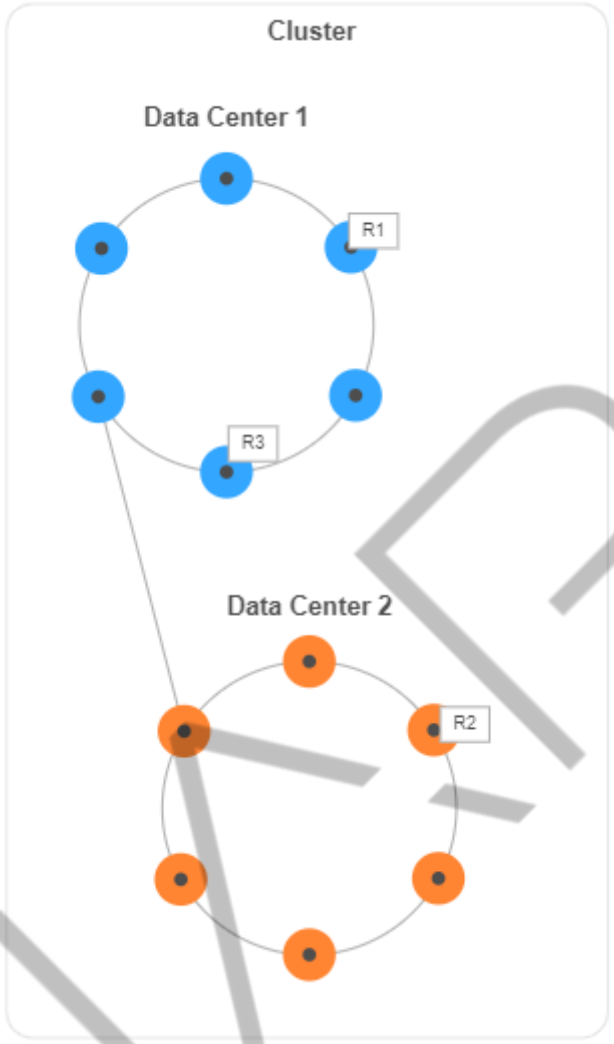

Figura 3.5 – Arquitetura do Cassandra
Fonte: Cassandra (2020)

- **Nó:** é o local onde os dados são armazenados; é o componente básico de Cassandra.
- **Data Center:** é uma coleção de nós. Muitos nós são categorizados como um data center. Repare que aqui, em outras tecnologias, chamamos o conjunto de nós interligados de cluster, porém no Cassandra, cluster tem outro significado.
- **Cluster:** é a coleção de muitos data centers.

Dito isso, sabemos que os dados são e sempre serão processados nos nós. Com relação à escrita e gravação no Cassandra, seu engine de banco de dados segue as mesmas premissas dos bancos de dados tradicionais como Oracle, SQL

Server etc., ou seja, os comandos são gravados em disco para possível recuperação, e depois entram na memória para atuar nos dados.

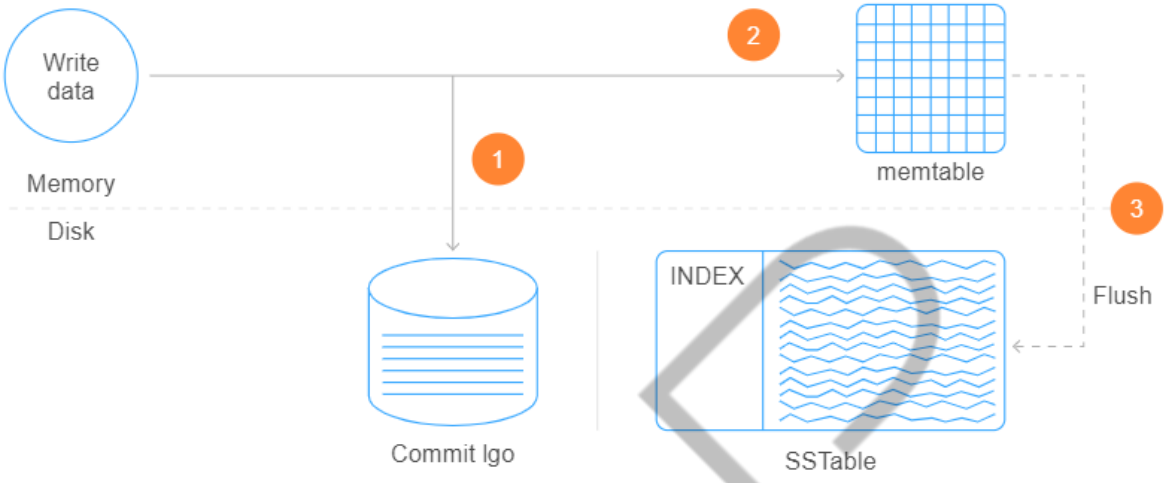

Figura 3.6 – Processo de I/O Cassandra
Fonte: Cassandra (2020)

(1) Quando o pedido de gravação chega ao nó, primeiro, ele registra no log de confirmação.

(2) Em seguida, o Cassandra grava os dados na tabela-mem. Os dados gravados na tabela mem em cada solicitação de gravação também são gravados no log de confirmação separadamente. Mem-table é um dado armazenado temporariamente na memória enquanto o log Commit registra os registros da transação para fins de backup.

(3) Quando a tabela-mem está cheia, os dados são liberados para o arquivo de dados SSTable.

Desta forma, cada operação de gravação é gravada no log de confirmação. O log de confirmação é usado para recuperação de falhas.

- **Mem-table:** após os dados gravados no log de confirmação, os dados são gravados em Mem-table (temporariamente).
- **SSTable:** quando o Mem-table atinge um certo limite, os dados são liberados para um arquivo de disco.

Os dados são replicados para garantir a ocorrência de nenhum ponto único de falha em caso de problemas de hardware ou rede/conectividade. Neste caso, o

Cassandra coloca réplicas de dados em diferentes nós com base nesses dois fatores:

- Onde colocar a próxima réplica é determinado pela **Estratégia de Replicação**.
- Enquanto o número total de réplicas colocadas em nós diferentes é determinado pelo **Fator de Replicação**.

Um fator de replicação significa que há apenas uma única cópia de dados, enquanto três fatores de replicação significam que há três cópias dos dados em três nós diferentes. Para garantir que não haja um único ponto de falha, o fator de replicação deve ser **três**. Existem dois tipos de **estratégias de replicação** no Cassandra (você verá essas duas estratégias nos exercícios posteriormente):

- SimpleStrategy
- NetworkTopologyStrategy

O **SimpleStrategy** é usado quando você tem apenas um data center. SimpleStrategy coloca a primeira réplica no nó selecionado pelo particionador. Depois disso, as réplicas restantes são colocadas no sentido horário no anel do nó.

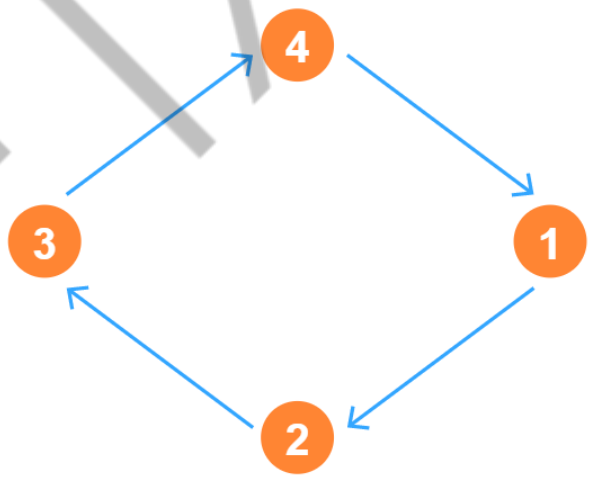

Figura 3.7 – Simple Strategy
Fonte: Elaborado pelo autor (2020)

Já o **NetworkTopologyStrategy** é usado quando você tem mais de dois data centers. As réplicas são definidas para cada data center separadamente. NetworkTopologyStrategy coloca réplicas no sentido horário no anel até atingir o primeiro nó em outro rack.

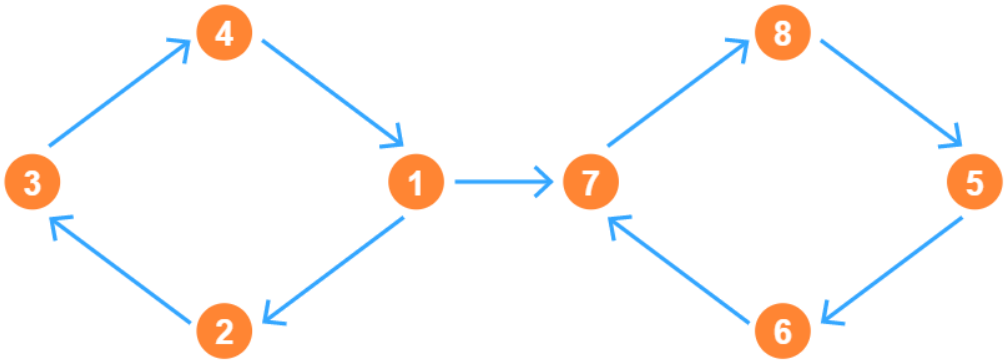

Figura 3.8 – Network Topology Strategy
Fonte: Elaborado pelo autor (2020)

Essa estratégia tenta colocar réplicas em diferentes racks no mesmo data center. Isso se deve porque, às vezes, falhas ou problemas podem ocorrer no rack. Em seguida, as réplicas em outros nós podem fornecer dados.

3.2.3.3 Modelo de dados do Cassandra

Dentro do Cassandra trabalharemos com o conceito de tabelas, porém em sua concepção, elas eram chamadas de Column Families. Outro ponto importante é que um conjunto de tabelas (column families) é chamado de Keyspace. Vamos detalhar um pouco:

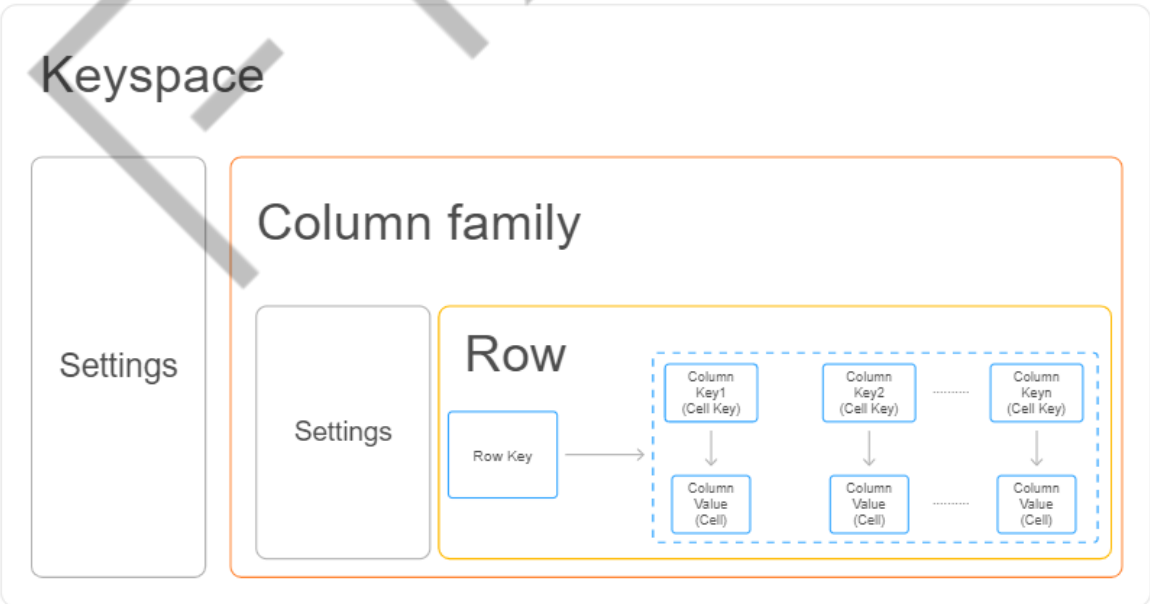

Figura 3.9 – Modelo de Dados Cassandra
Fonte: Elaborado pelo autor (2020)

O **Keyspace** é o contêiner mais externo para dados no Cassandra, dentre seus parâmetros (atributos) principais estão:

- **Fator de replicação**: especifica o número de máquinas no cluster que receberá cópias dos mesmos dados.
- **Estratégia de réplica**: é uma estratégia que especifica como colocar réplicas no data center. Existem três tipos de estratégias, tais como:
  - Estratégia simples.
  - Estratégia de topologia de rede antiga.
  - Estratégia de topologia de rede (estratégia compartilhada pelo datacenter).

As famílias de colunas (**column families** – tabelas) são colocadas sob o Keyspace. Um keyspace é um contêiner para uma lista de uma ou mais famílias de colunas, enquanto uma família de colunas é um contêiner de uma coleção de registros. Cada registro contém colunas ordenadas. As famílias de colunas representam a estrutura dos seus dados. Cada keyspace possui pelo menos uma e muitas famílias de colunas.

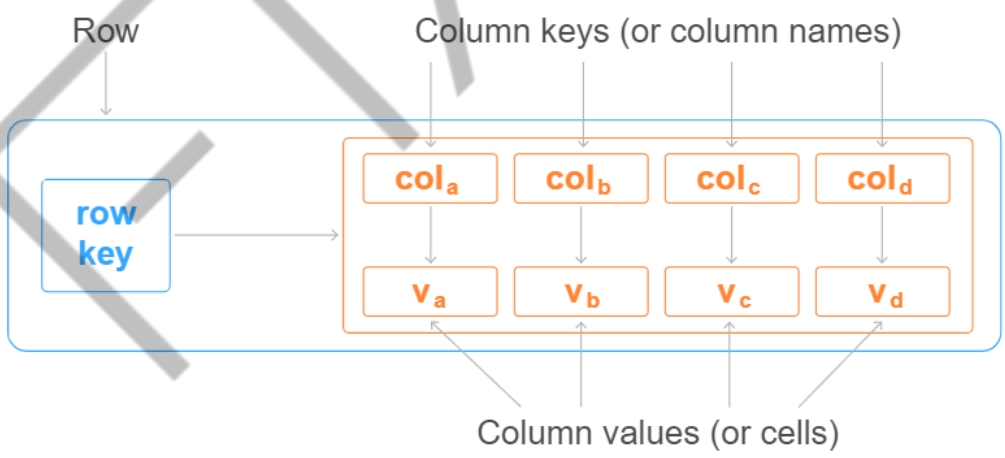

Figura 3.10 – Família de Colunas
Fonte: Cassandra (2020)

Desta forma, as camadas de um cluster Cassandra são:

- **Data center:** uma coleção de nós em um formato de anel que funcionam juntos. Ele pode abranger vários locais físicos. Os dados são distribuídos

entre os nós no cluster usando uma função consistente baseada em hashing.

- **Node:** componente básico da infraestrutura e é onde os dados são armazenados. Cada nó contém uma réplica de dados.
- **Keyspace:** uma coleção de famílias de colunas e é equivalente a um banco de dados em SGDBR. A definição de keyspace contém fator de replicação, estratégia de replicação (topologia simples ou de rede) e famílias de colunas.
- **Column Family:** um contêiner de uma coleção de linhas. Isso é equivalente a uma tabela no SGDBR.
- **Row:** identificada por uma chave exclusiva e cada linha pode ter colunas diferentes. A chave de linha é uma string única e não há limite para seu tamanho.
- **Column:** uma construção que possui um nome, um valor e um carimbo de data/hora definido pelo usuário. Cada linha pode ter um número variável de colunas.

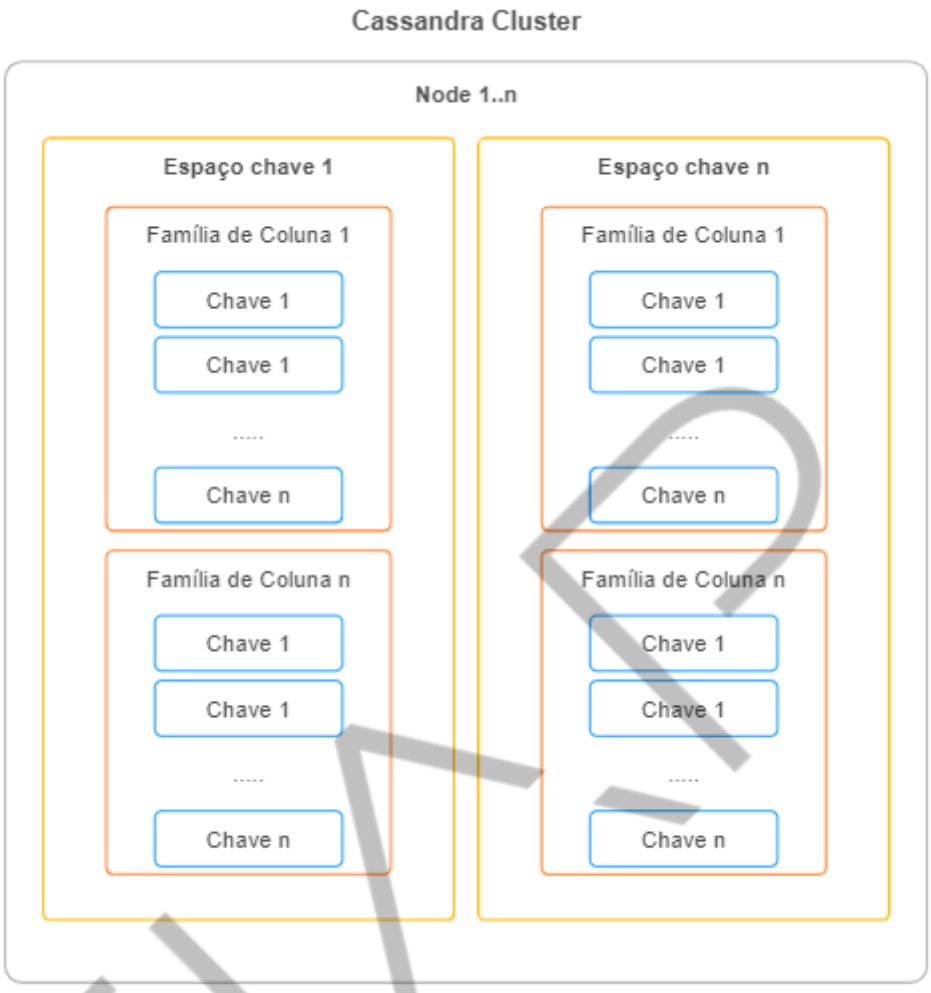

Figura 3.11 – Cluster Cassandra
Fonte: Cassandra (2020)

Em termos gerais, as boas práticas rezam:

- O Cassandra não suporta JOINs, grupos, cláusulas OR, agregações etc. Portanto, você precisa armazenar seus dados de tal forma que eles possam ser completamente recuperáveis. Por isso, essas regras devem ser consideradas durante a modelagem de dados no Cassandra.
- No Cassandra, as gravações não são onerosas, ele é otimizado para alto desempenho de gravação. Portanto, tente maximizar suas gravações para obter melhor desempenho de leitura e disponibilidade de dados. Existe uma compensação entre a gravação de dados e a leitura de dados. Assim, otimize seu desempenho de leitura de dados maximizando o número de gravações de dados.

- Desnormalização de dados e duplicação de dados são comuns no Cassandra. O espaço em disco não é mais caro que a memória, o processamento da CPU e a operação de IOs. Como o Cassandra é um banco de dados distribuído, a duplicação de dados fornece disponibilidade de dados instantânea e nenhum ponto único de falha.

Como meta, procure espalhar os dados uniformemente em torno do cluster. Para distribuir a mesma quantidade de dados em cada nó do cluster do Cassandra, você deve escolher inteiros como uma chave primária. Os dados são distribuídos nos diferentes nós com base nas chaves de partição que são a primeira parte da chave primária.

Maximize o número de partições lidas durante a consulta de dados: A partição é usada para ligar um grupo de registros com a mesma chave de partição. Quando a consulta de leitura é emitida, ela coleta dados de diferentes nós de diferentes partições.

No caso de muitas partições, todas essas partições precisam ser visitadas para coletar os dados da consulta. Isso não significa que as partições não devam ser criadas. Se seus dados são muito grandes, você não pode manter essa quantidade enorme de dados na partição única. A partição única será retardada. Então você deve ter um número equilibrado de partições.

No Cassandra usamos a linguagem CQL - Cassandra Query Language, muito rica e semelhante ao SQL tanto para definição das estruturas de dados - commandos DDL, como para manipulação dos dados - DML. Permite a criação de views materializadas, índices, triggers, usuários e suporte ao JSON (CASSANDRA, 2020).

Vamos pôr em prática alguns conceitos e comandos utilizando a tecnologia líder!!!!

3.2.3.4 Instalação no Windows

- Baixar e instalar última versão Oracle JDK 8 (Java Development Kit).

<https://www.oracle.com/br/java/technologies/javase/javase-jdk8-downloads.html>.

- Criar uma variável de sistema JAVA_HOME com o valor igual ao diretório instalado. Testar no cmd: echo %JAVA_HOME% (o sistema responde com o diretório da instalação).

  Teste para verificar o funcionamento: abrir o cmd e digitar 'java -version' (o sistema responde com a versão que está rodando).

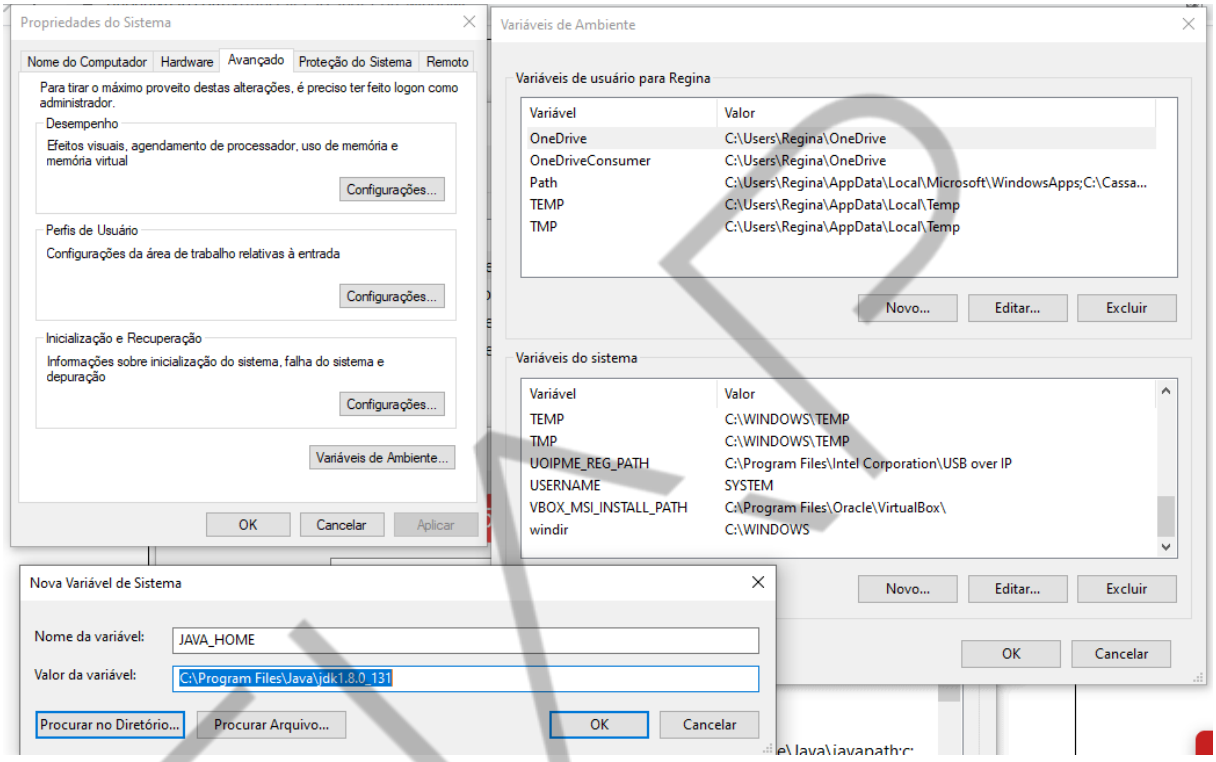

Figura 3.12 – JAVA_HOME
Fonte: Elaborado pelo autor (2020)

- Baixar e instalar Python 2.7 para Windows
  - Download da última versão

<https://www.python.org/downloads/release/python-2718/>.

  - Executar o instalador. Na opção 'Customize Python' do instalador, clicar em 'Add python.exe to Path' e selecionar a primeira opção 'Will be installed on local hard drive' para ativar a opção. Continue a instalação normalmente.
  - Teste para verificar o funcionamento: abrir o cmd e digitar 'python - -version' (o sistema responde com a versão que está rodando).

- Adicionar na variável Path - variável de sistema – diretório C:\Python27.

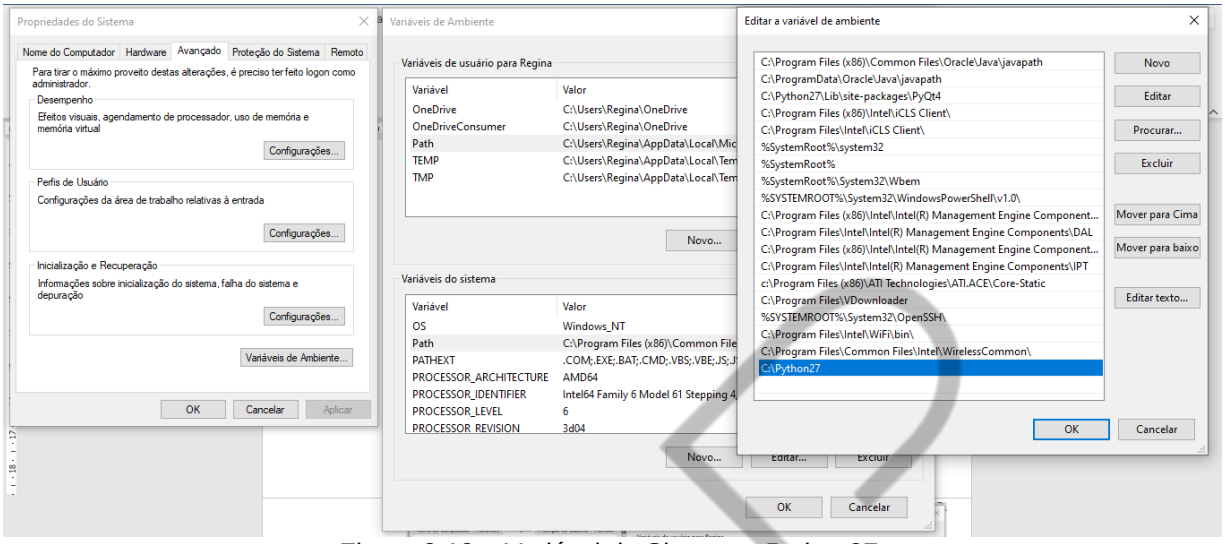

Figura 3.13 – Variável do Sistema - Python27
Fonte: Elaborado pelo autor (2020)

- Instalar Cassandra
  - Criar um diretório C:\Cassandra
  - Download última versão 3.11 release: 3.11.8 de 31 -08 -2020 no diretório C:/Cassandra

<https://cassandra.apache.org/download/>.

  - Unzip do arquivo (extrair na pasta).
  - Criar uma variável de sistema CASSANDRA_HOME com o valor = C:\Cassandra\apache-cassandra-3.11.8. Testar no cmd: echo %CASSANDRA_HOME% (o sistema responde com o diretório da instalação).

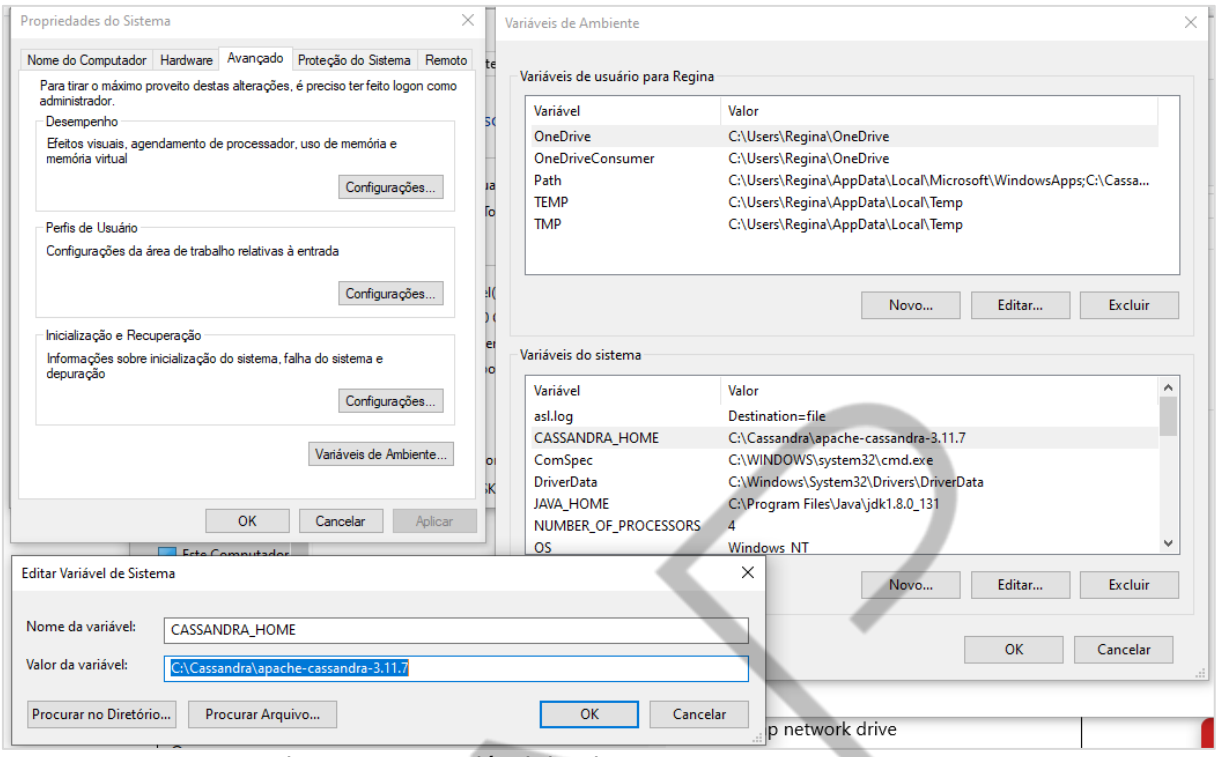

Figura 3.14 – Variável do Sistema CASSANDRA_HOME
Fonte: Elaborado pelo autor (2020)

- Inicializar o banco
  - Antes de tudo, verificar no PowerShell do Windows a política de execução, com o seguinte comando: Get-ExecutionPolicy. Ela deve estar 'Unrestricted'. Caso não esteja, digite o seguinte comando: Set-ExecutionPolicy Unrestricted. Veja mais sobre Execution Policy em:

< https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-5.1&viewFallbackFrom=powershell-Microsoft.PowerShell.Core>

  - Abrir cmd no diretório C:\Cassandra\apache-cassandra-3.11.7\bin
  - Executar cassandra.bat em cmd como administrador, com o seguinte comando: cassandra.bat -f.

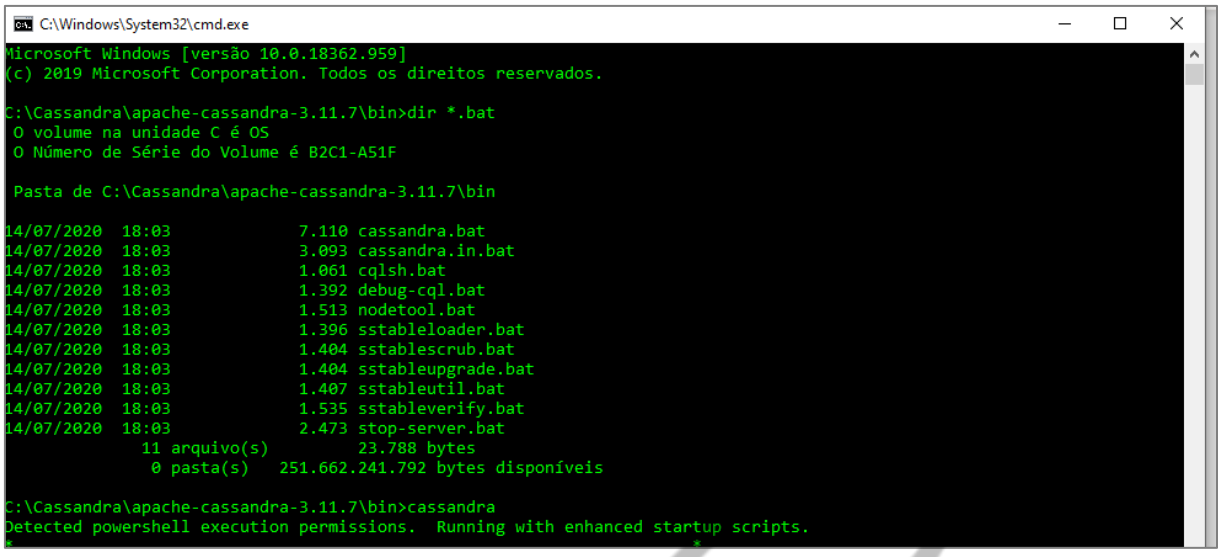

Figura 3.15 – Inicializar Cassandra
Fonte: Elaborado pelo autor (2020)

Atenção: não fechar a janela! Somente minimizar!

- Interface comando de linha
  - No diretório bin chamar 'cqlsh'. Será conectado ao Cluster.
  - Para resolver o 'Warning: pyreadline dependency missing': abrir outro cmd como administrador e digitar 'pip install pyreadline'. Após isso, chamar 'cqlsh' novamente. Deverá retornar sem nenhuma mensagem de erro. O ambiente está pronto para utilizar.

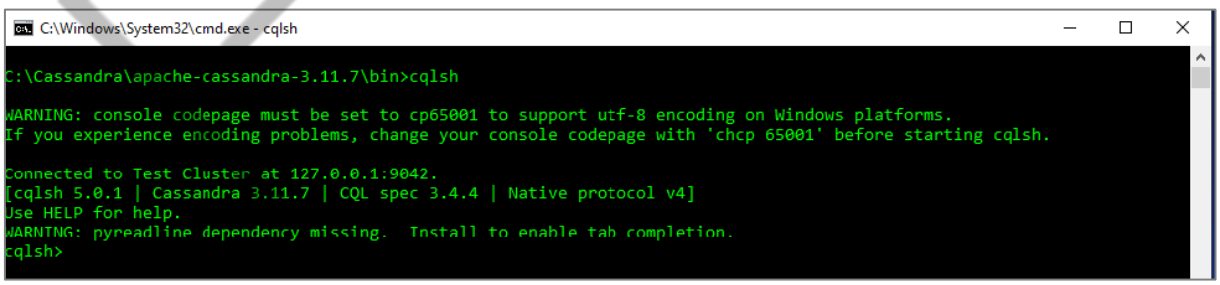

Figura 3.16 – Cassandra cqlsh
Fonte: Elaborado pelo autor (2020)

3.2.3.5 Comandos básicos

Exemplos e comandos presentes no Cassandra, The Definitive Guide 2nd Ed.

Comandos iniciais do ambiente Cassandra:

- Lista de comandos disponíveis

```
HELP
```

Código-fonte 3.1 – Sintaxe do comando HELP
Fonte: Cassandra (2020)

- Cluster atual no qual estamos trabalhando

```
DESCRIBE CLUSTER;
```

Código-fonte 3.2 – Sintaxe do comando CLUSTER
Fonte: Cassandra (2020)

- Quais keyspaces estão disponíveis no cluster

```
DESCRIBE KEYSPACES;
```

Código-fonte 3.3 – Sintaxe do comando KEYSPACES
Fonte: Cassandra (2020)

- Versão de cliente, servidor e protocolo em uso

```
SHOW VERSION;
```

Código-fonte 3.4 – Sintaxe do comando VERSION
Fonte: Cassandra (2020)

- Configurações de paginação padrão que serão usadas nas leituras

```
PAGING;
```

Código-fonte 3.5 – Sintaxe do comando PAGING
Fonte: Cassandra (2020)

- Nível de consistência padrão que será usada em todas as leituras

```
CONSISTENCY;
```

Código-fonte 3.6 – Sintaxe do comando CONSISTENCY
Fonte: Cassandra (2020)

- Opções de rastreamento padrão.

```
TRACING;
```

Código-fonte 3.7 – Sintaxe do comando TRACING
Fonte: Cassandra (2020)

3.2.3.6 Criando uma Keyspace e realizando vários comandos numa tabela

- Criando um Keyspace

```
CREATE KEYSPACE fiap_on WITH replication = {'class':
'SimpleStrategy', 'replication_factor': 1};
```

Código-fonte 3.8 – Comando CREATE KEYSPACE
Fonte: Elaborado pelo autor (2020)

A título de exemplo, veja que através dos parâmetros de estratégia de replicação e fator de replicação, conseguimos informar como será a replicação de dados entre os nós de um cluster, ou seja, de vários data centers. Veja o seguinte comando:

```
CREATE KEYSPACE Test
WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3};
```

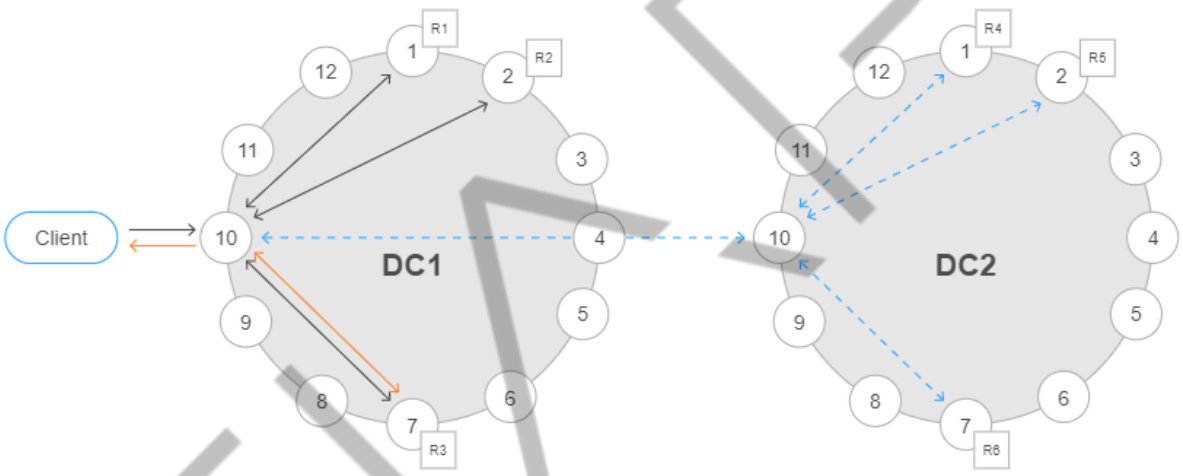

Figura 3.17 – Estratégia de replicação em 2 data centers
Fonte: Cassandra (2020)

No comando de exemplo, estamos especificando que haverá uma réplica no DC1 (data center 1) e três no DC2 (data center 2). Para alterar as configurações de um keyspace, podemos usar o comando **ALTER**:

```
ALTER KEYSPACE Test

WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
```

E, consequentemente, para apagá-la, o comando **DROP**:

```
DROP KEYSPACE Test;
```

Para mais informações sobre KEYSPACES, acesse a documentação: <https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateKeyspace.html>.

Voltemos agora ao hands-on:

- Detalhando um Keyspace

```
DESCRIBE KEYSPACE fiap_on;
```

Código-fonte 3.9 – Comando DESCRIBE
Fonte: Elaborado pelo autor (2020)

- assinalando um keyspace para trabalhar. Observar a troca do prompt após o comando.

```
USE fiap_on;
```

Código-fonte 3.10 – Comando USE
Fonte: Elaborado pelo autor (2020)

Vamos praticar um pouco mais? Execute, a partir do aprendido, a tarefa abaixo:

- Defina um **KEYSPACE** chamado twitter com estratégia de replicação simples e fator de replicação 1.

Ótimo, agora teremos 2 **KEYSPACES** criados para serem explorados e usados! Precisamos agora criar as **COLUMN FAMILIES** (Tabelas) dentro dos **KEYSPACES** para podermos armazenar os dados.

Você reparou que os **KEYSPACES** são similares aos schemas de alguns bancos relacionais? Para a criação de COLUMN FAMILIES, use a sintaxe simples:

CREATE TABLE IF NOT EXISTS keyspace_name.table_name

(column_definition, column_definition, ...)

WITH property **AND** property ...

Na definição das colunas, temos:

column_name cql_type **STATIC PRIMARY KEY**

|column_name <tuple<tuple_type> tuple<tuple_type>... > PRIMARY KEY

|column_name frozen<user-defined_type> PRIMARY KEY

|column_name frozen<collection_name><collection_type>... PRIMARY KEY

|PRIMARY KEY (partition_key)

A cláusula **IF NOT EXISTS** permite criar a tabela somente se ela não existir. E como regra, os nomes de tabela válidos são cadeias de caracteres alfanuméricos e sublinhados, que começam com uma letra. Você pode usar a notação de ponto para especificar um **KEYSPACE** para a tabela: o nome do **KEYSPACE** seguido por um ponto seguido do nome da tabela, o Cassandra criará a tabela no **KEYSPACE** especificado, mas não altera o **KEYSPACE** atual; caso contrário, se você não usar um nome de **KEYSPACE**, o Cassandra criará a tabela dentro do **KEYSPACE** atual.

Há também a possibilidade de se usar uma coluna estática para armazenar os mesmos dados em várias linhas em cluster de uma partição e, em seguida, recuperar esses dados com uma única instrução **SELECT**.

Você pode adicionar uma coluna de contador, que foi aprimorada no Cassandra 2.1, a uma tabela. Vamos analisar um exemplo da criação de uma tabela com valores ordenados:

```
CREATE TABLE timeseries (
  event_type text,
  insertion_time timestamp,
  event blob,
  PRIMARY KEY (event_type, insertion_time) )
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

Código-fonte 3.11 – Criação de uma tabela com valores ordenados
Fonte: Cassandra (2020)

Como apresentado acima, você pode ordenar os resultados da consulta para usar a classificação de colunas no disco. Você pode ordenar os resultados em ordem crescente ou decrescente. A ordem ascendente será mais eficiente do que descendente. Se precisar de resultados em ordem decrescente, você pode especificar uma ordem de cluster para armazenar colunas no disco na ordem inversa do padrão. As consultas descendentes serão mais rápidas do que as ascendentes.

Outra funcionalidade muito interessante na hora de criar tabelas é o uso de compactação, veja:

```
CREATE COLUMNFAMILY sblocks (
  block_id uuid,
  subblock_id uuid,
  data blob,
  PRIMARY KEY (block_id, subblock_id) )
WITH COMPACT STORAGE;
```

Código-fonte 3.12 – Uso de compactação
Fonte: Cassandra (2020)

Quando você cria uma tabela usando chaves primárias compostas, para cada dado armazenado, o nome da coluna precisa ser armazenado com ele. Em vez de cada coluna de chave não primária ser armazenada de forma que cada coluna corresponda a uma coluna no disco, uma linha inteira é armazenada em uma única coluna no disco. Se você precisar economizar espaço em disco, use a diretiva **WITH COMPACT STORAGE,** que armazena dados no formato de mecanismo de armazenamento legado (Thrift).

O uso da diretiva de armazenamento compacto impede que você defina mais de uma coluna que não faça parte de uma chave primária composta. Uma tabela compacta que usa uma chave primária que não é composta pode ter várias colunas que não fazem parte da chave primária.

Uma tabela compacta que usa uma chave primária composta deve definir pelo menos uma coluna de cluster. As colunas não podem ser adicionadas nem removidas após a criação de uma tabela compacta. A menos que você especifique WITH COMPACT STORAGE, o CQL cria uma tabela com armazenamento não compacto.

Ou seja, na criação das tabelas no Cassandra, podemos especificar propriedades que afetam o comportamento dos dados que irão residir na tabela, veja este outro exemplo:

```
CREATE TABLE monkeySpecies (
  block_id uuid,
  species text,
  average_size text,
  population varint,
  PRIMARY KEY (species, block_id) )
WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : '120' };
```

Código-fonte 3.13 – Comportamento dos dados
Fonte: Cassandra (2020)

Configure o cache criando um mapa de propriedade de valores para a propriedade de armazenamento em cache. Opções: **KEYS: ALL** ou **NONE**

rows_per_partition: número de linhas CQL (N), NONE ou ALL

De acordo com o valor de rows_per_partition, Cassandra armazena em cache apenas as primeiras N linhas em uma partição, conforme determinado pela ordem de armazenamento em cluster.

Como visto, usando a cláusula opcional **WITH** e os argumentos de palavra-chave, você pode configurar o armazenamento em cache, a compactação e várias outras operações que o Apache Cassandra executa na nova tabela. Por exemplo, para incorporar um comentário em uma tabela, você formata o comentário como uma propriedade de string:

```
CREATE TABLE MonkeyTypes (
block_id uuid,
species text,
alias text,
population varint,
PRIMARY KEY (block_id) )
WITH comment='Important biological records'
AND read_repair_chance = 1.0;
```

Código-fonte 3.14 – Incorporando comentário
Fonte: Cassandra (2020)

Para mais informações sobre COLUMN FAMILIES (Tabelas), acesse a documentação Cassandra:

<https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlCreateTable.html#cqlCreateTable>

Vamos experimentar alguns comandos:

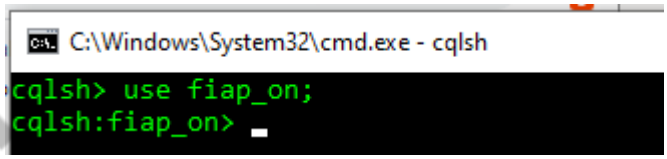

Figura 3.18 – Resultado Use
Fonte: Elaborado pelo autor (2020)

- Criando uma tabela para praticar alguns comandos

```
CREATE TABLE user ( first_name text, last_name text, PRIMARY
KEY (first_name));
```

Código-fonte 3.15 –Comando CREATE TABLE user
Fonte: Cassandra (2020)

- Detalhando uma tabela

```
DESCRIBE TABLE user;
```

Código-fonte 3.16 – Comando DESCRIBE TABLE
Fonte: Cassandra (2020)

Faça agora alguns exercícios:

Usando o Keyspace Twitter crie 2 tabelas:

- Crie uma tabela com 3 colunas, ID (integer), name (texto) e email (texto). Defina a primary key para coluna ID. Chame essa tabela de **users**.
- Crie uma tabela com 4 colunas, posted_on (big integer), user_id (integer), user_name (texto), body (texto - corpo da mensagem) e defina a primary key composta pelas colunas user_id e posted_on. Chame essa tabela de **messages**.
- Descreva ambas as tabelas criadas.

Assim como nos KEYSPACES, podemos alterar algumas propriedades e estruturas das tabelas, veja:

```
ALTER TABLE monkeySpecies ALTER average_size TYPE varint;
ALTER TABLE monkeySpecies ADD gravesite varchar;
```

No primeiro exemplo, mudamos o data type da coluna average_size da tabela monkeySpecies para varint. E no segundo exemplo, adicionamos uma nova coluna chamava gravesite do tipo varchar a mesma tabela.

Exercite:

- Adicionando uma coluna

```
ALTER TABLE user ADD title text;
```

Código-fonte 3.17 – Comando ALTER TABLE user
Fonte: Cassandra (2020)

- Detalhando a tabela

```
DESCRIBE TABLE user;
```

Código-fonte 3.18 –Comando DESCRIBE TABLE user
Fonte: Cassandra (2020)

Para remover (apagar) uma tabela, vamos com o comando **DROP TABLE**:

```
DROP TABLE monkeySpecies;
```

Podemos também usar o comando TRUNCATE:

```
TRUNCATE monkeySpecies;
```

Para mais informações sobre comandos e especificidades em criação e manutenção de tabelas no Cassandra, acesse:

<https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlDropTable.html>.

Execute agora mais alguns exercícios:

Alterando a estrutura das tabelas:

- Adicione uma nova coluna chamada password (texto) na tabela **users**.
- Adicione outra coluna chamada password_reset_token (texto) na tabela **users**.
- Descreva a tabela **users**.

Já que temos as estruturas prontas, podemos partir para a inserção de dados, e o comando INSERT segue as mesmas premissas dos bancos tradicionais. Portanto, vamos já executar alguns comandos para popular as tabelas criadas:

Insira os seguintes dados nas tabelas:

- Execute os seguintes inserts na tabela USERS:

```
INSERT INTO users (id, name, email) VALUES (101, 'otto', 'otto@abc.de');
INSERT INTO users (id, name) VALUES (102, 'jane');
INSERT INTO users (id, name) VALUES (103, 'karl');
INSERT INTO users (id, name, email) VALUES (104, 'linda', 'linda@abc.de');
INSERT INTO users (id, name, email) VALUES (105, 'gerd', 'g@rd.de');
INSERT INTO users (id, name, email) VALUES (106, 'heinz','heinz@xyz.de');
```

Código-fonte 3.19 – Inserts na tabela
Fonte: Klems (2013)

Insira os seguintes dados nas tabelas. Execute os seguintes inserts na tabela MESSAGES:

```
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (101,
1384895178, 'otto', 'Hello World!');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (101,
1384895319, 'otto', 'Hello again...');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (104,
1384895222, 'linda', 'Hi, Otto');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (103,
1384895223, 'karl', ‘Politic sucks');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (103,
1384895224, 'karl', 'rhabarber rhabarber rhabarber');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (103,
1384895225, 'karl', ‘want desperate eat a burger');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (103,
1384895226, 'karl', ‘Avengers Infinite War has lots of problems about quantic phisics
concepts');
INSERT INTO messages (user_id, posted_on, user_name, body) VALUES (103,
1384895227, 'karl', ‘A pale shelter');
```

Código-fonte 3.20 – Comando INSERT INTO messages
Fonte: Klems (2013)

Mais um pouco de hands-on: Inserindo dados na tabela.

```
INSERT INTO user (first_name, last_name) VALUES ('Bill',
'Nguyen');
```

Código-fonte 3.21 – Comando INSERT INTO user
Fonte: Cassandra (2020)

Total de linhas na tabela

```
SELECT COUNT (*) FROM user;
```

Código-fonte 3.22 – Comando SELECT COUNT(*)
Fonte: Cassandra (2020)

Reparou que os comandos são muito parecidos com os de uma base de dados relacional tradicional? Isso facilita bastante as coisas em termos de curva de aprendizagem e adoção. Há também a possibilidade de inserirmos dados no formato JSON, veja o exemplo de INSERT abaixo:

```
INSERT INTO cycling.cyclist_category JSON '{
  "category" : "GC",
  "points" : 780,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda",
  "lastname" : "SUTHERLAND" }';
```

Código-fonte 3.23 – Comando INSERT INTO JSON
Fonte: Datastax (2020)

Isso abre muitas possibilidades de integração de sistemas, correto? Afinal, o formato **JSON** é muito difundido e utilizado em vários ambientes. Agora, veja como solicitar uma **QUERY** no Cassandra, para que ele gere um **JSON**:

```
SELECT JSON name, checkin_id, timestamp
FROM checkin;
```

Viu? Só colocar a palavra-chave **JSON** após o **SELECT**, e ele formata os dados no padrão **JSON**:

```
 [json]
----------------------------------------------------------------------------------------------------
--------------
 {"name": "BRAND", "checkin_id": "50554d6e-29bb-11e5-b345-feff8194dc9f", "timestamp": "2016-08-28 2
1:45:10.406Z"}
  {"name": "VOSS", "checkin_id": "50554d6e-29bb-11e5-b345-feff819cdc9f", "timestamp": "2016-08-28 2
1:44:04.113Z"}
(2 rows)
```

Figura 3.19 – Resultado SELECT JSON
Fonte: Datastax (2020)

Para mais informações sobre essa funcionalidade, acesse:

<https://docs.datastax.com/en/cql/3.3/cql/cql_using/useQueryJSON.html>.

Mas, além do tradicional, o Cassandra tem suas peculiaridades, como uma base NOSQL, pois ele apresenta alguns tipos diferentes de dados, enquanto temos colunas cujo data type é **INT**eger, ou **VARCHAR**, ou mesmo **DATE**, podemos ter também colunas com data types compostos, ou os chamados **COLLECTIONS**.

3.2.3.7 Datatypes básicos e Collections

Abaixo temos uma lista compacta dos data types básicos:

ascii	strings
bigint	integers
blob	blobs
boolean	booleans
counter	integers
decimal	integers, floats
double	integers
float	integers, floats
inet	strings
int	integers
text	strings
timestamp	integers, strings
timeuuid	uuids
uuid	uuids
varchar	strings
varint	integers

Código-fonte 3.24 – lista de data types básicos Cassandra
Fonte: Cassandra (2020)

Segue a lista das collections, que vamos abordar a partir daqui:

map	Dicionário de dados (chave valor)
set	Coleção de único valor
list	Coleção

Código-fonte 3.25 – lista de collections Cassandra
Fonte: Cassandra (2020)

O Cassandra fornece tipos de coleção como uma forma de agrupar e armazenar dados em uma coluna. Por exemplo, em um banco de dados relacional, um agrupamento, como vários endereços de e-mail de um usuário, está relacionado a um relacionamento unindo muitos para um entre uma tabela de usuário e uma tabela de e-mail. O Cassandra evita **JOINs** entre duas tabelas, armazenando os endereços de e-mail do usuário em uma coluna de coleção na tabela do usuário. Cada coleção especifica o tipo de dados dos dados mantidos.

Uma coleção é apropriada se os dados para armazenamento da coleção forem limitados. Se os dados tiverem potencial de crescimento ilimitado, como mensagens enviadas ou eventos de sensor registrados a cada segundo, não use coleções. Em vez disso, use uma tabela com uma chave primária composta na qual os dados são armazenados nas colunas de cluster.

O Cassandra contém estes tipos de coleção:

- SETs (conjunto)
- LIST (lista)
- MAP (mapa)
- TUPPLE

Os nomes são parecidos com alguns objetos no Redis e no MongoDB. Observe as seguintes limitações das coleções:

- Nunca insira mais de 2 bilhões de itens em uma coleção, pois apenas esse número pode ser consultado.
- O número máximo de chaves para uma coleção de mapas é 65.535.
- O tamanho máximo de um item em uma lista ou coleção de mapas é 2 GB.
- O tamanho máximo de um item em uma coleção definida é de 65.535 bytes.

Mantenha as coleções pequenas para evitar atrasos durante a consulta. As coleções não podem ser "fatiadas"; Cassandra lê uma coleção em sua totalidade, afetando o desempenho. Portanto, as coleções devem ser muito menores do que os limites máximos listados. A coleção não é paginada internamente.

As listas podem incorrer em uma operação de leitura antes de gravar para algumas inserções. Os conjuntos são preferidos às listas sempre que possível. Vamos começar então com os **SETs**:

Um **SET** (conjunto) consiste em um grupo de elementos com valores únicos. Valores duplicados não serão armazenados de forma distinta. Os valores de um conjunto são armazenados de forma não ordenada, mas retornarão os elementos na ordem de classificação quando consultados.

Use o tipo de dados set para armazenar dados que tenham um relacionamento muitos para um com outra coluna. Observer que, no exemplo abaixo, um conjunto denominado tags armazena todas as tags/texto dos verbetes associados a um tipo de animal cadastrado na tabela de imagens:

```
CREATE TABLE images (
name text PRIMARY KEY,
owner text,
date timestamp,
tags set<text>) ;
```

Criada a tabela com a coluna tags do tipo SET de textos, partimos para a inserção do registro. Repare na variação do comando INSERT em que, para a coluna tags, colocamos a lista de valores (texto) entre chaves {}:

```
INSERT INTO images (name, owner, tags)
    VALUES ('cat.jpg', 'jsmith', { 'pet', 'cute' });
```

Veja como ficaria o dado na tabela (ilustração informativa):

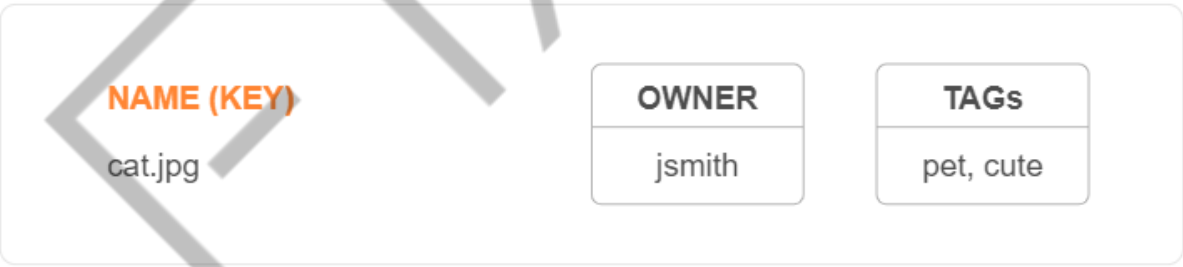

Figura 3.20 – Dados na tabela
Fonte: Elaborado pelo autor (2020)

Agora vamos elucidar o tipo **COLLECTION LIST** (lista):

Uma lista (**LIST**) tem uma forma semelhante a um **SET**, em que uma lista agrupa e armazena valores. Ao contrário de um **SET**, os valores armazenados em uma lista não precisam ser exclusivos e podem ser duplicados. Além disso, uma lista armazena os elementos em uma ordem específica e pode ser inserida ou recuperada de acordo com um valor de índice.

Use o tipo de dados de lista para armazenar dados que têm um possível relacionamento muitos para muitos com outra coluna. No exemplo abaixo, uma lista chamada scores armazena a avaliação de todos os participantes de um jogo. Cada jogador (id, game, player) pode ter vários scores.

A lista pode ser ordenada de forma que os scores apareçam na ordem em que foram inseridos, em vez de na ordem alfabética.

```
CREATE TABLE plays (
id text PRIMARY KEY,
game text,
players int,
scores list<int>) ;
```

Inserindo um registro, repare que no caso da lista (LIST), o insert sofre uma variação, ou seja, os valores da coluna LIST devem ser passados entre colchetes []:

INSERT INTO plays (id, game, players, scores)

VALUES ('123-afde', 'quake', 3, [17, 4, 2]);

Veja como ficaria o dado na tabela (ilustração informativa):

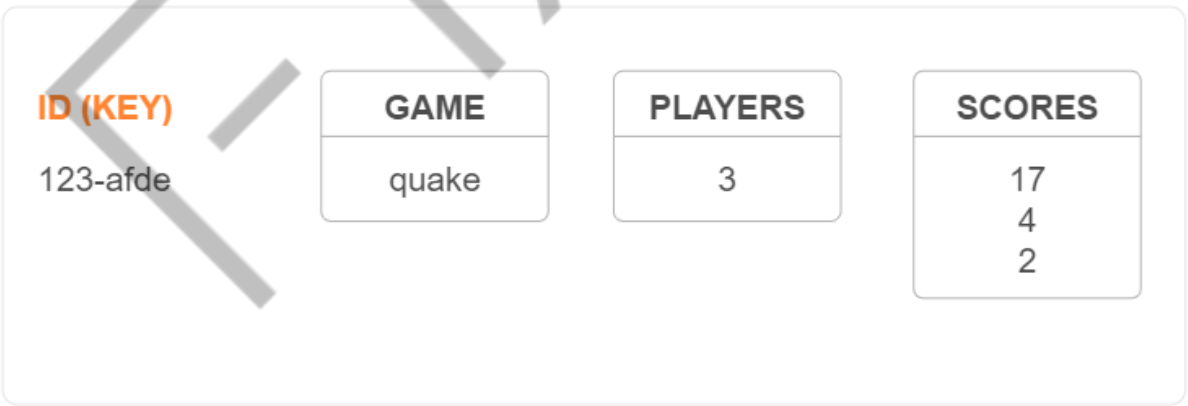

Figura 3.21 – Ilustração LIST
Fonte: Elaborado pelo autor (2020)

Falemos sobre os **MAPs** (mapas)?

Um **MAP** (mapa) relaciona um item a outro com um par de chave-valor. Para cada chave, apenas um valor pode existir e duplicatas não podem ser armazenadas. Tanto a chave quanto o valor são designados com um tipo de dados.

Usando o tipo de mapa, você pode armazenar informações relacionadas ao timestamp em perfis de usuário. Cada elemento do mapa é armazenado internamente como uma coluna do Cassandra que você pode modificar, substituir, excluir e consultar.

Veja este exemplo de uma tabela de usuários que contém uma lista de hobbies (favoritos – favs) do tipo **MAP**:

```
CREATE TABLE users (
id text PRIMARY KEY,
given text,
surname text,
favs map<text, text>);
```

Vamos inserir um registro para demonstrar seu potencial, veja também que, assim como as colunas do tipo **SET**, as colunas **MAP** também usam no comando **INSERT** as chaves, contudo, passamos agora, pares de valores (key-value):

INSERT INTO users (id, given, surname, favs)

VALUES ('jsmith', 'John', 'Smith', {'fruit' : 'apple', 'band' : 'Beatles' });

Veja como ficaria o dado na tabela (ilustração informativa):

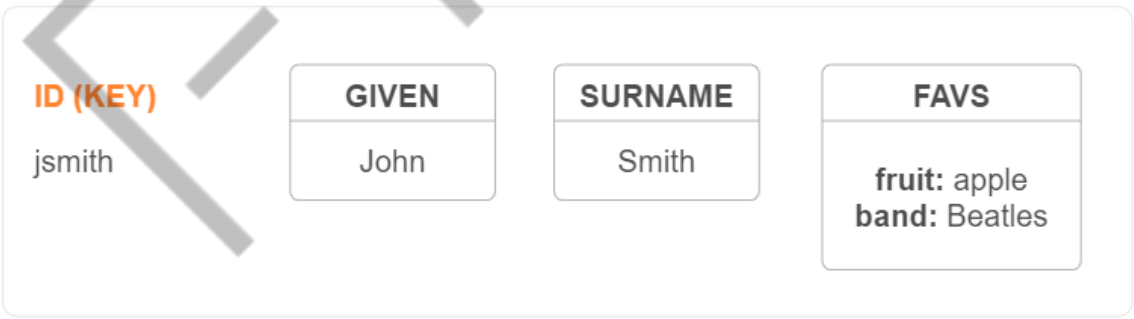

Figura 3.22 – Ilustração MAP
Fonte: Elaborado pelo autor (2020)

Há, porém, um outro tipo de **COLLECTION** interessante de se estudar, e que abre algumas outras possibilidades: o **TUPPLE**:

TUPPLEs são um tipo de dados que permitem que dois ou mais valores sejam armazenados juntos em uma coluna. Um tipo definido pelo usuário pode ser usado, mas para agrupamentos simples, um TUPPE é uma boa escolha, veja só:

```
CREATE TABLE collect_things (
  keyId int PRIMARY KEY,
  valueNames tuple<int, text, text>);
```

Inserindo um registro:

```
INSERT INTO collect_things (keyId, valueNames)
VALUES (‘Jennifer’, (0 , ‘CD’, ‘Madonna’) );
```

Veja como ficaria o dado:

Figura 3.23 – Ilustração TUPPLE
Fonte: Elaborado pelo autor (2020)

E seu uso não tem limites, veja este outro exemplo:

```
CREATE TABLE collect_things (
  keyId int PRIMARY KEY,
  valueNames tuple <int, tuple<text, double>>);
```

Código-fonte 3.26 – Comando CREATE TABLE collect_things
Fonte: DATASTAX (2020)

Começamos a trabalhar com o conceito de matrizes de dados em colunas, não é?!

Para mais informações sobre usos e práticas de COLLECTIONS, acesse a documentação Cassandra em:

<https://docs.datastax.com/en/cql/3.3/cql/cql_reference/tupleType.html>.

Vamos exercitar um pouco?

Trabalhando com **COLLECTIONs**:

- Altere a tabela **users** adicionando uma collection chamada hobbies do tipo SET de textos.

- Crie uma nova tabela dentro do keyspace twitter chamada **followers** com 2 colunas, sendo user_id (integer e primary key) e uma collection chamada followers do tipo LIST de textos. Execute um insert:
    - INSERT INTO followers (user_id, followers) VALUES (101, ['willi','heinz']);
- Adicione uma collection chamada comments do tipo MAP (texto, texto) na tabela **messages**.

Agora vamos entender outra característica específica do Cassandra relacionada à indexação, para, depois, entrarmos em índices. Para isso, execute os seguintes exercícios:

Consulte os seguintes dados:

- Escreva uma query, que retorne o nome, e email da tabela **users**.
- Altere a querie acima para somente trazer os dados para o usuário 101.
- Selecione todos os campos da tabela **messages**.
- Selecione todos os dados da tabela messages filtrando pelo user-name ‘karl’.

O que aconteceu?

Provavelmente você deve ter se deparado com um erro mais ou menos assim, não é?

```
cqlsh> use dbrnd;
cqlsh:dbrnd> CREATE TABLE IF NOT EXISTS tbl_Employee
         ... (
         ...   EmpID INT PRIMARY KEY
         ...   ,EmpFirstName VARCHAR
         ...   ,EmpLastName VARCHAR
         ...   ,EmpSalary INT
         ... );
cqlsh:dbrnd> INSERT INTO tbl_Employee
         ... (EmpID,EmpFirstName,EmpLastName,EmpSalary)
         ... VALUES
         ... (1,'Anvesh','Patel',50000);
cqlsh:dbrnd> SELECT *FROM tbl_Employee WHERE EmpFirstName='Anvesh';
InvalidRequest: code=2200 [Invalid query] message="Cannot execute this query as
it might involve data filtering and thus may have unpredictable performance. If
you want to execute this query despite the performance unpredictability, use ALL
OW FILTERING"
```

Figura 3.24 – CQL CREATE TABLE
Fonte: Patel (2016)

Pois é, quando buscamos através de um SELECT informações, e na cláusula WHERE colocamos uma coluna que não tenha índice criado, o Cassandra (assumindo que é detentor de bilhões e bilhões de registros) avisa que a query poderá demorar muito, e que você deverá especificar uma cláusula (**ALLOW FILTERING**) se responsabilizando pela possível demora no retorno dos dados. Espertinho, não?

Por isso, sempre que formos consultar uma tabela, cujos filtros se darão em colunas não indexadas, temos obrigatoriamente que adicionar esta cláusula:

```
SELECT *
FROM tbl_Employee
WHERE EmpFirstName='Anvesh'
ALLOW FILTERING;
```

Para mais informações sobre essa característica e funcionamento, acesse: <https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlSelect.html>.

Essa regra também se aplica para consultas em COLLECTIONS, veja:

```
SELECT *
FROM tbl_Employee
WHERE Address CONTAINS 'Gujarat' ALLOW FILTERING;
```

```
SELECT *
FROM tbl_Employee
WHERE Hobbies CONTAINS 'Blogging' ALLOW FILTERING;
```

```
SELECT *
FROM tbl_Employee
WHERE Emails CONTAINS 'test1@gmail.com' ALLOW FILTERING;
```

Vamos aprender a criar índices no Cassandra, para otimização, performance e evitar esse inconveniente acima?

3.2.3.8 Índices

Como já sabemos, um índice fornece um meio de acessar dados nos bancos de dados usando atributos diferentes da chave de partição. O benefício é a pesquisa rápida e eficiente de dados que correspondem a uma determinada condição.

O índice indexa os valores da coluna em uma tabela separada e oculta daquela que contém os valores que estão sendo indexados. O Cassandra possui várias técnicas para se proteger contra o cenário indesejável em que os dados podem ser recuperados incorretamente durante uma consulta envolvendo índices com base em valores obsoletos no índice.

Os índices podem ser usados para coleções, colunas de coleção e quaisquer outras colunas, exceto colunas de contador e colunas estáticas. Sua criação segue os preceitos e regras do comando em bases tradicionais, veja:

```
CREATE INDEX users_country ON users(country);
CREATE INDEX users_state ON users(state);
```

Para se remover um índice, usaremos o comando **DROP INDEX**:

```
DROP INDEX users_country;
```

Vamos exercitar?

Criação de índices:

- Crie um índice chamado name_index na tabela **users** coluna name.
- Crie um índice chamado user_name na tabela **messages** na coluna user_name.
- Selecione todos os campos da tabela **messages** filtrando por user_name igual a ‘otto’ e ‘karl’.
- Selecione as colunas nome e email da tabela users filtrando pelo email ‘Michael@abc.de’.

O que aconteceu? Corrija o erro.

Agora, e nas colunas especiais, as **COLLECTIONS,** como seria a criação de índices nelas?

Bom, essas estruturas podem ser indexadas e consultadas para encontrar uma coleção que contenha um valor específico. Conjuntos (**SETs**) e listas (**LISTs**) são indexados de maneira um pouco diferente dos mapas (**MAPs**), dada a natureza de chave-valor dos mapas. Conjuntos (**SETs**) e listas (**LISTs**) podem indexar todos os valores encontrados indexando a coluna da coleção.

```
CREATE TABLE cycling.cyclist_career_teams
( id UUID PRIMARY KEY, lastname text, teams set<text> );

CREATE INDEX team_idx ON cycling.cyclist_career_teams ( teams );

SELECT * FROM cycling.cyclist_career_teams
WHERE teams CONTAINS 'Nederland bloeit';
```

Código-fonte 3.27 – Criando índices SET e LIST para cyclist_career_teams
Fonte: Datastax (2020)

```
 id                                   | lastname | teams
--------------------------------------+----------+-----------------------------------------------------------------------------------------------------------------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |      VOS | {'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
```

Figura 3.25 – Resultado da criação de índices em cyclist_career_teams
Fonte: Datastax (2020)

Os mapas podem indexar uma chave de mapa, valor de mapa ou entrada de mapa usando os métodos mostrados abaixo. Vários índices podem ser criados na mesma coluna do mapa em uma tabela, para que as chaves, valores ou entradas do mapa possam ser consultados.

Criando Índices para MAPs (***Map key, Map value, Map Entries***):

```
CREATE TABLE cycling.cyclist_teams
( id UUID PRIMARY KEY,
cyclist_name text,
blist map<text,text> );

CREATE INDEX team_blist_idx
ON cycling.cyclist_teams(KEYS(blist));

SELECT * From cycling.cyclist_teams
WHERE teams CONTAINS KEY ‘age’;
```

Código-fonte 3.28 – Criando índice Map Key para cyclist_teams
Fonte: Datastax (2020)

```
cyclist_name     | blist
------------------+-----------------------------------------------------------------
  Claudio HEINEN | {'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}
Laurence BOURQUE |  {'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}
```

Figura 3.26 – Resultado índice Map Key para cyclist_teams
Fonte: Datastax (2020)

Criando Índices para MAPs (Map key, Map value, Map Entries):

```
CREATE TABLE cycling.birthday_list
(cyclist_name text PRIMARY KEY,
blist map<text,text>);

CREATE INDEX blist_idx ON cycling.birthday_list (VALUES(blist));

SELECT * FROM cycling.birthday_list
WHERE blist CONTAINS 'NETHERLANDS';
```

Código-fonte 3.29 – Criando índice Map Value para birthday_list
Fonte Datastax (2020)

```
cyclist_name  | blist
---------------+-----------------------------------------------------------------
Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
  Toine POELS | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
```

Figura 3.27 – Resultado índice Map Value para birthday_list
Fonte: Datastax (2020)

Criando Índices para MAPs (Map key, Map value, Map Entries):

```
CREATE TABLE cycling.birthday_list
(cyclist_name text PRIMARY KEY,
blist map<text,text>);

CREATE INDEX blist_idx ON cycling.birthday_list (ENTRIES(blist));

SELECT * FROM cycling.birthday_list WHERE blist['age'] = '23';
```

Código-fonte 3.30 – Criando índices Map Entries para birthday_list
Fonte: Datastax (2020)

```
cyclist_name     | blist
------------------+-----------------------------------------------------------------
  Claudio HEINEN | {'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}
Laurence BOURQUE |  {'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}
```

Figura 3.28 – Resultado índice Map Entries para birthday_list
Datastax (2020)

```
SELECT * FROM cyclist.birthday_list
WHERE blist['nation'] = 'NETHERLANDS';
```

Código-fonte 3.31 – Comando Select em birthday_list
Fonte: Datastax (2020)

```
 cyclist_name  | blist
---------------+-------------------------------------------------------------------
Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
  Toine POELS | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
```

Figura 3.29 – Resultado SELECT em birthday_list
Fonte: Datastax (2020)

E podemos também criar tabelas com multi **COLLECTIONS**, veja o exemplo:

```
CREATE TABLE IF NOT EXISTS tbl_Employee
(
   EmpID INT PRIMARY KEY
   ,EmpName VARCHAR
   ,Emails SET<text>
   ,Hobbies LIST<text>
   ,Address MAP<text,text>
) ;
```

Código-fonte 3.32 – Comando CREATE TABLE IF NOT EXISTS
Fonte: Patel (2016)

Veja como seria um exemplo de INSERT na tabela acima:

```
INSERT INTO tbl_Employee
(EmpID,EmpName,Emails,Hobbies,Address)
VALUES
(
   1
   ,'Anvesh'
   ,{'test1@gmail.com','test2@gmail.com'}
   ,['Blogging','Animation','Photography']
   ,{'home': 'Gujarat', 'office': 'Hyderabad'}
);
```

Código-fonte 3.33 Comando INSERT INTO tbl_Employee
Fonte: Patel (2016)

Fácil, não? E muito útil!

Você já ouviu falar sobre comandos DML (Data Manipulation Language) ou DDL (Data Definition Language) certo? Muito mencionados em instruções para bancos de dados relacionais. Porém no mundo NOSQL usamos outro termo: **CRUD** (Create, Read, Update e Delete).

Desta forma, dentro do acrônimo CRUD, já mostramos acima o “C” (Create), e um pouco do “R” (Read). Ainda sobre Read, você também viu que as queries são muitíssimo parecidas em termos de sintaxe das bases relacionais, certo? Vamos rever:

```
SELECT name, occupation
FROM users
WHERE userid IN (199, 200, 207);
SELECT name AS user_name, occupation AS user_occupation
FROM users;
```

Código-fonte 3.34 – Comando SELECT em users
Fonte: Patel (2016)

```
SELECT time, value
FROM events
WHERE event_type = 'myEvent'
AND time > '2011-02-03'
AND time <= '2012-01-01'
```

Código-fonte 3.35 – Comando SELECT em events
Fonte: Datastax (2020)

```
SELECT COUNT(*) FROM users;
SELECT COUNT(*) AS user_count FROM users;
```

Código-fonte 3.36 – Comando SELECT count(*)
Fonte: Datastax (2020)

Ainda sobre o C (Create), vale informar que os comandos INSERT e COPY estão nessa categoria, ou seja, CREATE de estruturas e de dados. E por falar no comando COPY, vamos mostrar um exemplo de carregamento de dados a partir de um arquivo externo, veja:

```
COPY cycling.cyclist_catgory
FROM 'cyclist_category.csv'
WITH DELIMITER='|' AND HEADER=TRUE
```

Código-fonte 3.37 – Comando COPY
Fonte: Datastax (2020)

Neste caso, estamos executando esse comando no prompt do sistema operacional, ou seja, fora do Cassandra. A intenção é ler os dados do arquivo CSV, informando alguns parâmetros, como delimitadores de campos e carga (ou não) do header (cabeçalho). Veja um exemplo de arquivo CSV:

```
category|point|id|lastname
GC|1269|2003|TIRALONGO
One-day-races|367|2003|TIRALONGO
GC|1324|2004|KRUIJSWIJK
```

Figura 3.30 – Exemplo de Arquivo CSV
Fonte: Datastax (2020)

Para mais informações sobre o comando COPY, acesse:

<https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlshCopy.html?hl=copy>.

Já que introduzimos o CRUD a você, vamos focar agora no U (Update)?

Você verá que não fugirá muito do habitual em termos de sintaxe de comandos. Vamos analisar o seguinte INSERT:

```
INSERT INTO NerdMovies (movie, director, main_actor, year)
VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005);
```

Agora, vamos ver um UPDATE:

```
UPDATE NerdMovies
SET director = 'Joss Whedon',
main_actor = 'Nathan Fillion',
year = 2005
WHERE movie = 'Serenity';
```

Código-fonte 3.38 – Comando UPDATE
Fonte: Cassandra (2020)

Exercite:

Inserindo linhas e preenchendo algumas colunas diferentes e consultando o resultado

```
INSERT INTO user (first_name, last_name, title) VALUES
('Bill', 'Nguyen', 'Mr.');
INSERT INTO user (first_name, last_name) VALUES ('Mary',
'Rodriguez');
SELECT * FROM user;
```

Código-fonte 3.39 – Sintaxe do comando INSERT e SELECT
Fonte: Cassandra (2020)

Visualizar o timestamp gerados para gravações anteriores:

```
SELECT first_name, last_name, writetime(last_name) FROM user;
```

Código-fonte 3.40 – Sintaxe do comando WRITETIME
Fonte: Cassandra (2020)

Observar que não temos permissão para solicitar o carimbo de data/hora nas colunas de chave primária:

```
SELECT WRITETIME(first_name) FROM user;
```

Código-fonte 3.41 – Sintaxe do comando WRITETIME
Fonte: Cassandra (2020)

Definir o timestamp em uma gravação:

```
UPDATE user USING TIMESTAMP 1434373756626000 SET last_name =
'Boateng' WHERE first_name = 'Mary' ;
```

Código-fonte 3.42 – Sintaxe do comando TIMESTAMP
Fonte: Cassandra (2020)

Verificar o TTL para uma coluna:

```
SELECT first_name, last_name, TTL(last_name) FROM user WHERE
first_name = 'Mary';
```

Código-fonte 3.43 – Sintaxe do comando TTL
Fonte: Cassandra (2020)

Definir TTL para coluna last_name para uma hora:

```
UPDATE user USING TTL 3600 SET last_name = 'McDonald' WHERE
first_name = 'Mary' ;
```

Código-fonte 3.44 – Sintaxe do comando TTL
Fonte: Cassandra (2020)

Visualizar a contagem regressiva. Execute várias vezes:

```
SELECT first_name, last_name, TTL(last_name) FROM user WHERE
first_name = 'Mary';
```

Código-fonte 3.45 – Sintaxe do comando SELECT
Fonte: Cassandra (2020)

O **UPDATE** acima altera de uma vez 3 colunas (director, main_actor e year), ou seja, como notado, segue os mesmos preceitos e regras dos comandos **UPDATE**

de bases relacionais tradicionais, agora, como ficariam os comandos **UPDATE** para **COLLECTIONS**?

Vamos analisar o exemplo da alteração de um elemento de uma coluna **SET,** lembra da tabela de animais de estimação?

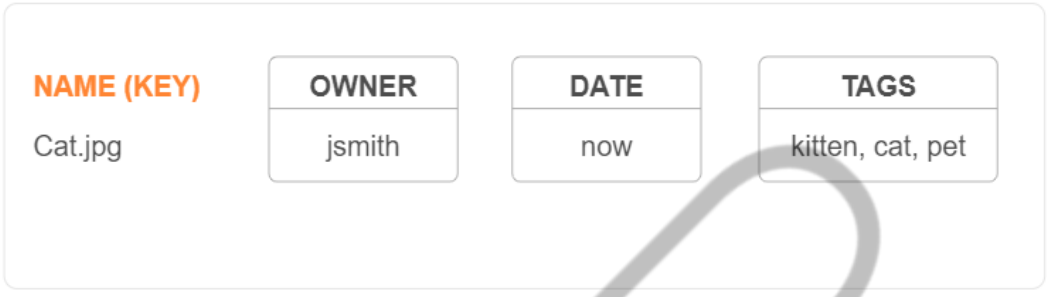

Figura 3.31 – Ilustração UPADTE SET
Fonte: Elaborado pelo autor (2020)

```
UPDATE images SET tags = tags + { 'cute', 'cuddly' }
WHERE name = 'cat.jpg';
```

No update, quando usamos a coluna recebendo ela mesma mais (+) outros valores, o update adicionará valores ao conjunto, como mostrado acima, em que os valores Kitten, Cat e Pet serão adicionados ao final do conjunto **TAGs**.

```
UPDATE images SET tags = tags - { 'lame' }
WHERE name = 'cat.jpg';
```

Já quando usamos o sinal de subtração, o valor especificado é removido do conjunto. No caso de colunas do tipo LIST, o funcionamento é um pouco diferente, vamos ilustrar aquela tabela com os scores:

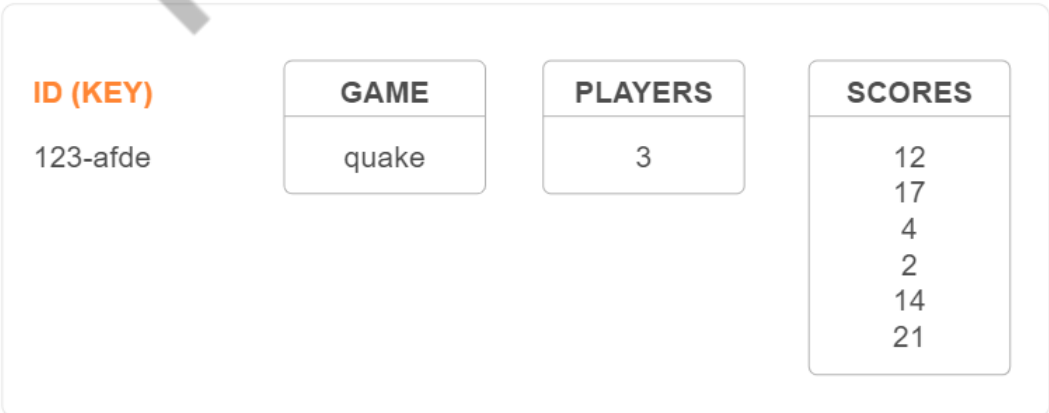

Figura 3.32 – Ilustração UPDATE LIST
Fonte: Elaborado pelo autor (2020)

```
UPDATE plays SET players = 5, scores = scores + [ 14, 21 ]
WHERE id = '123-afde';
```

Seguindo as mesmas regras do uso do sinal de soma (+) ou subtração (-), porém, agora, o posicionamento dos valores é importante; no exemplo acima, os valores 14 e 21 são colocados no final da lista. No exemplo abaixo, o valor 12 é colocado no início da lista:

```
UPDATE plays SET players = 5, scores = [ 12 ] + scores
WHERE id = '123-afde';
```

Veja outros exemplos:

```
UPDATE plays SET scores[1] = 7
WHERE id = '123-afde';
```

```
UPDATE plays SET scores = scores - [ 12, 21 ]
WHERE id = '123-afde';
```

Para mais detalhes, acesse o material Cassandra on-line:

<https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlUpdate.html>.

Agora, é para a **COLLECTION MAP**? Ela possui mais componentes e todos podem servir de insumos para o comando **UPDATE**, lembra da tabela com os hobbies (favoritos) por pessoa? Veja:

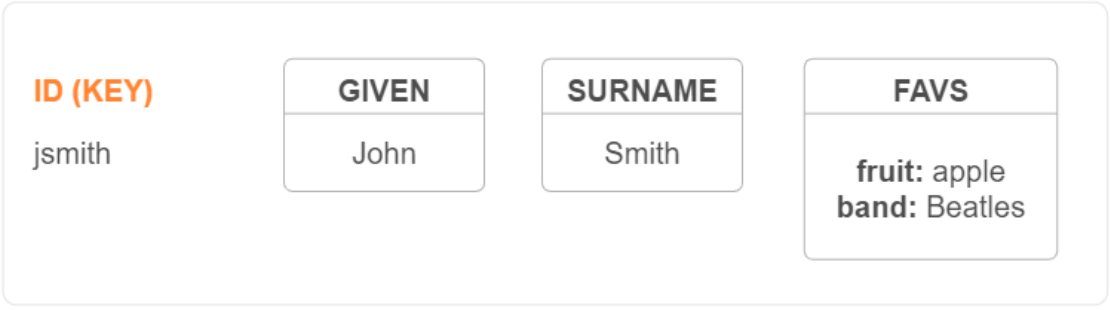

Figura 3.33 – Ilustração UPDATE MAP
Fonte: Elaborado pelo autor (2020)

Agora analise os UPDATEs:

```
UPDATE users SET favs['author'] = 'Ed Poe'
WHERE id = 'jsmith'
```

```
UPDATE users SET favs = favs + { 'movie' : 'Cassablanca' }
WHERE id = 'jsmith'
```

Entendeu? Referenciamos as porções estruturais do MAP no UPDATE, ou seu valor (chave-valor) completo, no segundo exemplo acima. Vamos praticar um pouco e fixar estes conceitos?

Updates:

- Altere a coluna email da tabela users para ‘jane@Smith.org’ quando o id for 102.
- Altere o registro 101 adicionando mais dois hobbies ‘badminton’ e ‘jazz’, na tabela hobbies.
- Altere a tabela messages adicionando o par de comentário ‘fiap’: ‘mba course!’ a coluna comments para o registro ID 103 e com valor 1384895223 na coluna posted_on.

Teste os seguintes comandos para ver sua funcionalidade:

```
INSERT INTO users (id, name, email) VALUES (110, 'franz', 'fr@nz.de') IF NOT EXISTS;
UPDATE users SET email = 'franz@gmail.com‘ WHERE id = 110 IF email = 'fr@nz.de';
UPDATE users   SET email = 'franz@ABC.de‘ WHERE id = 110 IF email = 'fr@nz.de';
UPDATE users USING TTL 77 SET password_reset_token = 'abc-xyz-123‘ WHERE id = 110;
UPDATE users SET password = 'geheim!‘ WHERE id = 110 IF password_reset_token = 'abc-xyz-123';
```

Código-fonte 3.46 – Sequencia de commandos em users
Fonte: Cassandra (2020)

Agora nos resta falar sobre o D de CRUD, ou seja, **DELETE**.

Seguiremos no mesmo caminho, e você verá que o comando não sofre muitas alterações dos usados em bases de dados tradicionais.

Vamos a um exemplo de remoção de um registro inteiro:

```
DELETE FROM NerdMovies
USING TIMESTAMP 1240003134
WHERE movie = 'Serenity';
```

Não inserindo nenhum nome de coluna após **DELETE**, remove toda a linha correspondente. Agora, removendo os valores de uma coluna somente:

```
DELETE lastname
FROM cycling.cyclist_name
WHERE id = c7fceba0-c141-4207-9494-a29f9809de6f;
```

Exclua os dados em colunas específicas listando-as após o comando **DELETE**, separadas por vírgulas. Removendo valores somente caso existam:

```
DELETE id
FROM cyclist_id
WHERE lastname = 'WELTEN' and firstname = 'Bram'
IF EXISTS;
```

Ou seja, adicione **IF EXISTS** ao comando para assegurar que a operação não seja executada se a linha especificada não existir. Também disponível para os comandos **UPDATE**. Removendo valores de **SETs**:

```
DELETE tags ['pussy']
FROM images
WHERE owner = 'jsmith';
```

TAGS
kitten, cat, pet

Removendo valores de **LISTs**:

```
DELETE scores[3]
FROM plays
WHERE game = 'quake';
```

SCORES
12
17
4

Removendo valores de **MAPs**:

DELETE favs

FROM users

WHERE fruit = ‘apple';

Já que temos todos estes conceitos em termos de CRUD, como será que funcionaria uma transação no Cassandra? Veja o exemplo abaixo da criação de um bloco de transações no Cassandra:

```
BEGIN BATCH
        INSERT INTO users (userid, password, name)
        VALUES ('user2', 'ch@ngem3b','second user');

      UPDATE users
        SET password = 'ps22dhds'
        WHERE userid = 'user3';

      INSERT INTO users (userid, password)
        VALUES ('user4', 'ch@ngem3c');

        DELETE name FROM users
        WHERE userid = 'user1';
APPLY BATCH;
```

Código-fonte 3.47 – Bloco de transação
Fonte: Cassandra (2020)

Com os comandos **BEGIN** e **APPLY BATCH**, podemos alinhar comandos para serem executados de uma só vez, garantindo assim que os dados manipulados sigam uma sequência lógica e de negócio e que não sejam interceptados por outros usuários no processo de manipulação.

Tranquilo até aqui, certo? Vamos exercitar?

Excluindo uma linha inteira:

```
DELETE FROM USER WHERE first_name='Bill';
```

Código-fonte 3.48 – Sintaxe do comando DELETE
Fonte: Cassandra (2020)

Verificando a remoção:

```
SELECT * FROM user WHERE first_name='Bill';
```

Código-fonte 3.49 – Sintaxe do comando SELECT
Fonte: Cassandra (2020)

Mais alguns exercícios?

Updates:

- Delete a coluna email da tabela users para o ID 105.
- Delete o registro 106 da tabela users.
- Crie uma transação com os seguintes comandos:

```
INSERT INTO users(id, name, email) VALUES(107, 'john','j@doe.net')

INSERT INTO users(id, name)   VALUES(108, 'michael')

UPDATE users SET email = 'michael@abc.de'  WHERE id = 108

DELETE FROM users WHERE id = 105
```

Podemos limpar um pouco o ambiente:

Zerar o conteúdo de uma tabela:

```
TRUNCATE user;
```

Código-fonte 3.50 – Sintaxe do comando TRUNCATE
Fonte: Cassandra (2020)

Remover uma tabela e seu conteúdo

```
DROP TABLE user;
```

Código-fonte 3.51 – Sintaxe do comando DROP
Fonte: Cassandra (2020)

3.3 Aprofundamento dos Comandos – Hands- ON

Criando um novo Keyspace e tabelas referentes ao modelo Hotel.

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

CREATE TYPE hotel.address (
    street text,
    city text,
    state_or_province text,
    postal_code text,
    country text
);
```

```
CREATE TABLE hotel.hotels_by_poi (
    poi_name text,
    hotel_id text,
    name text,
    phone text,
    address frozen<address>,
    PRIMARY KEY ((poi_name), hotel_id)
) WITH comment = 'Q1. Find hotels near given poi'
AND CLUSTERING ORDER BY (hotel_id ASC) ;

CREATE TABLE hotel.hotels (
    id text PRIMARY KEY,
    name text,
    phone text,
    address frozen<address>,
    pois set<text>
) WITH comment = 'Q2. Find information about a hotel';

CREATE TABLE hotel.pois_by_hotel (
    poi_name text,
    hotel_id text,
    description text,
    PRIMARY KEY ((hotel_id), poi_name)
) WITH comment = 'Q3. Find pois near a hotel';

CREATE TABLE hotel.available_rooms_by_hotel_date (
    hotel_id text,
    date date,
    room_number smallint,
    is_available boolean,
    PRIMARY KEY ((hotel_id), date, room_number)
) WITH comment = 'Q4. Find available rooms by hotel / date';

CREATE TABLE hotel.amenities_by_room (
    hotel_id text,
    room_number smallint,
    amenity_name text,
    description text,
    PRIMARY KEY ((hotel_id, room_number), amenity_name));
```

Código-fonte 3.52 – Sintaxe do comando KEYSPACES e TABLES
Fonte: Cassandra (2020)

Carregando tabela a partir de um arquivo CSV, realizando consultas sobre disponibilidade e classificando os resultados:

```
USE hotel;

COPY available_rooms_by_hotel_date FROM
'./z_available_rooms.csv' WITH HEADER=true;
```

```
SELECT * FROM available_rooms_by_hotel_date WHERE
hotel_id='AZ123' and date>'2020-01-05' and date<'2020-01-12';

SELECT * FROM available_rooms_by_hotel_date WHERE
hotel_id='AZ123' and room_number=101;

DESCRIBE TABLE available_rooms_by_hotel_date;

SELECT * FROM available_rooms_by_hotel_date WHERE date='2020-
01-25' ALLOW FILTERING;
SELECT * FROM available_rooms_by_hotel_date WHERE
hotel_id='AZ123' AND date IN ('2020-01-05', '2020-01-12');
SELECT * FROM available_rooms_by_hotel_date
  WHERE hotel_id='AZ123' AND date>'2020-01-05' AND
date<'2020-01-12'
  ORDER BY date DESC;

TRACING ON;
```

Código-fonte 3.53 – Sintaxe do comando COPY e SELECT
Fonte: Cassandra (2020)

3.4 Metadados Cassandra

Traremos agora, de forma resumida, as principais tabelas internas do Cassandra, que contêm os metadados do banco:

Dicionário de dados de Permissões e Roles - KeySpaces:

```
SELECT *FROM system_auth.roles;

SELECT *FROM system_auth.role_permissions;
```

Dicionário de dados de Objetos - KeySpaces:

```
SELECT *FROM system_schema.views;

SELECT *FROM system_schema.types;

SELECT *FROM system_schema.triggers;

SELECT *FROM system_schema.tables;

SELECT *FROM system_schema.keyspaces;

SELECT *FROM system_schema.indexes;

SELECT *FROM system_schema.functions;
```

```
SELECT *FROM system_schema.columns;

SELECT *FROM system_schema.dropped_columns;

SELECT *FROM system_schema.aggregates;
```

Dicionário de dados de Eventos e Traces - KeySpaces:

```
SELECT *FROM system_traces.sessions;

SELECT *FROM system_traces.events;
```

Dicionário de dados de Processos - KeySpaces:

```
SELECT *FROM system.available_ranges;

SELECT *FROM system.peers;

SELECT *FROM system.peer_events;

SELECT *FROM system.batchlog;

SELECT *FROM system.batches
```

Para mais informações sobre outras tabelas de metadados Cassandra, acesse:

<https://docs.datastax.com/en/cql/3.3/cql/cql_using/useQuerySystem.html?hl=system%2Ctables>

Agora, provocando você a pensar, sabemos que muitos clientes usam o Cassandra, inclusive o Facebook. Onde você veria espaço e utilidade para esse conjunto todo de funcionalidades colunares?

REFERÊNCIAS

CASSANDRA. **Manage massive amounts of data, fast, without losing sleep**. 2020. Disponível em: <https://cassandra.apache.org/doc/latest/cql/index.html>. Acesso em: 22 ago. 2020.

CATTELL, R. Scalable sql and nosql data stores. **SIGMOD Rec**, v. 39, n. 4, 2011, p. 12-27.

CHEN, J.-K., LEE, W.-Z. **An Introduction of NoSQL Databases Based on Their Categories and Application Industries**. 2019. Disponível em: <https://www.mdpi.com/1999-4893/12/5/106/pdf>. Acesso em: 17 jul. 2020.

CINTRA, J. **Sistema IoT para Aquisição de Dados com REDIS e Linguagem GO.** 2020. Disponível em: <https://josecintra.com/blog/iot-aquisicao-dados-sensores-redis-golang/>. Acesso em: 17 fev. 2021.

DATASTAX. Documentation. **CQL for the DataStax Distribution of Apache Cassandra 3.11**. Disponível em <https://docs.datastax.com/en/ddaccql/doc/cql/cql_using/useIndexColl.html >. Acesso em: 15 set. 2020.

DB-ENGINES. **DB-Engines Website**. Disponível em: <https://db-engines.com/en/>. Acesso em: 17 jul. 2020.

FINK, B. Distributed computation on dynamo-style distributed storage: riak pipe. **ACM SIGPLAN workshop on Erlang**, [s.e.], 2012, p.43-50.

HAINES, K. **Engine Yard Blog**: **To Redis or Not To Redis?** 2009. Disponível em: <https://blog.engineyard.com/2009/key-value-stores-for-ruby-part-4-to-redis-or-not-to-redis>. Acesso em: 17 jul. 2020.

HBASE. **Welcome to Apache HBase**. 2020. Disponível em: <https://hbase.apache.org/>. Acesso em: 17 jul. 2020.

KLEMS, M. **Apache Cassandra Lesson: Data Modelling and CQL3.** 2013. Disponível em: https://pt.slideshare.net/yellow7/cassandralesson-datamodelandcql3 Acesso em: 06 out 2020.

MATLI, P. S. R. **Apache Cassandra Data Modeling and Query Best Practices**. 2019. Disponível em: <https://www.red-gate.com/simple-talk/sql/nosql-databases/apache-cassandra-data-modeling-and-query-best-practices/>. Acesso em: 22 ago. 2020.

MCCREARY, D.; KELLY, A. **Making Sense of NoSQL**: A Guide for Managers and the Rest of Us. 1. ed. Nova York: Manning Publications, 2014.

PATEL, A. **SQL: Cassandra Collection Data Types – List, Set, Map**. 2016. Disponível em: https://www.dbrnd.com/tag/cassandra/ Acesso em: 07 out 2020.

SINHA, S. **Arquitetura HBase: Modelo de dados HBase e mecanismo de leitura/gravação HBase**. 2019. Disponpivel em: <https://www.edureka.co/blog/hbase-architecture/>. Acesso em: 22 jul. 2020.