

## Banco de dados não relacional Cassandra + Java

Rodrigo Pimenta Carvalho	21/03/2022 - 1.0	Versão inicial: visão geral do banco de dados, forma de instalação e método de interação usando Java.

# *Resumo !*

Versão 1.0

Pré-requisito: experiência com banco de dados relacional, Java, Maven, Docker e SQL.

## Sumário

Público alvo:	3
Visão geral do banco de dados Cassandra	4
Instalação do banco de dados Cassandra	9
Interação com o banco de dados Cassandra, via linha de comando	12
Manipulando o banco de dados Cassandra com Java	14
Referências consultadas:	21

# **Público alvo:**

Desenvolvedor de software com experiência em Java, Docker, Maven e banco de dados relacional. E sem qualquer conhecimento em Cassandra. Esse documento tem como objetivo introduzir resumidamente informações sobre o Cassandra, que são essenciais a quem deseja rapidamente iniciar interação com esse banco de dados por meio de objetos Java. Após seguir esse documento, será necessário visitar as referências para mais informações úteis.

# Visão geral do banco de dados Cassandra

O Cassandra é um banco de dados não relacional, portanto sua modelagem não usa diagramas de entidade-relacionamento. Entretanto, a mesma deve ser pensada de tal forma a dar grande performance de execução de *queries* nos dados. Isto é, a modelagem não montará relacionamentos entre tabelas no banco, mas ela deve implicar numa distribuição de dados entre tabelas e nodos, de tal forma que a arquitetura dessa distribuição crie condições para as *queries* serem executadas com a melhor performance possível. Por exemplo, se duas informações sobre pessoas serão consultadas sempre simultaneamente (imagine a busca por nome e endereços), elas devem constar numa mesma tabela. Assim uma única *query* dá conta de executar as busca dos dados. De fato, ao trabalhar com o Cassandra, existe a necessidade de seguir uma sistemática que implica em modelar os dados após modelar as *queries*. Ou seja, pensa-se nas *queries* que atenderão a demanda de um projeto e depois pensa-se no formato das tabelas e como serão distribuídas em nodos. E não há problema algum na existência de uma mesma coluna em tabelas diferentes (em bancos relacionais isso é evitado para evitar replicas de dados e consequentemente desperdício de espaço para armazená-los). Por exemplo, a coluna endereço pode vir numa tabela sobre dados pessoais e em outra tabela sobre dados residenciais. A economia de espaço em memória secundária já não é uma preocupação para bancos não relacionais.

O Cassandra foi pensado para ser escalável e também para replicar dados em nodos diferentes, aumentando a segurança contra falhas em máquinas. Os dados então são replicados entre máquinas participantes do banco de dados. Essas máquinas são os nodos. Mas, uma rede com  $N$  máquinas participantes não implica em  $N$  replicas dos dados. Na verdade, cada dado pode ser replicado  $M$  vezes, sendo que  $M \leq N$ . Quando a arquitetura do banco de dados é definida, define-se em quantos nodos cada dado será replicado. E caso a massa de dados do banco vá crescer significativamente, mais nodos podem entrar nessa participação para que haja mais máquinas a receber dados. Assim, fica desnecessário lidar com muitos dados e poucas máquinas. Cada máquina é considerada um nodo e a topologia de rede de máquinas participantes do banco é sempre um anel de nodos, em termos lógicos para o Cassandra. Os nodos do Cassandra comunicam-se entre si usando um protocolo chamado Gossip. Esse protocolo leva em consideração a topologia em anel dos nodos, para ajudar na distribuição dos dados replicantes. Como não é conveniente replicar todos os dados em todos os nodos no anel (caso contrário não adiantaria escalar para mais máquinas, quando a massa de dados cresce muito), os dados são divididos em partições. Cada partição é colocada num dado nodo do anel.

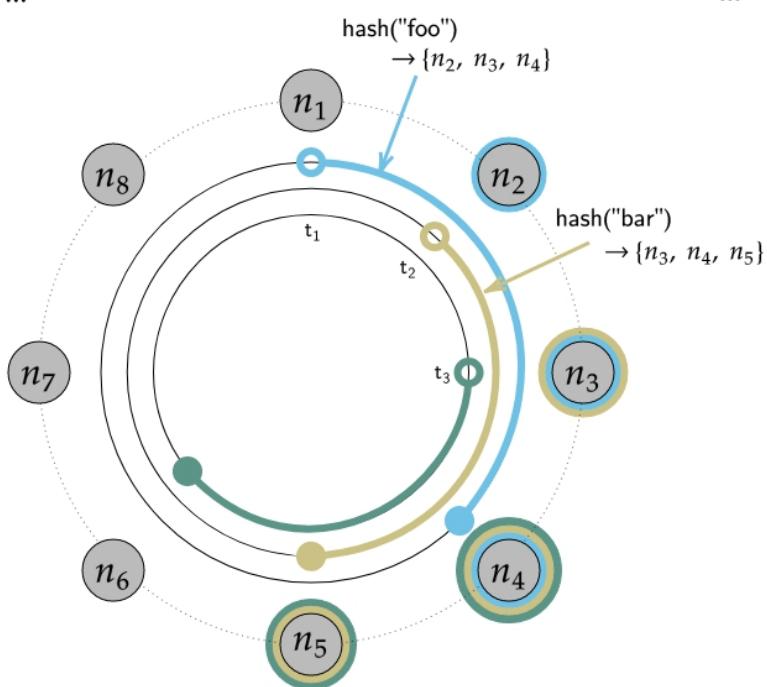
A distribuição das partições de dados tende a ser uniforme entre os nodos, porque a decisão sobre qual nodo recebe qual partição é tomada baseada no cálculo de chave *hash* (como numa hash table). A chave *hash* é calculada a partir da chave primária de uma linha de dados. Nesse caso, cada linha de dados de uma tabela deve ter uma PK. A PK é usada para gerar uma chave *hash*. Então, se a PK for bem bolada, a geração de chaves *hash* não terá colisões e implicará em uma boa distribuição de partições de dados. E como cada linha de dados tem sua PK específica, cada linha pode fazer parte de uma partição de dados diferente, independente de qual tabela pertence a linha.

Nesse caso, dados de uma mesma tabela podem ser distribuídos em partições diferentes. Portanto, enquanto nos bancos relacionais os dados são vistos organizados em tabelas (fisicamente), no Cassandra eles são organizados fisicamente em partições presentes em nodos. Mas, ainda é possível trabalhar com a noção lógica de tabelas. Por exemplo, pode-se executar *queries* de ‘*select*’ numa tabela logicamente, mas na prática a leitura de dados poderá ser feita em dois ou mais nodos, por exemplo, de forma transparente ao clinete do banco. Chaves estrangeiras em linhas de dados não são usadas, já que o banco não é relacional. Caso seja necessário relacionar dados entre tabelas, a lógica para isso deverá estar contida 100% no código do cliente do banco.

O motivo pela escolha da topologia em anel de nodos vem da forma como os dados são particionados e distribuídos entre os nodos. Após o cálculo do valor *hash* de uma chave primária, tal valor é usado para endereçar a partição na qual a linha de dados respectiva fará parte. Por exemplo, se existem 8 nodos no anel, haverá 8 partições e o *hash* calculado deverá endereçar um dos 8 nodos. Assim a linha respectiva é enviada para um desses 8. Mesmo que muitos valores *hashes* sejam gerados (muito mais que 8 diferentes), esses valores deverão ser agrupados em 8 conjuntos distintos. Os valores *hashes* calculados são chamados de Tokens. O desenho abaixo ajudará nessa explicação:

$$\begin{aligned} \text{token}(n_1) &= t_1 \\ \text{token}(n_2) &= t_2 \\ \text{token}(n_3) &= t_3 \\ &\dots \end{aligned}$$

$$\begin{aligned} \text{range}(t_1, t_2] &\rightarrow \{n_2, n_3, n_4\} \\ \text{range}(t_2, t_3] &\rightarrow \{n_3, n_4, n_5\} \\ \text{range}(t_3, t_4] &\rightarrow \{n_4, n_5, n_6\} \\ &\dots \end{aligned}$$



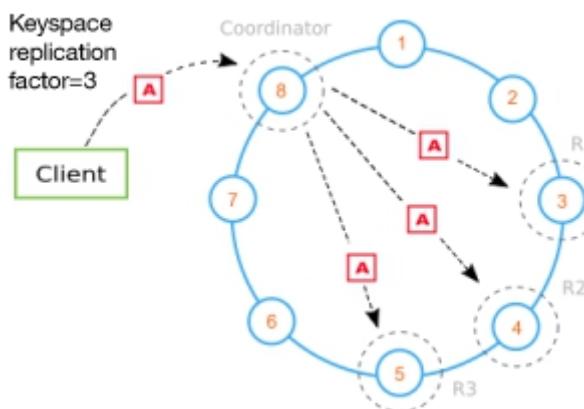
Conforme visto acima, um *token* calculado gerando o valor  $t_1$  implica em endereçar o nodo  $N_1$ . O mesmo para  $t_2$  e  $t_3$ . Um *token* calculado e gerando um valor *hash* entre  $t_1$  e  $t_2$  ( $t_1, t_2]$ ), implica num endereço de um nodo virtual que pode ser considerado existente entre  $N_1$  e  $N_2$ . Nesse caso, e seguindo o sentido horário, o primeiro nodo a realmente receber o dado (ou representar a partição para o *token* em questão) deve ser o nodo  $N_2$ . No Cassandra, o particionamento de dados pode assumir uma

multiplicidade de nodos. Isso quer dizer que para uma multiplicidade definida em 3, o dado replicado deve constar em 3 nodos em posições contiguas no anel, como visto acima. Ou seja, o *token* em questão significaria replicar os dados em N2, N3 e N4, por exemplo. No exemplo acima, o *hash (token)* calculado para a Pk “foo”, dá um valor tal que os nodos N2, N3 e N4 serão os nodos a receber dados. Então uma partição nesse caso estará presente em 3 nodos consecutivos no anel. A multiplicidade de replicação das partições pode ser definida no momento de definir a arquitetura do banco de dados. Essa definição e as *queries* que podem ser executadas são escritas numa linguagem chamada CQL. Essa linguagem é muito semelhante ao SQL. E podemos usar uma *shell* (interface de comunicação com o Cassandra) chamada *cqlsh*, como será visto mais adiante.

CQL = Cassandra Query Language.

O particionamento de dados com multiplicidade nos permite controlar quantas máquinas (nodos) devem ter dados replicados. Ao modelar o banco de dados, deve-se então prestar atenção nas *queries* que precisarão ser executadas, na escolha de dados para representar as chaves primárias e na quantidade de nodos disponíveis. Esses fatores influenciarão na modelagem final, que visará a melhor performance possível de execução de *queries*. E saber modelar muito bem um banco Cassandra, não depende somente desses fatores e qualquer metodologia documentada aplicada sobre eles, mas muito mais da experiência adquirida ao modelar outros bancos com o Cassandra, anteriormente.

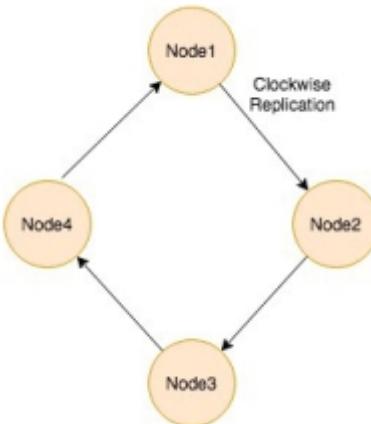
Para definir sobre a replicação de dados, um fator chamado “replication factor” é então configurado (seu valor é dado via cql), ao modelar o banco de dados. O fator de replicação pode ser do tipo *SimpleStrategy* ou *NetworkTopologyStrategy*. Com *SimpleStrategy*, define-se apenas o número de nodos a serem usados. Eles serão escolhidos no anel na ordem sentido horário, em posições contiguas a partir de um deles, conforme endereçado pelo token gerado.



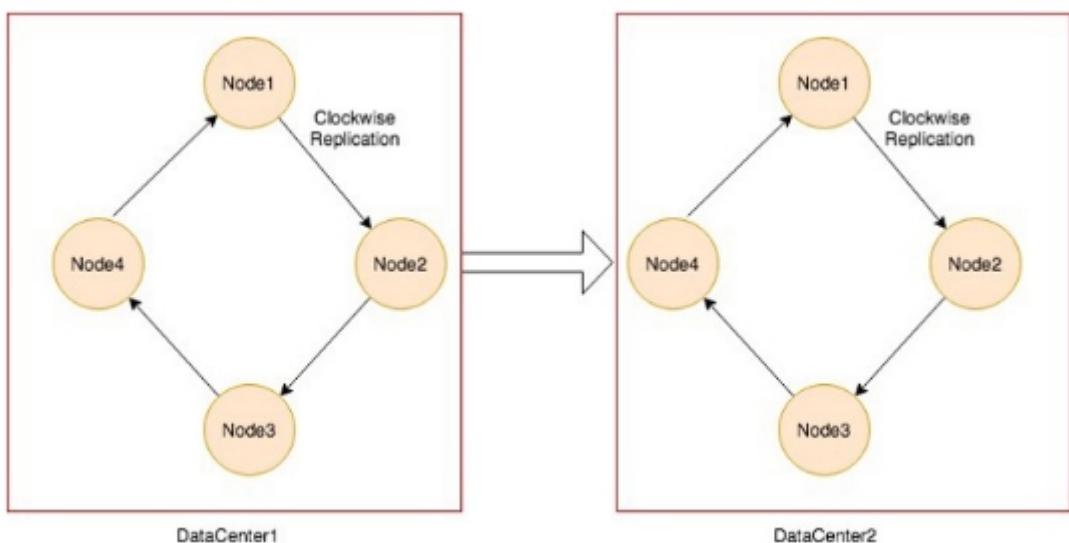
Um nodo coordenador estará responsável em definir e distribuir os dados em partições corretas.

Com *NetworkTopologyStrategy* pode-se montar a configuração de replicação levando em consideração também a qual *datacenter* pertence o anel e em quais *racks* estão presentes as máquinas. Ou seja, esse segundo tipo de fator de replicação é mais indicado para produção.

Estratégia simples:

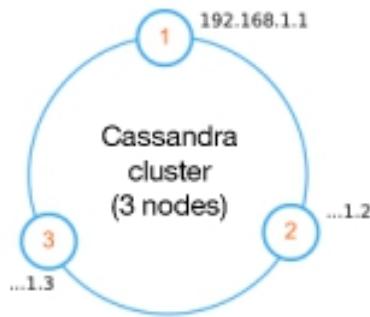


Estratégia de topologia de rede:



Caso um dos nodos esteja incomunicável no momento de uma mutação dos dados, o anel conterá inconsistências de dados entre os nodos. Então, para decidir qual nodo contem a verdade atual, o Cassandra analisa o *timestamp* de alteração da informação. O *timestamp* mais recente é o ‘vencedor’, ou seja, ele define que o dado respectivo é o verdadeiro e portanto atualizado.

Um conjunto de nodos associados entre si , que envolve a topologia anel por definição, é chamado de *cluster*.



Um *cluster* pode estar 100% contido numa única máquina, para fins de testes. Mas, o mais conveniente é manter cada nodo em uma máquina física diferente. Então um anel de nodos é na verdade formado por máquinas em rede com topologia anel em termos lógicos, controlada pelo Cassandra..

A chave primária de uma linha de dados pode ser formada por apenas um valor (uma dada coluna). Essa Pk será usada para definir a partição, como dito. Mas, uma PK pode também conter mais dados (mais colunas) em sua definição. Se uma PK contém N colunas, a primeira coluna é usada para calcular o *token* e as outras colunas são usadas para indexar os dados no *cluster*. Por exemplo, se uma Pk é formada pela tupla (p1, p2 , p3), o *token* é calculado com p1. Mas, uma query pode buscar um dado que contenha o valor X para p1, e/ou o valor Y para p2 , e/ou o valor Z para 3. Como esses 'p's estão indexados no banco, as consultas usando esses valores em seus predicados serão eficientes. Caso seja necessário usar, por exemplo, duas colunas para gerar o *token*, então duas coluna devem fazer parte da chave de partição, assim como definido abaixo com o auxílio dos parênteses:

((p1,p2), p3).

É o operador/criador do banco de dados que deve se responsabilizar em modelar os dados de tal forma que escolha a melhor decisão de configuração de chaves para partição no cluster.

# Instalação do banco de dados Cassandra

O banco de dados Cassandra pode ser instalado diretamente num sistema operacional, como o Ubuntu, mas também está disponível para uso com imagens prontas de *container* Docker. O uso de docker nesse caso traz vantagens, que facilitam a configuração de endereços IPs, por exemplo, para cada nodo.

Para instalar (copiar) o Cassandra no próprio sistema operacional, faz-se o seguinte:

- a) Download do instalador. Geralmente é um arquivo tar.gz.
- b) Descompactar com gzip. Ex: `gzip -dc apache-cassandra-4.0.3-bin.tar.gz | tar -xf -`
- c) Para executar o Cassandra: `cd apache-cassandra-4.0.3/ && bin/cassandra`
- d) Para ver o log: `tail -f logs/system.log`
- e) Para conectar ao banco com *prompt*: `bin/cqlsh`

Para instalar o Cassandra com imagens de *container* Docker, faz-se o seguinte:

Numa máquina onde haja o Docker instalado, vem

- a) Executar `docker run --name node1 -d cassandra:3.11` Esse comando obtém a imagem 3.11 do Cassandra, por exemplo, para *container* Docker. Ela será mantida na máquina. Um *container* com nome node1 passa a executar. Com o comando `docker ps -a` fica possível comprovar a existência da imagem versão 3.11 na máquina corrente. O comando `docker ps` mostra os *containers* em execução corrente.

Desse ponto em diante, mais nodos podem ser executados na mesma máquina. Por exemplo para montar um anel de nodos localmente no *host* atual. Mas, também é possível e mais conveniente executar outros nodos em outras máquinas, para que o anel de nodos seja fisicamente uma rede de máquinas com os nodos logicamente conectados em anel. Não é necessário criar uma rede com topologia em anel ou criar algum relacionamento em anel na camada de rede. O protocolo Gossip irá providenciar o anel lógico, em termos de funcionalidades do Cassandra. Então, para executar mais nodos do Cassandra em mais máquinas, basta repetir nestas o comando citado em (a).

- b) Execute esses 2 comandos para criar mais 2 nodos no mesmo *host*:

Comando para nodo2: `docker run --name node2 -d -e CASSANDRA_SEEDS="$(docker inspect --format='{{.NetworkSettings.IPAddress }}' node1)" cassandra:3.11`

Comando para nodo3: `docker run --name node3 -d -e CASSANDRA_SEEDS="$(docker inspect --format='{{.NetworkSettings.IPAddress }}' node1)" cassandra:3.11`

A variável `CASSANDRA_SEEDS` indica a qual máquina o nodo2 e nodo3 estarão relacionados. O comando `docker inspect --format='{{.NetworkSettings.IPAddress}}' node1` retorna o valor IP da máquina nodo1. Mas esse IP é válido do ponto de vista da rede criada pelos *containers* Docker. O valor dado a `CASSANDRA_SEEDS` fica registrado no *container* Docker do nodo respectivo, num arquivo de configuração. Geralmente, cada container Cassandra adicionado no *host* local receberá endereço IP como 172.17.0.2, 172.17.0.3, .....172.17.0.X. Esses endereços não são visíveis por outras máquinas na rede Lan, naturalmente.

Veja como checar a configuração usada por um nodo Cassandra executado via *container* Docker:

c) Execute o comando: `docker exec -it node2 /bin/bash`

Em seguida, já dentro do container, veja o conteúdo do arquivo.

d) Execute : `cat ./etc/cassandra/cassandra.yaml` O arquivo `cassandra.yaml` contém muitas configurações importantes para o funcionamento da topologia em anel, para a performance das *queries*, etc. Para saber mais sobre que valores podem ser alterados e a vantagem disso, consulte o link:

<https://cassandra.apache.org/doc/latest/cassandra/configuration/index.html>

Saia do *container* Docker do Cassandra usado o comando `exit`.

e) Execute o comando `docker exec -it node1 nodetool status` para ver o *status* do anel formado. O resultado mostra quais nodos existem e se estão conectados corretamente ao nodo `seed node1`.

### Construção do anel de nodos em hosts diferentes

Para montar um anel com cada nodo em cada máquina na rede, o que é a solução mais conveniente para produção, tem-se que executar um *container* Docker para Cassandra numa máquina e os outros *containers* em outras máquinas. Apenas um *container* por máquina. Todos esses *containers* devem ter endereçamento IP com IP alcançável na rede onde se encontrarão os nodos. Esses nodos conversarão entre si usando o protocolo Gossip, o qual por default usa a porta 7000 em cada *container*. Ao executar os *containers*, uma configuração é alterada em cada Cassandra, para declarar através de qual IP o mesmo escutará requisições (configuração de *broadcast*). Por exemplo, se um *container* rodará na máquina com IP 192.168.10.6, então esse será o IP declarado pelo *container*/Cassandra e através dele o banco de dados respectivo receberá/escutará as requisições. Um dos nodos do anel passa a ser o nodo SEED. Todos os outros fazem referência a ele, o que implica em mais uma configuração no Cassandra respectivo de cada um desses nodos (configuração de SEED). Veja comandos abaixo.

f) Em uma máquina qualquer da rede, com Docker instalado, execute:

```
docker run -it -p 9042:9042 -p 7000:7000 --name node36 -d -e  
CASSANDRA_BROADCAST_ADDRESS=192.168.10.6 cassandra:3.11
```

A porta 7000 é usada pelo Gossip. A porta 9042 é *default* para receber comandos vindos de um cliente do Cassandra. Ex.: objeto Java que interage com o banco de dados.

g) Em outra máquina qualquer da rede, com Docker instalado, execute:

```
docker run --name node49 -d -e  
CASSANDRA_BROADCAST_ADDRESS=192.168.10.14 -p 7000:7000 -p  
9042:9042 -e CASSANDRA_SEEDS=192.168.10.6 cassandra:3.11
```

Esses 2 comandos acima executam os nodos node36 e node49 em 2 máquinas diferentes. Qualquer modificação que for feita no banco de dados, num desses nodos, por exemplo via um cliente Java se comunicando pela porta 9042, será replicada automaticamente no outro nodo, se o fator de replicação definido for  $\geq 2$ .

Nesse exemplo, o node36 é o *seed* e foi executado na máquina com IP 192.168.10.6. O outro node49, executado no IP 192.168.10.14, referencia o nodo *seed* através de sua configuração **CASSANDRA\_SEEDS**. Sem essa amarração usando *seeds*, não haveria um link entre os nodos e o anel lógico não existiria.

Com o anel de nodos pronto, pode-se acessar um deles e definir o banco, tabelas, linhas de dado em tabelas, chaves primárias com valores para partições e também de *cluster*, etc. Isso pode ser feito via código Java ou pela *shell* com um *prompt* já dentro de um dos containers que rodam os nodos Cassandra.

# Interação com o banco de dados

## Cassandra, via linha de comando

Para manipular o banco de dados Cassandra, rodando em *container* Docker, basta acessar o *prompt* de comandos do banco, dentro de um dos *containers* e executar comandos usando a linguagem *cqlsh*. Isso pode ser feito da seguinte forma:

- a) Entrar no *container* Cassandra:

```
docker exec -it node1 cqlsh
```

- b) Execute o comando para definir o nome do banco de dados no seu cluster:

```
CREATE KEYSPACE IF NOT EXISTS store WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : '1' };
```

Se precisar de um fator de replicação maior que um, basta mudar o valor nesse comando. O banco de dados criado nesse caso se chama *store*. Repare que os comandos *cqlsh* se parecem com SQL, daí o motivo de conhecer SQL ser pré-requisito para entender esse documento.

- c) Em seguida, pode-se criar uma tabela para esse banco de dados, da seguinte forma:

```
CREATE TABLE IF NOT EXISTS store.shopping_cart (
    userid text PRIMARY KEY,
    item_count int,
    last_update_timestamp timestamp
);
```

Conforme esse comando acima, a tabela se chama *shopping\_cart* e participa do banco de dados chamado *store*. Essa tabela tem 3 colunas. Ou seja, cada linha de dados definida aqui terá 3 colunas de informações, independentemente de em qual partição estiver presente a linha. O valor *userid* é definido como chave primária (pk). Isso significa que, para cada linha de dados, o *userid*, que é um texto, será usado para gerar um *token*. Então cada linha terá um *token* particular. E cada *token* especificará em qual partição a linha de dados será posta. Caso existam muitas linhas nessa tabela e caso a geração de tokens seja uniformemente distribuída para as partições, os dados estarão bem distribuídos entre os nodos do cluster. Então, se um *text* é uma boa fonte de geração de *tokens* uniformemente distribuídos, será conveniente realmente definir *userid* como pk. Se não for uma boa fonte para *tokens*, o projetista do banco de dados deverá pensar em outro tipo de pk. Essa tabela acima contém apenas uma coluna na pk, que será então a chave de partição, não havendo aqui chave para *cluster*.

Para ver o log de um *container* em execução, fora do *container*, use o comando :

```
docker logs <node name>
```

d) O próximo comando, também a ser executado no *prompt* do Cassandra, mostra como inserir duas linhas de dados na tabela definida. Reparar que é extremamente semelhante a SQL. A primeira linha a ser inserida e a segunda podem ficar em partições diferentes, caso os *tokens* gerados impliquem em nodos diferentes do anel. Quanto maior é a quantidade de nodos no anel, maior é o número de partições, maior é a chance desses 2 *tokens* apontarem para nodos diferentes.

```
INSERT INTO store.shopping_cart (userid, item_count, last_update_timestamp)  
VALUES ('9876', 2, toTimeStamp(now()));
```

```
INSERT INTO store.shopping_cart (userid, item_count, last_update_timestamp)  
VALUES ('1234', 5, toTimeStamp(now()));
```

Depois de inserir dados no banco de dados, algumas consultas podem ser feitas, ainda no *prompt* do Cassandra em *container*. Por exemplo:

- **DESCRIBE keyspaces;** Descreve quais são os banco de dados existentes no *cluster* corrente.
- **SELECT \* FROM system schema.keyspaces;** Retorna detalhes sobre as *keyspaces*.
- **Use <keyspace name>;** Os próximos comandos são sobre essa *keyspace*.
- **DESCRIBE tables;** Mostra quais são as tabelas da *keyspace/banco* corrente.
- **Describe <table name>;** Descreve detalhes de uma tabela da *keyspace*.
- **Drop table <table name>;** Destroi completamente uma tabela do banco de dados;
- **Drop keyspace <keyspace name>;** Destroi completamente o banco de dados;

Esses comandos com *cqlsh* não são *case sensitive*.

# Manipulando o banco de dados

## Cassandra com Java

Para usar Java e Cassandra, pode-se usar uma biblioteca que facilita a conexão e execução de *queries* no banco de dados. Essa biblioteca contém *driver* JDBC (Java Database Connection). Existem vários *drivers* construídos, porém o *driver* oficialmente suportado pelo Apache Cassandra se encontra no endereço:

[https://docs.datastax.com/en/driver-matrix/doc/driver\\_matrix/common/driverMatrix.html](https://docs.datastax.com/en/driver-matrix/doc/driver_matrix/common/driverMatrix.html).

Então, para criar um projeto Java, pode-se usar por exemplo o Eclipse. Para tal, basta criar um projeto Maven Java. No arquivo **pom.xml**, usar um código baseado nesse abaixo:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.inatel.icc.hpe</groupId>
  <artifactId>cassandra-teste-inical</artifactId>
  <version>1.0</version>
  <name>PrototipoCassandra</name>
  <description>Protótipo para iniciar aprendizado com Cassandra e Java</description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.datastax.cassandra</groupId>
      <artifactId>cassandra-driver-core</artifactId>
      <version>3.1.0</version>
    </dependency>

    <dependency>
      <groupId>com.datastax.oss</groupId>
      <artifactId>java-driver-core</artifactId>
      <version>4.6.0</version>
    </dependency>

    <dependency>
      <groupId>org.cassandraunit</groupId>
      <artifactId>cassandra-unit</artifactId>
      <version>3.0.0.1</version>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <scope>provided</scope>
      <version>1.18.20</version>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
```

```

<artifactId>junit</artifactId>
<version>4.12</version>
<scope>test</scope>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.13.0</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.13.0</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.13.0</version>
</dependency>
</dependencies>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <artifactId>maven-clean-plugin</artifactId>
                <version>3.1.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-resources-plugin</artifactId>
                <version>3.0.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.1</version>
            </plugin>
            <plugin>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.0.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-install-plugin</artifactId>
                <version>2.5.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-dependency-plugin</artifactId>
            </plugin>
            <plugin>
                <artifactId>maven-deploy-plugin</artifactId>
                <version>2.8.2</version>
            </plugin>
            <plugin>
                <artifactId>maven-site-plugin</artifactId>
                <version>3.7.1</version>
            </plugin>
            <plugin>
                <artifactId>maven-project-info-reports-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>br.inatel.icc.main.Principal</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>

```

```

        </configuration>
    </plugin>
</plugins>
</pluginManagement>

<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <!--version>3.8.0</version -->
        <configuration>
            <source>1.8</source>
            <target>1.8</target>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <version>3.0.1</version>
        <executions>
            <execution>
                <id>attach-sources</id>
                <goals>
                    <goal>jar</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.0.1</version>
        <configuration>
            <javadocExecutable>${java.home}/bin/javadoc</javadocExecutable>
        </configuration>
    </plugin>

    <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <!-- version>3.0.2</version-->
        <configuration>
            <archive>
                <manifest>
                    <addClasspath>true</addClasspath>
                    <mainClass>br.inatel.icc.main.Principal</mainClass>
                    <classpathPrefix>lib/</classpathPrefix>
                </manifest>
            </archive>
        </configuration>
    </plugin>
    <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <!-- version>3.0.2</version-->
        <configuration>
            <outputDirectory>${project.build.directory}/config</outputDirectory>
            <resources>
                <resource>
                    <directory>./config</directory>
                    <filtering>true</filtering>
                </resource>
            </resources>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
            <execution>
                <id>copy-dependencies</id>
                <phase>prepare-package</phase>
                <goals>
                    <goal>copy-dependencies</goal>
                </goals>
                <configuration>
                    <outputDirectory>${project.build.directory}/lib</outputDirectory>

```

```

        </configuration>
    </execution>
</executions>
</plugin>

</plugins>
</build>

</project>

```

Supondo que um banco de dados sobre livros será manipulado, o seguinte código abaixo utiliza o Cassandra:

Será necessário uma classe para Livro. O código pode ser o seguinte:

```

package br.inatel.icc.main;

import java.util.UUID;

public class Livro {

    public UUID getId() {
        return id;
    }
    public void setId(UUID id) {
        this.id = id;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getAutor() {
        return autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }

    public Livro() {
    }

    public Livro (UUID uuld, String strTitulo, String strAutor) {
        id = uuld;
        titulo = strTitulo;
        autor = strAutor;
    }

    private UUID id;
    private String titulo;
    private String autor;
}

```

Em seguida uma classe para prover a conexão com o banco de dados pode ser feita da seguinte forma:

```

package br.inatel.icc.main;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Cluster.Builder;
import com.datastax.driver.core.ResultSet;

```

```

import com.datastax.driver.core.Session;
import com.datastax.driver.core.exceptions.InvalidConfigurationInQueryException;

public class CassandraConnector {

    private Cluster cluster;
    private Session session;
    private final String KEY_SPACE;
    private final String TABLE_NAME;

    public CassandraConnector(String keyspaceName, String table) {
        KEY_SPACE = keyspaceName;
        TABLE_NAME = table;
    }

    public boolean conexao(String node) {
        try {
            Builder b = Cluster.builder().addContactPoint(node);
            b.withPort(9042);
            cluster = b.build();
            session = cluster.connect();
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public void fechar() {
        session.close();
        cluster.close();
    }

    public boolean criarKeyspace(String replicationStrategy, int replicationFactor) {
        String query = "CREATE KEYSPACE IF NOT EXISTS " + KEY_SPACE + " WITH replication = {'class':'" +
replicationStrategy + "','replication_factor':" + replicationFactor + "};";
        session.execute(query);
        ResultSet result = session.execute("SELECT * FROM system_schema.keyspaces;");
        List<String> matchedKeyspaces = result.all().stream().filter(r ->
r.getString(0).equals(KEY_SPACE)).map(r ->
r.getString(0)).collect(Collectors.toList());
        return matchedKeyspaces.size() == 1;
    }

    public void criarTabela() {
        String query = "CREATE TABLE IF NOT EXISTS " + KEY_SPACE + "." + TABLE_NAME + "(id uuid
PRIMARY KEY, titulo text, autor text);";
        session.execute(query);
    }

    public void inserir(Livro livro) {
        String query = "INSERT INTO " + KEY_SPACE + "." + TABLE_NAME + "(id, titulo, autor) VALUES (" +
livro.getId() + ", " + livro.getTitulo() + ", " + livro.getAutor() + ")";
        session.execute(query);
    }

    public List<Livro> getAll() {
        String query = "SELECT * FROM " + KEY_SPACE + "." + TABLE_NAME;
        ResultSet rs = session.execute(query);
        List<Livro> livros = new ArrayList<>();
        rs.forEach(r -> { livros.add(new Livro(r.getUUID("id"), r.getString("titulo"), r.getString("autor"))); });
        return livros;
    }

    public void eliminarTabela() {
        String query = "DROP TABLE IF EXISTS " + KEY_SPACE + "." + TABLE_NAME;
        session.execute(query);
    }

    public void eliminarKeyspace() {
        String query = "DROP KEYSPACE " + KEY_SPACE;
    }
}

```

```

        try {
            session.execute(query);
        }
        catch (InvalidConfigurationInQueryException e) {
        }
    }
}

```

Alguns pontos na classe acima devem ser observados:

Essa classe permite conectar com qualquer um dos bancos de dados (*keyspace*) pertencentes a um *cluster*. E para ter acesso a tal *cluster*, basta citar um nodo do *cluster* no método Conexão. Reparar que a conexão usa a porta **default 9042**. Pode-se também, através de tal conexão, criar um novo *keyspace*. E uma tabela para o *keyspace* escolhido também pode ser criada. Ou seja, não é necessário manipular o banco de dados via *prompt*, quando se pode fazer isso via a linguagem de programação. Essa classe permite fazer operações de CRUD no banco, por exemplo.

Em seguida, o código de uma classe para testar tudo isso pode ser feito da seguinte forma:

```

package br.inatel.icc.main;

import java.util.List;
import java.util.UUID;

import com.datastax.driver.core.utils.UUIDs;

public class Principal {

    private final String keyspaceName = "livraria";
    private CassandraConnector cc;

    public static void main(String[] args) {
        new Principal().executar();
    }

    private void executar() {
        cc = new CassandraConnector(keyspaceName, "livro");
        if (cc.conexao("localhost")) { //IP do nodo Cassandra a receber as modificações.
            passoFatal();
            passo1();
            passo2();
            passo3();
            passo4();
            cc.fechar();
        }
    }

    private void passo1() {
        System.out.println("Criar o KeySpace");
        if (cc.criarKeyspace("SimpleStrategy", 1)) {
            System.out.println("KeySpace criado");
        }
    }

    private void passo2() {
        System.out.println("Criar a Tabela");
        cc.criarTabela();
    }
}

```

```

private void passo3() {
    System.out.println("Adicionar Registros");

    UUID id1 = UUIDs.timeBased();

    Livro firstLivro = new Livro(id1, "O Tempo e o Vento", "Érico Veríssimo");

    cc.inserir(firstLivro);
    cc.inserir(new Livro(UUIDs.timeBased(), "Mentiras que os Homens Contam", "Luis Fernando
Veríssimo"));
    cc.inserir(new Livro(UUIDs.timeBased(), "Vidas Secas", "Graciliano Ramos"));
    cc.inserir(new Livro(UUIDs.timeBased(), "Auto da Comadecida", "Ariano Suassuna"));
}

private void passo4() {
    System.out.println("Mostrar Registros");
    List<Livro> livros = cc.getAll();
    for (Livro livro : livros) {
        System.out.println(livro.getTitulo() + " " + livro.getAutor() + " " + livro.getId());
    }
}

private void passoFatal() {
    cc.eliminarTabela();
    cc.eliminarKeyspace();
}

}

```

Para testar esse código, deve-se fazer o seguinte:

Acesse a pasta onde se encontra o projeto;

Execute [mvn clean](#)

Em seguida execute [mvn install](#)

Em seguida execute [java -jar <...projeto.jar>](#)

Essa execução irá contatar o nodo Cassandra rondando em *localhost*, num *container* Docker. Irá criar os dados e depois listar.

# Referências consultadas:

**Cassandra com Java e Python,** Anselmo. F., Versão 1.1, 24 de outubro de 2021,  
[https://www.academia.edu/49113529/Cassandra\\_com\\_Java\\_e\\_Python](https://www.academia.edu/49113529/Cassandra_com_Java_e_Python), acessado em  
23/03/2022. PDF.

<https://developer.ibm.com/br/tutorials/ba-set-up-apache-cassandra-architecture/>

<https://github.com/beccam/quickstart-java>

[https://hub.docker.com/\\_/cassandra](https://hub.docker.com/_/cassandra)

[https://cassandra.apache.org/doc/latest/cassandra/getting\\_started/index.html](https://cassandra.apache.org/doc/latest/cassandra/getting_started/index.html)

[https://www.digitalocean.com/community/tutorials/how-to-run-a-multi-node-cluster-data  
base-with-cassandra-on-ubuntu-14-04](https://www.digitalocean.com/community/tutorials/how-to-run-a-multi-node-cluster-data-base-with-cassandra-on-ubuntu-14-04)