

## Banco de Dados NoSQL

# Aula 8: Modelo de dados NoSQL baseado em colunas – parte II

## Apresentação

---

Exploraremos ainda mais a linguagem CQL com os comandos da linguagem de manipulação de dados nas operações de inclusão, alteração e exclusão, bem como os comandos de seleção e suas variações.

# Objetivo

---

- Escrever comandos CQL para manipulação de dados: insert, update e delete;
- Escrever comandos CQL para consultas simples e com o uso de funções;
- Escrever comandos CQL para consultas envolvendo agregações.

## Introdução

---

O Cassandra é um dos bancos de dados orientado em colunas (ELMASRI; NAVATHE, 2018) mais usados. Entre os motivos que o levaram até esse nível, podemos indicar a linguagem de consulta CQL, considerando o uso, principalmente, nas operações de recuperação de dados por meio de comandos SELECT.

## Comandos CQL para manipulação de dados: insert, update e delete

---

Antes de começarmos a trabalhar nos exemplos das operações de manipulação, devemos destacar duas diferenças importantes entre o modelo relacional e o orientado a colunas.

### Modelo relacional

Nesse modelo, normalmente, começamos a criação do banco de dados com o projeto e depois avaliamos as consultas que podem ser respondidas pelo modelo.



### Modelo orientado a colunas

O início do projeto é feito com o modelo de consultas e não com o modelo de dados, possibilitando que os dados sejam organizados para responder a essas consultas de forma eficiente.

## Comentário

A ideia é que, após a identificação das consultas principais ao uso do banco de dados, sejam criadas as tabelas necessárias para dar suporte a elas. Assim, será criada uma tabela para atender a cada consulta identificada. Dessa forma, o aumento do desempenho justificará a criação de múltiplas tabelas e, é claro, com redundância de dados.

Para melhor entendimento dessa visão diferenciada da modelagem do banco de dados, vamos criar um exemplo. Assim, primeiramente vamos criar uma tabela geral com os dados dos clientes. Seu código ficaria assim:

```
CREATE TABLE cliente (  
  
    codigo uuid primary key ,  
  
    nome varchar ,  
  
    login varchar ,  
  
    email varchar ,  
  
    cidade varchar ) ;
```

Agora, considerando como requisito de dados a recuperação dos clientes por cidades, será apresentado em seguida o código para a criação de uma tabela com o objetivo de atender a essa demanda da aplicação. Vejamos:

```
CREATE TABLE clientes_por_cidade (  
  
    login varchar PRIMARY KEY,  
  
    cidade varchar ) ;
```

Para a inclusão de registros na tabela clientes\_por\_cidade, pode-se usar também o comando INSERT que avalia a existência de registros com o mesmo valor no campo criado como chave primária. Se o valor não existir, o registro será incluído e uma mensagem de confirmação indicará o sucesso na operação com o retorno True, como abaixo:

```
INSERT INTO clientes_por_cidade ( login, cidade ) VALUES ( 'ana', 'Rio de Janeiro' )  
  
IF NOT EXISTS ;  
  
[applied]  
  
-----  
  
True
```

No caso da existência de um registro que já contenha o valor da chave primária na tabela, a mensagem de retorno indicará um False e os dados não inseridos.

```
[applied] | login | cidade
```

```
-----+-----+-----
```

```
False | ana | Rio de Janeiro
```

Se fosse identificada a necessidade de consultar os clientes por meio do atributo e-mail, outra tabela poderia ser criada considerando o atendimento a essa demanda. O código é apresentado a seguir:

```
CREATE TABLE clientes_por_email ( email varchar PRIMARY KEY, login varchar ) ;
```


**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

**Além da diferença entre termos de modelagem de dados, um outro conceito fundamental para o uso de um banco de dados de colunas é o conceito de partição de dados (SADALAGE; FOWLER, 2013). Os sistemas distribuídos armazenam os dados recebidos em partições.**

Todas as operações comuns nesse tipo de sistema, como a replicação, a distribuição e a indexação de dados, são possíveis de serem implementadas apenas quando há a definição de partição(ões), ou seja, podemos considerar como uma unidade de trabalho.

No Cassandra, a chave primária pode ser composta por duas chaves consideradas especiais:

- Chave de partição (*partition key*); e
- Chave de armazenamento em clusters (*clustering columns*), de uma forma opcional.

 Clique nos botões para ver as informações.

[Chave de partição](#)



A chave de partição tem por objetivo facilitar o controle de distribuição dos dados de modo uniforme pelos *clusters*.

O particionamento de dados faz uso de um atributo da tabela para ser configurado como a "chave de partição", permitindo o agrupamento dos dados em partições distintas. O tamanho máximo de uma partição no Cassandra não deve ultrapassar o valor de 100 MB e, de uma forma ideal, deve ser inferior a 10 MB.

Já a outra chave, que é a chave de armazenamento em *clusters*, também conhecida como chave de colunas de armazenamento em *clusters*, tem como tarefa facilitar o controle do armazenamento em *clusters*.

Com essa segunda chave, é possível obter uma melhor organização dos dados de uma partição possibilitando a criação de consultas eficientes. Para exemplificar, vamos criar uma tabela para o controle de logs de um sistema de acesso dos usuários de filiais.

A tabela log faz uso do atributo de tipo *timeuuid* (*universal unique identifier*), uma variante de identificador único universal que inclui informações de tempo.

```
CREATE TABLE log (
  filial varchar,
  data timestamp ,
  datahora timeuuid ,
  login varchar ,
  PRIMARY KEY ( ( filial , data ), datahora ) );
```

Os campos referem-se ao nome da filial, a data/hora do log do usuário, a data/hora em formato *timeuuid* e o nome de login do usuário. A chave primária da tabela será composta pela chave de partição formada pelos atributos *filial* e pelo atributo *data*; a chave de armazenamento de *clusters* é o campo *datahora*.

A Figura 1 apresenta a estrutura da tabela log em que se pode-se observar a composição da chave primária em destaque. Para o cadastro dos registros que aparecem na figura, pode-se executar os comandos de inclusão apresentados a seguir.

Figura 1 - Exemplo de tabela com chave de partição e chave de armazenamento em cluster

		filial	Data	<u>datahora</u>	login
Nó 1	{	'filial_2'	2020-11-17 00:00:00+0000	7bd8b5b1-28df-11eb-84c0-793c4ada966f	luis
		'filial_2'	2020-11-16 00:00:00+0000	7bd9ee31-28df-11eb-84c0-793c4ada966f	<u>ana</u>
		'filial_2'	2020-11-16 00:00:00+0000	7bdad891-28df-11eb-84c0-793c4ada966f	rui
Nó 2	{	'filial_1'	2020-11-16 00:00:00+0000	7bd644b0-28df-11eb-84c0-793c4ada966f	<u>ana</u>
		'filial_1'	2020-11-16 00:00:00+0000	7bd6e0f1-28df-11eb-84c0-793c4ada966f	<u>nei</u>
		'filial_1'	2020-11-16 00:00:00+0000	7bd7cb51-28df-11eb-84c0-793c4ada966f	lia

Chave de partição
Chave de armazenamento em clusters

```

INSERT INTO log ( filial , data , datahora , login ) VALUES ( 'filial_1 ' , toTimestamp(toDate(now())) , now() ,
'ana' ) ;
INSERT INTO log ( filial , data , datahora , login ) VALUES ( 'filial_1' , toTimestamp(toDate(now())) , now() ,
'nei' ) ;
INSERT INTO log ( filial , data , datahora , login ) VALUES ( 'filial_1' , toTimestamp(toDate(now())) , now() ,
'lia' ) ;
INSERT INTO log ( filial , data , datahora , login ) VALUES ( 'filial_2' , toTimestamp(toDate(now())) , now() ,
'luis' ) ;
INSERT INTO log ( filial , data , datahora , login ) VALUES ( 'filial_2' , toTimestamp(toDate(now())) , now() ,
'ana' ) ;
INSERT INTO log ( filial , data , datahora , login ) VALUES ( 'filial_2 ' , toTimestamp(toDate(now())) , now() ,
'rui' ) ;

```

Observe como os dados ficaram armazenados na tabela log com um select simples:

```
SELECT * FROM log ;
```

filial	data	datahora	login	
filial_2	2020-11-17	00:00:00+0000	7bd8b5b1-28df-11eb-84c0-793c4ada966f	luis
filial_2	2020-11-17	00:00:00+0000	7bd9ee31-28df-11eb-84c0-793c4ada966f	ana
filial_2	2020-11-17	00:00:00+0000	7bdad891-28df-11eb-84c0-793c4ada966f	rui
filial_1	2020-11-17	00:00:00+0000	7bd644b0-28df-11eb-84c0-793c4ada966f	ana
filial_1	2020-11-17	00:00:00+0000	7bd6e0f1-28df-11eb-84c0-793c4ada966f	nei
filial_1	2020-11-17	00:00:00+0000	7bd7cb51-28df-11eb-84c0-793c4ada966f	lia

Agora vamos verificar como aplicar o comando de exclusão (DELETE) no Cassandra por meio de exemplos, entendendo cada mensagem de erro gerada pelo banco de dados. Primeiro, vamos tentar excluir dois registros informando no comando os valores armazenados no campo datahora:

```

DELETE FROM log

WHERE datahora = '7bdad891-28df-11eb-84c0-793c4ada966f'

OR datahora = '7bd6e0f1-28df-11eb-84c0-793c4ada966f' ;

```

O comando está aparentemente correto, porém apresenta o erro apresentado a seguir. Isso é devido ao fato de a CQL não ter suporte para o uso do operador lógico OU ( OR ) em suas consultas. Assim, observe que na mensagem retorna " Syntax error in CQL query ":

```

SyntaxException: <ErrorMessage code=2000 [Syntax error in CQL query] message="line 3:0 missing EOF at 'OR'
(...logWHERE datahora = '7bdad891-28df-11eb-84c0-793c4ada966f'[OR] datahora...)">

```

Agora vamos tentar o mesmo comando, substituindo o OR por um IN, como apresentado a seguir:

```
DELETE FROM log

WHERE datahora IN ( '7bdad891-28df-11eb-84c0-793c4ada966f' ,

'7bd6e0f1-28df-11eb-84c0-793c4ada966f' ) ;
```

O comando ainda apresenta erro! O motivo agora é diferente, o problema é na indicação de parte da chave primária para condição. Lembra que a chave primária é formada pelos atributos filial e data? A mensagem indica que foi informada apenas parte da chave primária no comando de exclusão " for PRIMARY KEY part datahora "

```
SyntaxException: <ErrorMessage code=2000 [Syntax error in CQL query] message="line 2:18 no viable alternative at
input '7bdad891-28df-11eb-84c0-793c4ada966f' (DELETE FROM logWHERE [datahora] IN...)">
```

Agora iremos tentar excluir os registros informando os valores de dois logins de usuários. Vejamos:

```
DELETE FROM log

WHERE login IN ( 'rui' , 'nei' ) ;
```

Como resultado, temos uma nova mensagem de erro. O motivo agora é que o campo login não faz parte da chave primária. Segue o comando com a mensagem explicativa " Non PRIMARY KEY login found " :

```
InvalidRequest: code=2200 [Invalid query] message="Non PRIMARY KEY login found in where clause"
```

Em outra tentativa, vamos informar no comando de exclusão o valor completo da chave primária. O comando apagará os registros da filial\_1 e em uma determinada data. Ele será então executado com sucesso. Vejamos:

```
DELETE FROM log

WHERE filial = 'filial_1'

AND data = '2020-11-17 00:00:00+0000' ;
```

Vamos verificar as exclusões com o select simples abaixo:

```
SELECT * FROM log ;
```

O resultado após e exclusão é apresentado abaixo:

```
SELECT * FROM log ;
```

O resultado após e exclusão é apresentado abaixo:

filial	data	datahora	login
-----+-----+-----+-----			
filial_2	2020-11-17 00:00:00+0000	7bd8b5b1-28df-11eb-84c0-793c4ada966f	luis
filial_2	2020-11-17 00:00:00+0000	7bd9ee31-28df-11eb-84c0-793c4ada966f	ana
filial_2	2020-11-17 00:00:00+0000	7bdad891-28df-11eb-84c0-793c4ada966f	rui

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

Na Aula 7, apresentamos alguns exemplos de comandos para a atualização (UPDATE) dos registros de uma tabela. Foi feita a opção de não apresentar esse tipo de comando na presente aula, por ser redundante e para explorar mais detalhes do comando SELECT.

## Comandos CQL para consultas simples e com o uso de funções

As linguagens SQL e a CQL apresentam sintaxes muito semelhantes. Porém, deve ser destacado que, apesar dessa semelhança, o que realmente difere é a estrutura dos bancos de dados.

Comentário



Assim, para o uso correto dos comandos de consulta que usam a cláusula WHERE, deve-se compreender que essas diferenças foram implementadas com o objetivo de evitar a criação de consultas sem eficiência.

Explorando mais um pouco os conceitos das chaves usadas no Cassandra, as colunas de chave de partição e as colunas de chave de armazenamento em clusters formam a chave primária de uma linha na tabela. Esse segundo tipo de chave é usado para agrupar os dados de uma partição, possibilitando dessa forma uma recuperação muito mais eficiente dos dados.

Assim, deve ser compreendido que cada parte de uma linha é usada de forma diferente e isso precisa ser considerado na escrita da consulta que faz uso da cláusula WHERE.

Para exemplificar vamos criar a tabela usuario com a seguinte estrutura e realizar as inclusões de registros a seguir:

```
CREATE TABLE usuario (  
  
    codigo uuid,  
  
    nome varchar ,  
  
    login varchar ,  
  
    email varchar ,  
  
    cidade varchar,  
  
    primary key ( codigo, cidade ) );  
  
INSERT INTO usuario ( codigo, nome, login, email, cidade ) values ( uuid(), 'Ana', 'ana',  
'ana@correioeletronico.com', 'Rio de Janeiro' ) ;  
  
INSERT INTO usuario ( codigo, nome, login, email, cidade ) values ( uuid(), 'Nei', nei,  
'nei@correioeletronico.com', 'Rio de Janeiro' ) ;  
  
INSERT INTO usuario ( codigo, nome, login, email, cidade ) values ( uuid(), 'Rui', rui,  
'rui@correioeletronico.com', 'Rio Grande do Sul' ) ;  
  
INSERT INTO usuario ( codigo, nome, login, email, cidade ) values ( uuid(), 'Lia', lia,  
'lia@correioeletronico.com', 'Rio Grande do Sul' ) ;  
  
INSERT INTO usuario ( codigo, nome, login, email, cidade ) values ( uuid(), 'Maria', 'ana',  
'maria@correioeletronico.com', 'Rio Grande do Norte' ) ;
```

Execute um comando simples para ver o resultado:

```
SELECT * FROM usuario ;  
  
codigo | cidade | email | login | nome  
-----+-----+-----+-----+-----  
76ddb15b-6055-4b35-8eb1-528b9d5a47d5 | Rio Grande do Sul | lia@correioeletronico.com | ana | Lia  
  
4267f6a7-1eb1-442f-966b-53cf1568646c | Rio de Janeiro | ana@correioeletronico.com | ana | Ana
```

33a3c936-7766-46f0-8967-c92136f33f2c | Rio Grande do Sul | rui@correioeletronico.com | rui | Rui

7fac0ac2-5f92-44b6-8069-eace87b50fd5 | Rio de Janeiro | nei@correioeletronico.com | nei | Nei

c36ce28f-69ae-489f-8555-04cc9bc209b4 | Rio Grande do Norte | maria@correioeletronico.com | maria | Maria

No caso de consultas aplicadas nas chaves de partição, as restrições permitem apenas o uso de dois tipos de operadores: o = ( igual ) e o IN. Pode-se usar o operador IN em qualquer parte (coluna) de chave de partição. Vamos exemplificar com uma consulta para listar os dados de um usuário pelo valor do código em uuid (sem aspas):

```
SELECT * FROM usuario WHERE codigo = 4267f6a7-1eb1-442f-966b-53cf1568646c ;
```

codigo	cidade	email	login	nome
4267f6a7-1eb1-442f-966b-53cf1568646c	Rio de Janeiro	ana@correioeletronico.com	ana	Ana

No caso de consultas aplicadas nas chaves de partição, as restrições permitem apenas o uso de dois tipos de operadores: o = ( igual ) e o IN. Pode-se usar o operador IN em qualquer parte (coluna) de chave de partição. Vamos exemplificar com uma consulta para listar os dados de um usuário pelo valor do código em uuid (sem aspas):

```
SELECT * FROM usuario WHERE codigo = 4267f6a7-1eb1-442f-966b-53cf1568646c ;
```

codigo	cidade	email	login	nome
4267f6a7-1eb1-442f-966b-53cf1568646c	Rio de Janeiro	ana@correioeletronico.com	ana	Ana

O mesmo resultado anterior pode ser obtido com a consulta usando o IN :

```
SELECT * FROM usuario

WHERE codigo IN (4267f6a7-1eb1-442f-966b-53cf1568646c)

AND cidade IN ('Rio de Janeiro');
```

Agora vamos escrever uma consulta relativamente simples, mas que apresentará um erro. Trata-se de uma consulta para

recuperar os dados de um usuário pelo seu nome. Vejamos:

```
SELECT * FROM usuario WHERE nome = 'Lia' ;
```

A mensagem de erro é a seguinte:


```
InvalidRequest: code=2200 [Invalid query] message="No secondary indexes on the restricted columns support the provided operators: "
```

Para que esse tipo de consulta possa ser executado, precisa-se de um índice secundário, que pode ser criado como o comando a seguir:

```
CREATE INDEX ON usuario (nome) ;
```

Execute a consulta novamente e verificará o resultado.

#### Limitações de consultas CQL

 Clique no botão acima.

De forma resumida, podemos destacar as limitações impostas na linguagem de consulta Cassandra (CQL), que são:

- a. não tem suporte para consultas de agregação como, por exemplo, as que usam função como o valor máximo, o mínimo e a média (max, min e avg);
- b. não tem suporte para consultas a partir de partes de cadeias de caracteres;
- c. não tem suporte para agrupamento;
- d. não tem suporte para a construção e uso de junções;
- e. não suporta consultas com o comparador OR;
- f. não tem suporte para consultas de união e interseção;
- g. os campos comuns da tabela não podem ser filtrados sem criar o índice secundário e, por fim;
- h. consultas com operadores de comparação maior ( > ) e menor que ( < ) só podem ser executadas na chave de armazenamento em clusters.

# Comandos CQL para consultas envolvendo agregações

---

A construção de consultas envolvendo agregações necessita do acréscimo de uma permissão de habilitação de filtragem. Isso é feito na própria consulta, no final dela. O Cassandra alerta que isso pode causar uma performance imprevisível. Vejamos um exemplo para contar o número de usuários para uma determinada cidade:

```
SELECT count( nome ) as total  
  
FROM usuario  
  
WHERE cidade = 'Rio de Janeiro'  
  
allow filtering ;
```

O resultado é apresentado abaixo:

```
system.count(nome)  
  
-----  
  
2  
  
(1 rows)
```

**Atenção!** Aqui existe uma videoaula, acesso pelo conteúdo online

# Atividade

---

1. Sobre o código apresentado a seguir, podemos afirmar, exceto:

```
CREATE TABLE log_usuario (  
  login varchar ,  
  data timestamp ,  
  datahora timeuuid ,  
  PRIMARY KEY ( ( login , data ), datahora ) ) ;  
  
INSERT INTO log_usuario ( login, data, datahora ) VALUES ( 'aluno_1', toTimestamp(toDate(now())) , now() ) ;  
INSERT INTO log_usuario ( login, data, datahora ) VALUES ( 'aluno_2', toTimestamp(toDate(now())) , now() ) ;  
INSERT INTO log_usuario ( login, data, datahora ) VALUES ( 'aluno_3', toTimestamp(toDate(now())) , now() ) ;
```

- a) Existe uma chave primária tripla.
  - b) A função `toTimestamp(toDate(now()))` permite obter o valor timestamp.
  - c) A função `now()` permite o preenchimento do atributo `timeuuid`.
  - d) A tabela tem dois atributos no campo chave de partição.
  - e) A tabela tem o campo `datahora` como chave de armazenamento em clusters.
- 

2. Assinale a afirmativa **correta** sobre como a consulta abaixo pode ser executada no Cassandra:

```
SELECT * FROM usuario WHERE nome = 'Lia' ;
```

- a) Não pode usar o `*`, como na SQL precisa da indicação dos atributos a serem exibidos.
  - b) Não pode ser executada, pois o banco trabalha apenas com o IN.
  - c) Necessita da criação de um índice secundário no campo `nome`.
  - d) Pode ser executada sem necessidade de qualquer recurso extra.
  - e) Só pode ser executada se o atributo `nome` fizer parte da chave de partição.
- 

3. Considerando a necessidade de uma consulta para a contagem de registros, com o uso da função `count()`, de acordo com uma determinada condição, ela pode ser realizada no Cassandra desde que:

- a) Não pode ser realizada, pois o Cassandra não tem agregações.
  - b) Pode ser executada sem nenhum recurso extra.
  - c) Pode ser executada com o uso de um índice secundário.
  - d) Uma permissão seja incluída no fim da consulta com um *allow filtering*.
  - e) Pode ser executada com o uso de um índice primário.
- 

## Notas

## Referências

---

ELMASRI, R.; NAVATHE, S.B. **Sistemas de banco de dados**. 6.ed. São Paulo: Pearson Education do Brasil, 2018.

## Próxima aula

---

- Características do banco de dados baseados em grafos;
- Vantagens e tipos de aplicações indicadas para uso do banco de dados em grafos;
- Comandos NOSQL para a definição da estrutura do banco de dados.

## Explore mais

---

- Leia o livro *Sistema de banco de dados*
- Leia o capítulo 10 do livro *NoSQL Essencial - Um Guia Conciso para o Mundo Emergente da Persistência Poliglota*