CAPÍTULO 4

Ambiente de Programação

Este capítulo consiste na apresentação do ambiente de programação do ETL4NoSQL. Ele foi desenvolvido utilizando a linguagem de programação orientada a objetos Python. É demonstrado também os aspectos de implementação, as classes de software e as instanciações dos objetos das classes. As classes são interfaces de programação orientada a objetos fundamentais para o modelo de abstração de frameworks, e as classes deste trabalho serão utilizadas para a importação, mapeamento de BDs NoSQL e criação de processos de ETL a partir desses BDs.

4.1 Implementação

A implementação do ETL4NoSQL foi feita utilizando a linguagem de programação orientada a objetos Python. A escolha dessa linguagem justifica-se pelo fato dela utilizar o paradigma de orientação a objetos que é adequada para a implementação dos padrões de projeto desenvolvidos na proposta deste trabalho. Além disso, Python tem uma sintaxe de fácil aprendizado e pode ser usada em diversas áreas, como Web e computação gráfica. Ela é uma linguagem de alto nível interpretada, completamente orientada a objetos e também é um software livre.

Assim, a implementação do framework foi baseada nos princípios do design orientado a objetos de inversão de controle, onde determina que os módulos de alto nível não devem ser dependentes de módulos de baixo nível, e sim, de abstrações, ou seja, os detalhes devem depender das abstrações. Esse princípio sugere que dois módulos não devem ser ligados diretamente, pois devem estar desacoplados com uma camada de abstração entre eles. Para suprir esse princípio, o ETL4NoSQL tem uma classe abstrata para o Esquema de Dados, que utiliza dos mesmos comportamentos, para as diversas variações de esquemas que os paradigmas NoSQL possui, porém aplicados de acordo com a especificidade de cada um. Outro princípio importante utilizado é o da segregação de interfaces onde os usuários não devem ser forçados a depender de interfaces que não necessitam, deve-se escrever interfaces enxutas com métodos que sejam específicos da interface.

Portanto, o framework foi dividido em interfaces de importação, mapeamento dos BDs NoSQL, mecanismos e operações de processos de ETL. As ferramentas utilizadas para implementação do ETL4NoSQL foram:

- Notebook com sistema operacional MacOS X; processador de 2,5 GHz Intel Core i5; e memória 12 GB 1333 MHz DDR3;
- Python 2.7: Linguagem de programação orientada a objetos.
  Disponível em: https://www.python.org/download/releases/2.7/;
- LiClipse 3.4.0: plataforma de programação (IDE) open-source.
  Disponível em: http://www.liclipse.com/download.html;
- SGBD MariaDB versão 10.0.27.
  Disponível em: https://downloads.mariadb.org/mariadb/10.0.27/;
- SGBD Redis versão 3.2. Disponível em: https://redis.io/download;
- SGBD Cassandra 3.0. Disponível em: http://cassandra.apache.org/download/

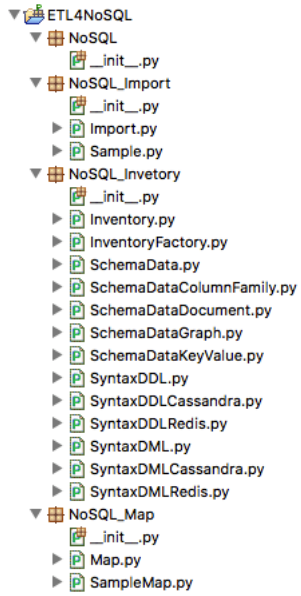
4.2 Interfaces de Programação

4.2.1 Módulo NoSQL

O módulo NoSQL é responsável por lidar com toda a parte que diz respeito ao dados modelados a partir dos paradigmas NoSQL. É neste módulo que serão feitas as importações dos

dados e mapeamentos dos esquemas das bases não relacionais. A árvore de organização das classes do módulo pode ser vista na Figura 4.1.

Figura 4.1 Árvore das Classes do Módulo NoSQL



4.2.1.1 Esquema de Dados - SchemaData

Esquema de dados é uma classe que utiliza o padrão factory method, esse padrão permite que as interfaces criem objetos, porém a responsabilidade de criação fica a cargo da subclasse, as classes que derivam dela são as variações de esquemas existentes dos vários paradigmas de armazenamento de dados presentes na literatura. O trecho do código está ilustrado na Figura 4.2.

O método createSchema é um método abstrato, e é por meio dele que as subclasses implementarão a criação dos seus esquemas de maneira personalizada.

4.2.1.2 Esquema de Dados Família de Coluna - SchemaDataColumnFamily

Classe do tipo ConcreteCreator derivada do Esquema de Dados onde define a criação do esquema das bases sob o paradigma NoSQL Família de coluna. Segundo Nasholm (2012), o esquema de dados do paradigma Família de Coluna é uma coleção de colunas em uma tabela. As linhas são similares às linhas do esquema relacional, exceto que todas as linhas na mesma tabela não necessariamente tem a mesma estrutura. Um valor numa célula, por exemplo, a intersecção de uma linha e uma coluna, é uma sequencia não interpretada de bit. Cada célula é versionada, significando que ela contém múltiplas versões do mesmo dado e que cada versão tem um timestamp atrelada a ela.O trecho do código está ilustrado na Figura 4.3.

Figura 4.2 Classe Schema Data

```
from abc import abstractmethod

class SchemaData:
    def __init__(self, name, describe):
        self.columns = []
        self.name = name
        self.describe = describe
        self.createSchema()

    @abstractmethod
    def createSchema(self):
        pass

    def getSchema(self):
        return "Name Schema: " + self.name + " Describe Schema: " + self.describe

    def putColumn(self, key):
        self.columns.append(key)

    def getColumn(self, key):
        index = self.columns.index(key)
        return self.columns[index]

    def getColumns(self):
        return self.columns

    def popColumn(self, key):
        self.columns.pop(self.columns.index(key))
```

Figura 4.3 Classe Schema Data Column Family

```
from NoSQL_Invetory.SchemaData import SchemaData


class SchemaDataColumnFamily(SchemaData):

    def createSchema(self):
        key = raw_input("Insert key value: ")
        timestamp = raw_input("Insert timestamp ")
        self.putColumn([key, timestamp])
```

4.2.1.3 Esquema de Dados Documento - SchemaDataDocument

Esquema de Dados Documento é uma subclasse do Esquema de Dados do tipo Concrete-Creator, ela define o esquema das bases sob o paradigma NoSQL Orientado a documento. Conforme Nasholm (2012), um documento é geralmente um conjunto de campos onde o campo é um par chave-valor. Chaves são strings atomicas ou sequencia de bits, e valores também são atomicos, por exemplo, inteiros ou strings, ou complexos, por exemplo, listas, mapas, entre outros. Um armazenamento de dados de documentos pode armazenar muitos documentos ou até mesmo muitas coleções de documentos. O trecho do código está ilustrado na Figura 4.4.

Figura 4.4 Classe Schema Data Document

```
from NoSQL_Invetory.SchemaData import SchemaData


class SchemaDataDocument(SchemaData):

    def createSchema(self):
        key = raw_input("Insert key value: ")
        document = raw_input("Insert document ")
        self.putColumn([key, document])
```

4.2.1.4 Esquema de Dados Grafo - SchemaDataGraph

Subclasse de SchemaData, define o esquema das bases sob o paradigma NoSQL Baseada em Grafos. De acordo com Nasholm (2012), bases de dados baseada em grafos é estruturada em grafos matemáticos. Um grafo G=(V, E) geralmente consiste em um conjunto de vertices V e um conjunto de arestas E. Uma aresta e $\in$ E é um parte de vértices (v1, v2) $\in$ V x V. Se o grafo é direto esses pares são ordenados. Os vértices do grafo são chamados de nós, e as arestas de relações. Cada nó contém um conjunto de propriedades. O trecho do código está ilustrado na Figura 4.5.

Figura 4.5 Classe Schema Data Graph

```
from NoSQL_Invetory.SchemaData import SchemaData


class SchemaDataGraph(SchemaData):

    def createSchema(self):
        node = raw_input("Insert node value: ")
        edge = raw_input("Insert edge value: ")
        vertice = raw_input("Insert vertice value:")
        self.putColumn([node, edge, vertice])
```

4.2.1.5 Esquema de Dados Chave Valor - SchemaDataKeyValue

Subclasse derivada do SchemaData onde define o esquema das bases sob o paradigma NoSQL Chave-Valor. Para Nasholm (2012), o modelo de dados Chave Valor é baseado na abstração de dados do tipo Map. Ele contém a coleção de pares chave-valor onde todas as chaves são únicas. O trecho do código está ilustrado na Figura 4.6.

Figura 4.6 Classe Schema Data Key Value

```
from NoSQL_Invetory.SchemaData import SchemaData


class SchemaDataKeyValue(SchemaData):

    def createSchema(self):
        key = raw_input("Insert key value: ")
        self.putColumn(key)
```

4.2.1.6 Sintaxe DDL - SyntaxDDL

Classe que utiliza o padrão factory method e define o comportamento da linguagem de definição de dados para cada tipo de SGBD e esquema de dados a serem criados, alterados ou excluídos. O trecho do código está ilustrado na Figura 4.7.

4.2.1.7 Sintaxe DDL Cassandra - SyntaxDDLCassandra

Subclasse de Sintaxe DDL onde define a linguagem de definição de dados do SGBD Cassandra para criação, alteração e exclusão de esquemas com dados oriundos do SGBD Cassan-

Figura 4.7 Classe Syntax DDL

```
from abc import abstractmethod

class SyntaxDDL:


    @abstractmethod
    def createSyntaxDDL(self):
        pass

    @abstractmethod
    def alterSyntaxDDL(self):
        pass

    @abstractmethod
    def dropSyntaxDDL(self):
        pass
```

dra.

4.2.1.8 Sintaxe DML - SyntaxDML

Classe que utiliza o padrão factory method e define o comportamento da linguagem de manipulação de dados para cada tipo de SGBD e esquema de dados a serem manipulados. O trecho do código está ilustrado na Figura 4.8.

Figura 4.8 Classe Syntax DML

```
from abc import abstractmethod

class SyntaxDML:

    @abstractmethod
    def selectSyntaxDML(self):
        pass

    @abstractmethod
    def updateSyntaxDML(self):
        pass

    @abstractmethod
    def insertSyntaxDML(self):
        pass

    @abstractmethod
    def deleteSyntaxDML(self):
        pass
```

4.2.1.9 Sintaxe DML Cassandra - SyntaxDMLCassandra

Subclasse de Sintaxe DML onde define a linguagem de manipulação de dados do SGBD Cassandra para a manipulação dos dados oriundos do SGBD Cassandra. Por meio dessa classe é possível buscar os dados da base de origem Cassandra.

4.2.1.10 Inventário - Inventory

Classe que define um inventário. Um inventário possui nome, descrição, dados de conexão e um conjunto de esquemas. O trecho do código está ilustrado na Figura 4.9.

Figura 4.9 Classe Syntax DML

```
class Inventory:

    def __init__(self, nameInventory, describeInventory, connection):
        self.schemas = []
        self.nameInventory = nameInventory
        self.describeInventory = describeInventory
        self.connection = connection

    def getInventoryName(self):
        return self.nameInventory

    def getInventoryDescribe(self):
        return self.describeInventory

    def getSchemas(self):
        return self.schemas

    def addSchema(self, schema):
        self.schemas.append(schema)
```

4.2.1.11 Fábrica de Inventário - InventoryFactory

Classe que cria inventários, ela é responsável por manter e remover inventários. O trecho do código está ilustrado na Figura 4.10.

Figura 4.10 Classe Syntax DML

```
from NoSQL_Invetory.Inventory import Inventory

class InventoryFactory:

    def __init__(self):
        self.inventories = []

    def addInventory(self, schemaData, nameInventory, describeInventory):
        self.inventories.append(Inventory(schemaData, nameInventory, describeInventory))

    def removeInventory(self, inventory):
        self.inventories.remove(inventory.getInventoryName())

    def getInventories(self):
        return self.inventories
```

4.2.1.12 Importação - Import

Classe responsável pela importação dos dados das bases NoSQL. Ela utiliza as classes de sintaxes, esquemas, acessa ao inventário e mantém os dados importados.

4.2.1.13 Amostra - Sample

Classe responsável por criar o esquema da amostra que será importada pela classe de importação. Ela acessa o inventário para utilizar o esquema de dados, sintaxes e dados de conexão para criar a amostra.