Cassandra com Java e Python

Fernando Anselmo
http://fernandoanselmo.orgfree.com/wordpress/

Versão 1.1 em 24 de outubro de 2021

Resumo

Apache Cassandra é um Sistema para Banco de Dados distribuído e altamente escalável de segunda geração, que reúne a arquitetura do DynamoDB, da Amazon Web Services e modelo de dados baseado no BigTable, do Google. O Modelo de Dados do Cassandra é um amplo armazenamento de colunas e essencialmente um híbrido entre o que conhecemos por valor-chave e um sistema de gerenciamento tabular. Consiste com um armazenamento de linha particionado permitindo uma consistência ajustável. Linhas são organizadas em tabelas; o primeiro componente da chave primária de uma tabela é a chave de partição; dentro de uma partição, as linhas são agrupadas pelas colunas restantes da chave. Outras colunas podem ser indexadas separadamente da chave primária. Neste tutorial veremos o que vem a ser o banco Cassandra [1] e como proceder sua utilização utilizando como pano de fundo a linguagem de programação Java [2] e Python [3].

1 Parte inicial

Avinash Lakshman, um dos autores do DynamoDB da Amazon, e Prashant Malik [1] desenvolveram inicialmente o Cassandra no Facebook para gerenciar o recurso de pesquisa da caixa de entrada do Facebook. O Facebook lançou o Cassandra como um projeto de código aberto no código do Google em julho de 2008. Em março de 2009, tornou-se um projeto da Incubadora Apache. Em 17 de fevereiro de 2010, ele se formou em um projeto de nível superior.

Figura 1: *Logo do Apache Cassandra*

A Arquitetura Cassandra consiste nos seguintes componentes:

[1] Os desenvolvedores nomearam o banco de dados em homenagem ao profeta mitológico **Trojan Cassandra**, com alusões clássicas à maldição de um oráculo.

- **Node (Nó)**. É o componente básico dos dados, uma máquina onde os dados são armazenados.
- **DataCenter**. Uma coleção de nós relacionados. Pode ser físico ou virtual.
- **Cluster**. Contém um ou mais *DataCenter*, ele pode se estender por vários locais.
- **Commit Log**. Cada operação de gravação é primeiro armazenada no *log* (registro) de confirmação. Normalmente é utilizado para recuperação de falhas.
- **MemTable**. Depois que os dados são gravados no *log* de confirmação, são armazenados em uma *Memory Table* (tabela em memória), que permanece lá até atingir o limite.
- **SSTable**. *Sorted-String Table* é um arquivo de disco que armazena dados da *MemTable* assim que atinge o limite. As SSTables são armazenadas em disco sequencialmente e mantidas para cada tabela do banco de dados.

1.1 Estratégias de replicação de dados

Uma das partes mais importantes para entedermos o Cassandra é sua capacidade de "Replicação dos Dados", isso não é opcional mas um recurso obrigatório para garantir que nenhum dado seja perdido devido a falha de hardware ou rede. Uma estratégia de replicação determina em quais nós colocar réplicas. Cassandra oferece duas estratégias de replicação diferentes.

Simple Strategy - Estratégia Simples

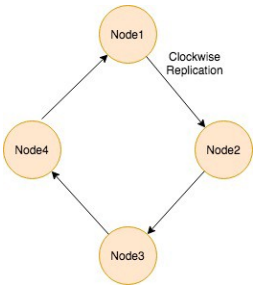

Figura 2: *Estrutura da Estratégia Simples*

Utilizada quando se possui um único *DataCenter*. É colocada a primeira réplica no nó selecionado pelo particionador. Um particionador determina como os dados são distribuídos pelos nós do cluster (incluindo suas réplicas). Depois disso, as réplicas restantes são colocadas no sentido horário no anel do nó.

Network Topology Strategy - Estratégia de topologia de rede

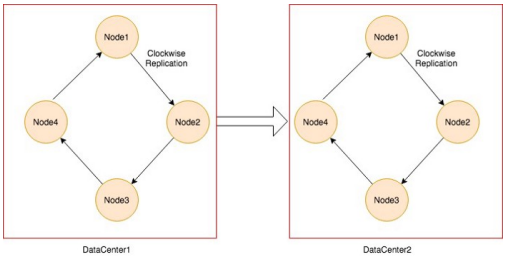

Figura 3: *Estrutura da estratégia de topologia de rede*

Utilizada quando se possui implantações em vários *Datacenters*. Essa estratégia coloca réplicas no mesmo *Datacenter*, percorrendo o anel no sentido horário até chegar ao primeiro nó em outro *rack*.

Isso ocorre porque às vezes podem ocorrer falhas ou problemas no *rack*. Então, as réplicas em outros nós podem fornecer dados. Esta estratégia é altamente recomendada para fins de escalabilidade e expansão futura.

1.2 Criar o contêiner Docker

A forma mais simples de termos o Apache Cassandra é através de um contêiner no Docker, assim facilmente podemos ter várias versões do banco instalada e controlar mais facilmente qual banco está ativo ou não. E ainda colhemos o benefício adicional de não termos absolutamente nada deixando sujeira em nosso sistema operacional ou áreas de memória.

Baixar a imagem oficial:
```
$ docker pull cassandra
```

Criar uma instância do banco em um contêiner:
```
$ docker run -it -p 9042:9042 --name meu-cassandra -d
-v /home/[seuUsuario]/cassandra/data/node1:/var/lib/cassandra/data cassandra
```

Nessa instância criada estamos associando a porta 9042 para acessarmos o banco e um volume em nossa máquina para armazenar os dados do contêiner.

Podemos acessar o Shell de comandos do Cassandra no contêiner (espere um pouco até o banco levantar):
```
$ docker exec -it meu-cassandra cqlsh
```

```
1 cqlsh> SELECT * FROM system_schema.keyspaces;
2 cqlsh> exit
```

Podemos parar o contêiner com:
```
$ docker stop meu-cassandra
```

Ou iniciá-lo novamente:
```
$ docker start meu-cassandra
```

1.3 Cassandra Shell (cqlsh) - a console de comandos

O Cassandra Shell ou simplesmente cqlsh, também conhecida como Console de Comandos, onde é possível realizar operações administrativas como consultas ou manutenções de dados através de comandos CQL (Cassandra Query Language) extremamente parecido com o SQL, porém temos algumas diferenças:

- **Keyspace**: Como um conjunto de dados é replicado, por exemplo, em quais *DataCenters* e quantas cópias. As Keyspaces contêm tabelas.
- **Table**: Esquema que mostra uma coleção de partições. As tabelas do Cassandra têm adição flexível de novas colunas às tabelas com tempo de inatividade zero. As tabelas contêm partições, que contêm linhas, que contêm colunas.
- **Partição**: Parte obrigatória da chave primária que todas as linhas do Cassandra devem ter. Todas as consultas de desempenho fornecem a chave de partição na consulta.
- **Linha**: Uma coleção de colunas identificadas por uma chave primária exclusiva composta pela chave de partição e, opcionalmente, por chaves de *clusters* adicionais.

- **Coluna**: Um único dado com um tipo que pertence a uma linha.

Mostrar as KeySpaces existentes:
```
> DESCRIBE keyspaces;
```

Também podemos ter detalhes sobre as KeySpaces:
```
> SELECT * FROM system_schema.keyspaces;
```

Criar uma *KeySpace*:
```
> CREATE KEYSPACE nomeKey
WITH replication = {'class':  'SimpleStrategy', 'replication_factor' :  1};
```

Criamos a *KeySpace* com uma replicação do tipo *Single Strategy*. Usar uma KeySpace:
```
> USE nomeKey;
```

Mostrar as tabelas existentes na KeySpace atual:
```
> DESCRIBE tables;
```

Agora vamos ver como o CQL torna-se extremamente similar ao SQL, criar uma Tabela:
```
> CREATE TABLE professor(matricula int PRIMARY KEY, nome text, cidade text);
```

Descrever os detalhes de uma tabela:
```
> DESCRIBE professor;
```

Adicionar novo campo na tabela (professor):
```
> ALTER TABLE professor ADD email text;
```

Remover campo na tabela (professor):
```
> ALTER TABLE professor DROP email;
```

Adicionar linhas na tabela existente (professor):
```
> INSERT INTO professor (matricula, nome, cidade)
VALUES (1, 'Fernando', 'Brasilia');
```

Adicionar linhas via JSON na tabela existente (professor):
```
> INSERT INTO professor
JSON '{"matricula":  2, "nome":  "Joana", "cidade":  "Recife"}';
```

Listar as linhas de uma tabela existente (professor):
```
> SELECT * FROM professor;
```

Listar linhas específicas de uma tabela existente (professor) - Uma instrução *SELECT* contém pelo menos uma cláusula e o nome da tabela na qual a seleção está ativada (CQL não faz junções ou subconsultas e, portanto, uma instrução *SELECT* se aplica apenas a uma única tabela). Para refinar usamos a cláusula *WHERE* e, opcionalmente, pode ter cláusulas adicionais para ordenar ou limitar os resultados. Porém, as consultas que requerem filtragem podem ser permitidas se o sinalizador ALLOW FILTERING for fornecido:
```
> SELECT * FROM professor WHERE nome = 'Fernando' ALLOW FILTERING;
```

Para evitar isso podemos criar um índice para a coluna que desejamos usar como pesquisa:
```
> CREATE INDEX ON professor(nome);
```

E utilizar a cláusula **WHERE** normalmente:
```
> SELECT * FROM professor WHERE nome = 'Fernando';
```

Quantas linhas de uma tabela existente (professor):
```
> SELECT count(*) FROM professor;
```

Modificar linha(s):
```
> UPDATE professor SET nome = 'Maria' WHERE matricula = 2;
```

Eliminar linha(s):
```
> DELETE FROM professor WHERE matricula = 2;
```

Eliminar uma tabela:
```
> DROP TABLE professor;
```

Eliminar uma KeySpace:
```
> DROP KEYSPACE nomeKey;
```

Se percebemos bem a única diferença do MongoDB para bancos relacionais é entendermos como é o relacionamento entre os objetos:

RDBS	CASSANDRA
Database	Keyspace
Tabela	Tabela
Chave Primária	Partição
Tupla	Linha
Dado	Coluna

Figura 4: *Comparativo entre Bancos SQL tradicionais (RDBS) e o Cassandra*

Para conhecer mais comandos da CQL, podemos acessar o seguinte endereço: `https://cassandra.apache.org/doc/latest/`.

2 Linguagem Java

Java é considerada a linguagem de programação orientada a objetos mais utilizada no Mundo, é a base para construção de ferramentas como Hadoop, Pentaho, Weka e muitas outras utilizados comercialmente. Foi desenvolvida na década de 90 por uma equipe de programadores chefiada por *James Gosling* para o projeto Green, na empresa Sun Microsystems - tornou-se nessa época como a linguagem que os programadores mais baixaram e o sucesso foi instantâneo. Em 2008 o Java foi adquirido pela Oracle Corporation.

2.1 Driver JDBC de Conexão

Para proceder a conexão com Java, é necessário baixar um 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`

Para utilizar o driver é necessário criar um projeto (Utilizaremos o **Spring Tool Suite 4**, porém pode utilizar qualquer outro editor de sua preferência).

No STS4 acessar a seguinte opção no menu: File ▷ New ▷ Java Project. Informar o nome do projeto (meucass), não esquecer de modificar a opção "Use an environment JRE"para a versão correta da Java Runtime desejada e pressionar o botão Finish. Se está tudo correto teremos a seguinte situação na aba *Project Explorer*.

Vamos convertê-lo para um projeto Apache Maven. Clicar com o botão direito do mouse no projeto e acessar a opção: Configure ▷ Convert to Maven Project. Na janela apenas pressione o botão *Finish*. Se tudo está correto observamos que o projeto ganhou uma letra **M** o que indica agora é um projeto padrão Maven. Então foi criado um arquivo chamado **pom.xml**.

Acessar este arquivo e antes da tag BUILD, inserir a tag DEPENDENCIES:

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

Observamos que na pasta **Maven Dependencias** foi baixado a versão 3.1.0 do driver JDBC do Cassandra.

2.2 Classe Livro

Estamos prontos para testarmos a conexão entre porém vamos criar uma classe chamada **Livro** no pacote **meucass** e inserir nesta a seguinte codificação:

```
package meucass;

import java.util.UUID;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;
import lombok.ToString;

@ToString
```

```
@AllArgsConstructor
public class Livro {
   @Getter @Setter private UUID id;
   @Getter @Setter private String titulo;
   @Getter @Setter private String autor;
}
```

Graças ao pacote Lombok eliminamos a necessidade de criar métodos padrões Gets/Sets, o construtor com os argumentos e método toString. Veja mais detalhes sobre esse pacote em: `https://projectlombok.org/`.

2.3 Classe Conexão

Agora vamos criar o escopo de uma classe que realiza ações básicas com o Banco de dados:

```
package meucass;

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;

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) {
```

```
41   String query = "CREATE KEYSPACE IF NOT EXISTS " + KEY_SPACE + " WITH replication =
       {'class':'" + replicationStrategy + "','replication_factor':" + replicationFactor +
       "};";
42   session.execute(query);
43   ResultSet result = session.execute("SELECT * FROM system_schema.keyspaces;");
44   List<String> matchedKeyspaces = result.all().stream().filter(r ->
       r.getString(0).equals(KEY_SPACE)).map(r ->
       r.getString(0)).collect(Collectors.toList());
45   return matchedKeyspaces.size() == 1;
46  }
47
48  public void criarTabela() {
49   String query = "CREATE TABLE IF NOT EXISTS " + KEY_SPACE + "." + TABLE_NAME + "(id
       uuid PRIMARY KEY, titulo text, autor text);";
50   session.execute(query);
51  }
52
53  public void inserir(Livro livro) {
54   String query = "INSERT INTO " + KEY_SPACE + "." + TABLE_NAME + "(id, titulo, autor)
       VALUES (" + livro.getId() + ", '" + livro.getTitulo() + "', '" + livro.getAutor() +
       "');";
55   session.execute(query);
56  }
57
58  public List<livro> getAll() {
59   String query = "SELECT * FROM " + KEY_SPACE + "." + TABLE_NAME;
60   ResultSet rs = session.execute(query);
61   List<Livro> livros = new ArrayList<>();
62   rs.forEach(r -> {
63    livros.add(new Livro(r.getUUID("id"), r.getString("titulo"), r.getString("autor")));
64   });
65   return livros;
66  }
67
68  public void eliminarTabela() {
69   String query = "DROP TABLE IF EXISTS " + KEY_SPACE + "." + TABLE_NAME;
70   session.execute(query);
71  }
72
73  public void eliminarKeyspace() {
74   String query = "DROP KEYSPACE " + KEY_SPACE;
75   session.execute(query);
76  }
77 }
```

O método construtor recebe duas variáveis o nome da *KeySpace* e Tabela. O método conexao() realiza a conexão com o banco a partir de um node e o valor da porta, cria uma sessão de conexão (variável *session*) retorna um lógico se conseguiu ou não realizar essa conexão. O método fechar() encerra a conexão com a sessão e o cluster.

Para os métodos de ações temos: criarKeyspace() responsável por criar a *KeySpace*. criarTabela() que cria a estrutura da nossa tabela. inserir() que a partir de um objeto livro adiciona este na tabela. getAll() retorna uma lista de livros que foram inseridos. eliminarTabela() e eliminarKeyspace() removem a tabela e *KeySpace* respectivamente.

Se pararmos um pouco para pensar, veremos que tirando o método conexão todos os outros se

comportam como qualquer outro driver de JDBC realizando as ações padrões como se estivéssemos usando um banco Postgres ou MySQL.

2.4 Classe Teste

Por fim vamos criar uma classe que testa toda essa conexão:

```
1  package meucass;
2
3  import java.util.List;
4  import com.datastax.driver.core.utils.UUIDs;
5
6  public class Principal {
7   private final String keyspaceName = "livraria";
8   private CassandraConnector cc;
9
10  public static void main(String[] args) {
11   new Principal().executar();
12  }
13
14  private void executar() {
15   cc = new CassandraConnector(keyspaceName, "livro");
16   if (cc.conexao("localhost")) {
17    passo1();
18    passo2();
19    passo3();
20    passo4();
21    passoFatal();
22    cc.close();
23   }
24  }
25
26  private void passo1() {
27   System.out.println("Criar o KeySpace");
28   if (cc.criarKeyspace("SimpleStrategy", 1)) {
29    System.out.println("KeySpace criado");
30   }
31  }
32
33  private void passo2() {
34   System.out.println("Criar a Tabela");
35   cc.criarTabela();
36  }
37
38  private void passo3() {
39   System.out.println("Adicionar Registros");
40   cc.inserir(new Livro(UUIDs.timeBased(), "O Tempo e o Vento", "Érico Veríssimo"));
41   cc.inserir(new Livro(UUIDs.timeBased(), "Mentiras que os Homens Contam", "Luis
       Fernando Veríssimo"));
42   cc.inserir(new Livro(UUIDs.timeBased(), "Vidas Secas", "Graciliano Ramos"));
43   cc.inserir(new Livro(UUIDs.timeBased(), "Auto da Compadecida", "Ariano Suassuna"));
44  }
45
46  private void passo4() {
47   System.out.println("Mostrar Registros");
48   List<Livro> livros = cc.getAll();
```

```
49   for (Livro livro : livros) {
50    System.out.println(livro);
51   }
52  }
53
54  private void passoFatal() {
55   cc.eliminarTabela();
56   cc.eliminarKeyspace();
57  }
58 }
```

Esta classe será a nossa principal, agora podemos nos divertir a vontade com esse banco, tente explorar melhor, criar os métodos para alterar, excluir ou mesmo trazer um determinado livro - a CQL não é muito diferente em relação a SQL tradicional. Lembre-se que a Programação Orientada a Objetos é uma metodologia e não uma linguagem, se pratica essa forma ao usarmos os princípios da Orientação a Objetos e aproveitar a qualidade de extensibilidade do código.

3 Python

Python é uma linguagem de programação de alto nível, interpretada a partir de um script, Orientada a Objetos e de tipagem dinâmica. Foi lançada por Guido van Rossum em 1991. Não pretendo nesta apostila COMPARAR essa linguagem com Java (espero que nunca o faça), fica claro que os comandos são bem mais fáceis porém essas linguagens possuem diferentes propósitos.

Todos os comandos descritos abaixo foi utilizado no JupyterLab [5], então basta abrir um Notebook e digitá-los em cada célula conforme se apresentam.

3.1 Ações com o Banco de Dados

Baixar o pacote necessário:
`!pip install cassandra-driver`

Importar os pacotes necessários:
```
from cassandra.cluster import Cluster
from cassandra.query import SimpleStatement
```

Nos conectamos ao servidor desta forma:
```
cluster = Cluster()
session = cluster.connect()
```

A partir desse objeto *session* podemos dar qualquer comando para o Cassandra. Criar uma KeySpace: `session.execute("CREATE KEYSPACE usuarios WITH replication={'class': 'SimpleStrategy', 'replication_factor': 1}")`

Usar uma KeySpace:
`session.execute('USE usuarios;')`

Criar uma tabela:
```
session.execute('CREATE TABLE usuario (id int primary key, nome text,
creditos int)')
```

Adicionar uma linha:

```
session.execute('INSERT INTO usuario (nome, creditos, id) VALUES (%s, %s, %s)',
('Fernando Anselmo', 42, 1))
```

Nos conectamos a uma coleção desta forma:

```
rows = session.execute('SELECT id, nome, creditos FROM usuario')
for u in rows:
   print(u.id, u.nome, u.creditos)
```

3.2 Programa Completo

Mas antes de encerramos realmente vejamos o seguinte programa completo em linguagem Python:

```
1  from cassandra.cluster import Cluster
2  from cassandra.query import SimpleStatement
3  from random import randint
4
5  # Passo 1: Conectar ao Mongo
6  cluster = Cluster()
7  session = cluster.connect()
8
9  session.execute(
10  "CREATE KEYSPACE negocios WITH replication={'class': 'SimpleStrategy',
       'replication_factor': 1}")
11 session.execute('USE negocios;')
12
13 session.execute(
14  'CREATE TABLE restaurante (id int PRIMARY KEY, nome text, nota int, cozinha text)')
15
16 # Passo 2: Criar Amostras de Dados
17 nomes = ['Kitchen', 'Espiritual', 'Mongo', 'Tastey', 'Big', 'Jr', 'Filho', 'City',
       'Linux', 'Tubarão', 'Gado', 'Sagrado', 'Solo', 'Sumo', 'Lazy', 'Fun', 'Prazer',
       'Gula']
18 tipo_emp = ['LLC', 'Inc', 'Cia', 'Corp.']
19 tipo_coz = ['Pizza', 'Bar', 'Fast Food', 'Italiana', 'Mexicana', 'Americana', 'Sushi',
       'Vegetariana', 'Churrascaria']
20
21 for x in range(1, 501):
22   nome = nomes[randint(0, (len(nomes)-1))] + ' ' + nomes[randint(0, (len(nomes)-1))] + '
       ' + tipo_emp[randint(0, (len(tipo_emp)-1))]
23   result = session.execute('INSERT INTO restaurante (id, nome, nota, cozinha) VALUES
       (%s, %s, %s, %s)', (x, nome, randint(1, 5), tipo_coz[randint(0, (len(tipo_coz)-1))]))
24
25   # Passo 4: Mostrar no console o Object ID do Documento
26   print('Criado {0} de 500 como {1}'.format(x, result))
27
28   # Passo 5: Mostrar mensagem final
29   print('500 Novos Negócios Culinários foram criados...')
```

O programa está auto-documentado e criar uma base com 500 registros.

4 Conclusão

Penso que depois dessa apostila, será possível iniciar a descobrir o Apache Cassandra para seus trabalhos, pois como vimos é bem fácil realizar os passos nesse banco e pouco importa a linguagem de programação. Não busquei nesta mostrar um exemplo mais completo para não limitar suas pesquisas e devemos considerar esta apenas como um pontapé inicial (*KickStart*) para seus projetos.

Como visto este banco de dados pode ser facilmente utilizado com aplicações em linguagem Java ou gerar os modelos para *Machine Learning* com Python e ainda colher o benefício de substituir os bancos de dados relacionais para grandes quantidades de dados, sendo que esta é a grande motivação para NoSQL como forma de resolver o problema de escalabilidade dos bancos tradicionais.

As principais linguagens de programação possuem suporte e aqui vimos apenas Java e Python, porém existem muitas outras como PHP, C, C++, C#, JavaScript, Node.js, Ruby, R e Go. Esta apostila faz parte da série dos quatro tipos para Bancos de Dados no padrão NoSQL que estou tentando desmistificar e torná-los mais acessíveis tanto para as comunidades de Java e Python voltada especificamente para desenvolvedores ou cientistas de dados.

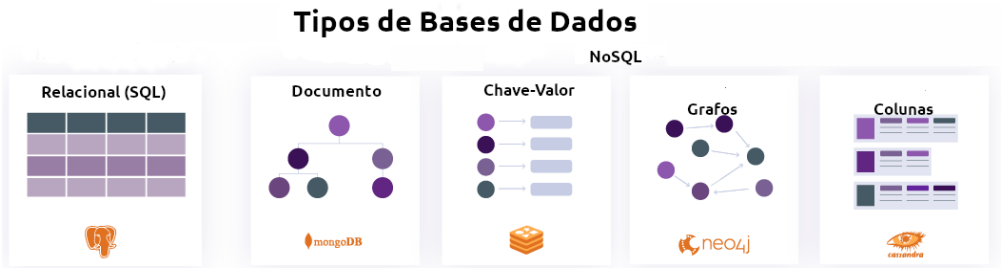

Figura 5: *Tipos de Bancos de Dados*

Sou um entusiasta do mundo **Open Source** e novas tecnologias. Qual a diferença entre Livre e Open Source? Livre significa que esta apostila é gratuita e pode ser compartilhada a vontade. Open Source além de livre todos os arquivos que permitem a geração desta (chamados de arquivos fontes) devem ser disponibilizados para que qualquer pessoa possa modificar ao seu prazer, gerar novas, complementar ou fazer o que quiser. Os fontes da apostila (que foi produzida com o LaTex) está disponibilizado no GitHub [8]. Veja ainda outros artigos que publico sobre tecnologia através do meu Blog Oficial [6].

Referências

[1] Página do Banco Apache Cassandra
`https://cassandra.apache.org`

[2] Página do Oracle Java
`http://www.oracle.com/technetwork/java`

[3] Página do Python
`https://www.python.org`

[4] Editor Spring Tool Suite para códigos Java
`https://spring.io/tools`

[5] Página do Jupyter
https://jupyter.org

[6] Fernando Anselmo - Blog Oficial de Tecnologia
http://www.fernandoanselmo.blogspot.com.br/

[7] Encontre essa e outras publicações em
https://cetrex.academia.edu/FernandoAnselmo

[8] Repositório para os fontes da apostila
https://github.com/fernandoans/publicacoes