II Simpósio Regional de Agrimensura e Cartografia
"Ampliando os horizontes e discutindo o futuro da geoinformação e do cadastro territorial do Brasil"
Universidade Federal de Uberlândia – UFU / Campus Monte Carmelo
22 a 24 de novembro de 2021

ANÁLISE DE DIFERENTES IMPLEMENTAÇÕES DE UM PERFIL LADM: BANCO DE DADOS RELACIONAL, POO E BANCO DE DADOS NoSQL

NATHALIA ROSE SILVA DA PURIFICAÇÃO[1], ANDREA FLÁVIA TENÓRIO CARNEIRO[2], MÍLTON HIROKAZU SHIMABUKURO[3]

RESUMO

O Modelo de Domínio de Administração Territorial (*Land Administration Domain Model* – LADM) descrito na ISO 19.152:2012 criou uma linguagem semântica para a descrição de sistemas de administração territorial. Recentemente, este modelo entrou em revisão para aprimoramento da modelagem proposta originalmente, considerando perfis mais específicos e o uso de novas tecnologias. Modelos *NoSQL* e Programação Orientada a Objetos (POO) têm sido pouco utilizadas ou subutilizadas na administração territorial. No caso brasileiro, a implementação de perfis LADM ocorre, geralmente, por meio de banco de dados relacionais. O presente trabalho tem como objetivo analisar a implementação de um perfil LADM utilizando um banco de dados relacional (PostgreSQL), uma linguagem de programação orientada a objetos (Java) e um banco de dados NoSQL (MongoDB), verificando como são representadas as entidades nestas diferentes formas de implementação.

Palavras-chave: LADM. PostgreSQL. NoSQL. POO.

1 INTRODUÇÃO

O Modelo de Domínio de Administração Territorial (*Land Administration Domain Model* – LADM) descrito na ISO 19.152:2012 criou uma linguagem semântica para a descrição de sistemas de administração territorial, padronizando conceitos, terminologias e aplicações do cadastro territorial em todo o mundo.

Algumas pesquisas testaram a aplicabilidade do modelo LADM para o caso dos cadastros brasileiros, entretanto, a implementação destes modelos é geralmente realizada por meio de banco de dados relacionais (centralizados) que possuem uma estrutura rígida e bem definida.

A revisão do LADM (LADM II), descrita por Lemmen, Oosterom e Kalantari (2018), busca aprimorar a modelagem proposta originalmente, indicando etapas para as implementações do modelo, perfis mais específicos, modelos técnicos em várias codificações e uso de novas

[1] Universidade Estadual "Júlio de Mesquita Filho" Unesp, nathalia.purificacao@unesp.br
[2] Universidade Federal de Pernambuco, andrea.carneiro@ufpe.br
[3] Universidade Estadual "Júlio de Mesquita Filho" Unesp, milton.h.shimabukuro@unesp.br

tecnologias nos processos de administração de terras, como uso de dados não estruturados e bancos de dados não relacionais, que ainda são pouco utilizados no campo da administração territorial.

A estrutura do LADM é definida utilizando a *Unified Modeling Language* (UML), que é uma linguagem visual, ou gráfica, de modelagem padronizada e amplamente aceita na modelagem de softwares (GUEDES, 2011). A UML está totalmente inserida no contexto de orientação a objetos. Atualmente, várias ferramentas CASE (*Computer Aided Software Engineering*) permitem a construção de um modelo conceitual baseado na UML e já realizam a conversão do modelo em *scripts* SQL (*Structured Query Language*) de forma automática para criação de um banco de dados relacional da aplicação. Entretanto, para desenvolver aplicações em alguns bancos de dados não relacionais, como o MongoDB, por exemplo, é necessário escrever seu próprio código.

Deste modo, o presente trabalho tem como objetivo analisar a implementação de um perfil LADM utilizando um banco de dados relacional, uma linguagem de programação orientada a objetos e um banco de dados NoSQL – *Not Only SQL,* verificando como são representadas as entidades nestas diferentes formas de implementação.

2 METODOLOGIA

A metodologia adotada para o desenvolvimento da pesquisa considerou um fluxo de trabalho, dividido em três etapas principais. Inicialmente, foi elaborado um modelo conceitual para o cadastro de monumentos, criando um perfil LADM com base nas especificações da ISO 19.152:2012. O modelo conceitual foi criado utilizando o *plug-in* da ferramenta OMT-G, disponível na IDE (*Integrated Development Environment*) *Eclipse Modeling*. Posteriormente, deu-se início às etapas de implementações, que foram executadas por meio de um banco de dados relacional, utilizando o sistema gerenciador de banco de dados (SGBD) PostgreSQL; por programação orientada a objetos, implementada em linguagem Java, também na plataforma *Eclipse*; e a análise de implementação em um banco de dados *NoSQL,* utilizando o MongoDB (apenas de forma conceitual). Por fim, foi realizada a análise e comparação das diferentes implementações.

3 RESULTADOS E DISCUSSÃO

3.1 Modelo conceitual

O modelo conceitual proposto considerou algumas especificações definidas nas Normas de classificação para fins estatísticos dos monumentos históricos e artísticos do Brasil (IBGE, 1959) e também o banco de dados de Obras de Arte e Monumentos em Espaços Públicos da Cidade de São Paulo, disponibilizado pelo Departamento do Patrimônio Histórico da Secretaria Municipal de Cultura da Cidade de São Paulo.

No cadastro de monumentos foram consideradas como informações básicas as partes que se relacionam com os monumentos, se o monumento registrado é de propriedade pública ou privada, informações sobre documentos administrativos como resoluções de tombamento, sua localização e informações gerais sobre a obra, incluindo seu tipo (arquitetônico, escultórico, artístico-utilitário, geográfico ou topográfico); sua finalidade (civil, militar, religiosa, memorativa, decorativa ou utilitária); sua espécie (estátua, grupo escultórico, obelisco, figuras mitológicas, ruína, templo, fortaleza etc.); o nome do autor da obra, a nacionalidade do autor, data de inauguração e o material empregado na construção (bronze, mármore, dentre outros).

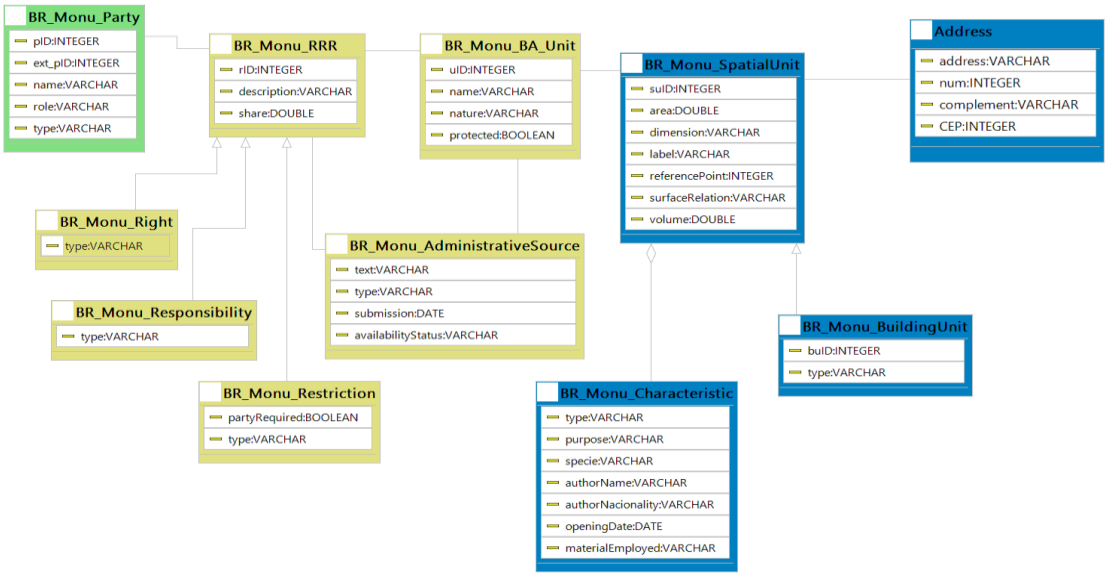
A estrutura de organização do LADM contempla o conceito fundamental do cadastro, descrevendo os relacionamentos entre pessoas e terras, que envolvem direitos, restrições e responsabilidades. Ela dispõe de três pacotes básicos (*Party*, *Administrative* e *Spatial Unit*), pelos quais pode-se representar as relações existentes entre estes elementos; e um subpacote (*Surveying and Representation*), referente à representação geométrica e topologia das unidades espaciais.

Cada pacote do LADM contém um conjunto de classes que define os atributos, métodos e como elas se relacionam entre si. Uma classe é representada graficamente por um retângulo e é vista como um agrupamento de objetos que possuem características comuns. Os objetos podem se comunicar entre si por meio dos relacionamentos que são definidos entre as classes, representados por uma linha que conecta uma classe com outra. Na Figura 1, apresenta-se o modelo proposto para o cadastro de monumentos. Nela podemos observar a presença de três tipos de relacionamentos característicos da UML: o relacionamento de associação (—), que ocorre quando há um vínculo entre as instâncias de duas classes distintas; a agregação ( ), que ocorre quando há um vínculo do tipo "Parte-todo" entre duas classes distintas; e o relacionamento de especialização/generalização (—▷), que ocorre quando há herança de atributos e métodos entre duas ou mais classes que possuem características muito semelhantes.

As cores das classes representadas no modelo conceitual (Figura 1) fazem alusão aos

pacotes do LADM. Em verde, o pacote das partes (*party*), que registrará as pessoas, organizações ou instituições que possuem algum tipo de direito, restrição ou responsabilidade sobre os monumentos. Em amarelo, o pacote administrativo (*administrative*), responsável pelo registro de direitos (propriedade pública ou privada), restrições (visitação, alteração construtiva) e responsabilidades (conservação, manutenção e restauração) que incidem sobre o monumento; e em azul, o pacote das unidades espaciais (*spatial units*), que guarda informações gerais sobre as características do monumento. Neste trabalho não foi modelado o subpacote dos levantamentos e representações.

Figura 1 – Modelo conceitual para o cadastro de monumentos.



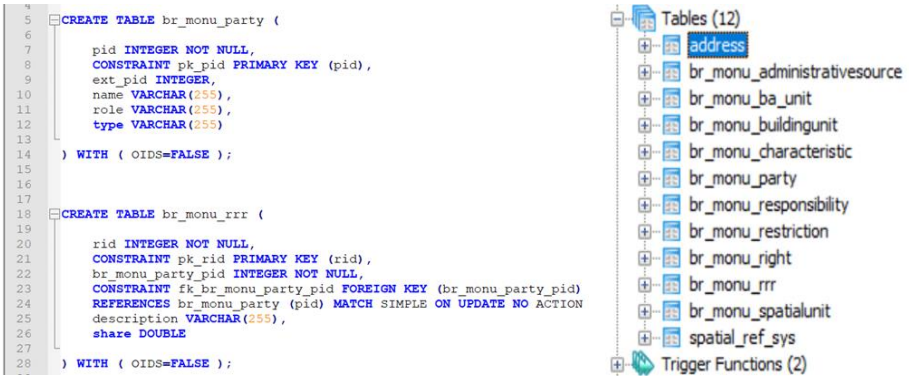
Elaboração: Os autores (2020).

4.2 Implementação do modelo em um banco de dados relacional

Os bancos de dados relacionais são organizados em tabelas, nas quais os atributos são representados em suas colunas e as instâncias nas linhas. Eles possuem uma forte padronização de conceitos e estrutura dos dados, onde cada informação tem um tipo de dado pré-definido (*integer, string, double, boolean* etc) e utiliza a linguagem padrão para consultas em bancos de dados relacionais, o SQL (LÓSCIO, OLIVEIRA E PONTES, 2011). Cada uma das classes definidas no modelo conceitual representa uma tabela no banco de dados. O *plug-in* da ferramenta OMT-G, disponível na plataforma Eclipse, permite a exportação do modelo desenvolvido para *scripts SQL* do *PostgreSQL* de forma automática. A Figura 2 (a) apresenta

um trecho do código SQL gerado para a implementação das tabelas no banco de dados.

Figura 2 – Implementação utilizando um banco de dados relacional.



```
CREATE TABLE br_monu_party (

    pid INTEGER NOT NULL,
    CONSTRAINT pk_pid PRIMARY KEY (pid),
    ext_pid INTEGER,
    name VARCHAR(255),
    role VARCHAR(255),
    type VARCHAR(255)

) WITH ( OIDS=FALSE );


CREATE TABLE br_monu_rrr (

    rid INTEGER NOT NULL,
    CONSTRAINT pk_rid PRIMARY KEY (rid),
    br_monu_party_pid INTEGER NOT NULL,
    CONSTRAINT fk_br_monu_party_pid FOREIGN KEY (br_monu_party_pid)
    REFERENCES br_monu_party (pid) MATCH SIMPLE ON UPDATE NO ACTION
    description VARCHAR(255),
    share DOUBLE

) WITH ( OIDS=FALSE );
```

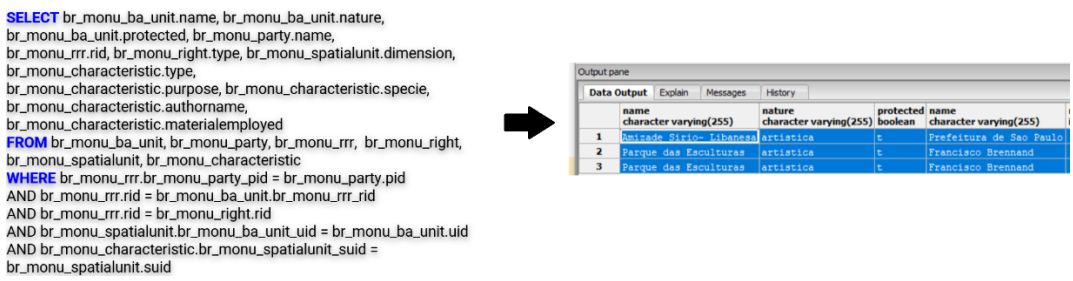

(a) (b)

Elaboração: Os autores (2020).

A partir do código apresentado na Figura 2(a), observa-se que o relacionamento entre as classes definidos no modelo conceitual são codificados automaticamente no *script* SQL através da definição de chaves estrangeiras (*Foreign Key*). Na Figura 2(b), verifica-se as tabelas implementadas no PostgreSQL: onze correspondentes às onze classes do modelo conceitual e uma (*spatial_ref_sys*) implementada automaticamente no banco de dados quando se ativa a extensão espacial PostGIS.

Para visualizarmos os dados inseridos no modelo implementado, utilizamos o comando SQL "SELECT FROM". Como as informações sobre os monumentos são distribuídas em várias tabelas, é necessário fazer uma série de referências na seleção, o que pode acabar complicando uma tarefa simples de consulta, caso o sistema possua um grande número de tabelas. A Figura 3 apresenta o resultado de uma consulta realizada através do comando citado.

Figura 3 – Resultado da Consulta através do comando "SELECT FROM".



```
SELECT br_monu_ba_unit.name, br_monu_ba_unit.nature,
br_monu_ba_unit.protected, br_monu_party.name,
br_monu_rrr.rid, br_monu_right.type, br_monu_spatialunit.dimension,
br_monu_characteristic.type,
br_monu_characteristic.purpose, br_monu_characteristic.specie,
br_monu_characteristic.authorname,
br_monu_characteristic.materialemployed
FROM br_monu_ba_unit, br_monu_party, br_monu_rrr,  br_monu_right,
br_monu_spatialunit, br_monu_characteristic
WHERE br_monu_rrr.br_monu_party_pid = br_monu_party.pid
AND br_monu_rrr.rid = br_monu_ba_unit.br_monu_rrr_rid
AND br_monu_rrr.rid = br_monu_right.rid
AND br_monu_spatialunit.br_monu_ba_unit_uid = br_monu_ba_unit.uid
AND br_monu_characteristic.br_monu_spatialunit_suid =
br_monu_spatialunit.suid
```

Elaboração: Os autores (2020).

4.3 Implementação do modelo utilizando programação orientada a objetos

Uma linguagem de programação orientada a objetos possui os mesmos conceitos existentes na UML (classe, objetos, relacionamentos) e tem três características principais: o encapsulamento, que permite a divisão das responsabilidades de um programa em várias unidades menores; a herança, utilizada para criar classes baseadas na definição de outras já existentes, nas quais as subclasses herdam todos os atributos, comportamentos e implementações das classes-mãe; e o polimorfismo, que permite que um mesmo identificador possua diferentes assinaturas, no qual um único nome pode assumir várias formas e representar comportamentos diferentes (SINTES, 2002).

Aqui, o objetivo foi verificar a forma que se dá a implementação dos diferentes tipos de classes e dos relacionamentos existentes no modelo conceitual para o cadastro de monumentos, que possui classes convencionais, uma classe abstrata e relacionamentos de herança, agregação e associação.

As classes convencionais são declaradas com uma visibilidade seguida da palavra-chave "*class*" e o nome da classe. Dentro da classe são escritos seus atributos, que também possuem visibilidade, e a definição do seu tipo, que pode ser um tipo primitivo de dados (*integer, double, boolean*...) ou de um tipo que segue a estrutura de outra classe implementada. O código a seguir apresenta a implementação da classe *BR_Monu_Party*.

```
public class MonuParty {
	private int pID; private String ext_pID; private String name; private String role;
	private String type; private MonuRRR rrr;
	}
```

Para cada um dos atributos descritos, implementaram-se os métodos construtores, *getter* e *setter*, que permitem inicializar um objeto durante sua criação, recuperar o valor de um objeto e alterar o seu estado interno, respectivamente. A seguir apresenta-se um trecho dos métodos criados para a classe *BR_Monu_Party*.

```
public String getExtID() {
	return this.ext_pID;}
public void setExtID(String extID){
	this.ext_pID=extID;}
public String getName() {
	return this.name;}
public void setName(String nm) {
	this.name=nm;
```

```
}
```

Uma classe abstrata é útil em casos de herança ou polimorfismo, como acontece na classe *BR_Monu_RRR*, onde se tem instâncias do tipo *Right*, *Restriction* e *Responsibility*. Os objetos são instanciados respectivamente nas subclasses *BR_Monu_Right*, *BR_Monu_Restriction* e *BR_Monu_Responsibility* que herdam os atributos e métodos de sua superclasse. Em Java, uma classe abstrata é definida utilizando a palavra-chave "*abstract*" logo após a visibilidade da classe, conforme apresentado no seguinte código:

```
public abstract class MonuRRR {
    protected int rID;
    protected String description;
    protected double share;
    protected MonuBaUnit monuBau;
    protected MonuAdministrativeSource admSource;

}
```

Já nos relacionamentos do tipo especialização/generalização, há uma extensão da superclasse através das suas subclasses, onde ocorre herança de atributos ou métodos. Este relacionamento fica definido nas subclasses, marcadas com a palavra-chave "*extends*", conforme apresentado na implementação da subclasse *BR_Monu_Restriction* a seguir.

```
public class MonuRestriction extends MonuRRR {
    private boolean partyRequired;
    private String type;
}
```

Os relacionamentos de associação e agregação são implementados com a mesma lógica de programação, apesar de possuírem semântica diferentes em UML. Quando instâncias de uma classe se associam ou se agregam com instâncias de uma outra classe, é necessário fazer uma referência ao objeto, definindo um atributo cujo tipo será o da classe pela qual a relação ocorre. Essa situação pode ser evidenciada na classe *BR_Monu_SpatialUnit* definida no modelo conceitual, a qual possui um relacionamento de associação com as classes *BR_Monu_BA_Unit* e com a classe *Address*; e um relacionamento de agregação com a classe *BR_Monu_Characteristic*, visualizado na Figura 1. Logo, foram definidos três atributos nesta classe que são instâncias dos tipos *baunit*, *address* e *characteristic*.

```
public class MonuSpatialUnit {
       protected int suID; protected double area;
       protected String dimension; protected String label;
       protected int referencePoint; protected String surfaceRelation;
       protected double volume; protected Address addr;
       protected MonuBaUnit baunit; protected MonuCharacteristic characteristic;
    }
```

Em Java, todo programa orientado a objeto possui uma classe principal, onde está definido o método *main* e onde todos os objetos podem ser instanciados (criados), através da utilização da palavra-chave "*new*". A inserção de valores em cada classe pode ser realizada através dos métodos construtores ou dos métodos *setter* definidos durante a implementação da classe. O código a seguir mostra um trecho do programa principal, no qual as classes *MonuParty* e *MonuRight* estão sendo instanciadas, sendo p1 um objeto do tipo parte e r1, um objeto do tipo direito.

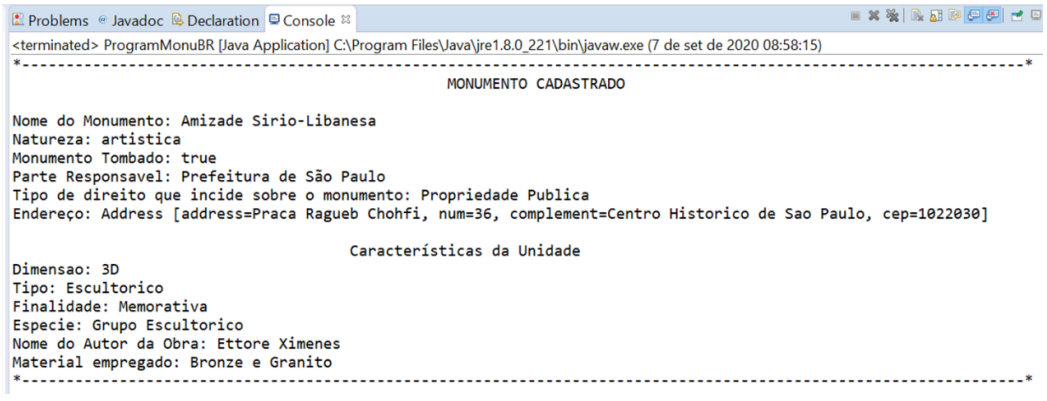
```
public class ProgramMonuBR {

	public static void main(String[] args) {
	MonuParty p1 = new MonuParty("46.395.000/0001-39.", "Prefeitura de São Paulo",
	"estadoAdministrador", "Pessoa Juridica");
	MonuRight r1= new MonuRight("Propriedade Publica");
	r1.setDescription("Direito de Propriedade");
	r1.setShare(1.0);
	r1.setrID();

   }
```

O objeto p1 foi construído passando as referências dos atributos código externo, que neste caso refere-se ao CNPJ da parte, nome da parte (Prefeitura de São Paulo), papel da parte (Estado Administrador) e seu tipo (Pessoa Jurídica). O objeto r1 foi construído passando apenas o valor para o tipo de direito (Propriedade Pública). A identificação sequencial de cada um dos objetos foi atribuída chamando o método *setID*. A Figura 4 apresenta o resultado para um monumento instanciado no programa.

Figura 4 – Impressão dos dados do monumento instanciado.



Elaboração: Os autores (2020).

4.4 Análise da implementação utilizando um banco de dados NoSQL

A análise de como seria a implementação do modelo para o cadastro de monumentos por meio de um banco NoSQL foi realizada tomando como base o banco de dados MongoDB, que possui modelo de dados orientado a documentos e armazena uma coleção de documentos. Cada documento é considerado como um objeto que possui um identificador unívoco e uma série de campos (atributos).

O modelo orientado a documentos não depende de uma estrutura bem definida como ocorre nos bancos de dados relacionais. Logo, como seu escopo é flexível e não possui necessariamente um esquema, não é exigido que se defina uma estrutura prévia para ser utilizada na aplicação. Deste modo, ao criarmos um banco de dados no MongoDB para o cadastro de monumentos, poderíamos apenas inserir uma coleção, para os monumentos, utilizando documentos JSON, com um conjunto de campos que equivaleria aos atributos modelados nas tabelas definidas na parte conceitual, o que poderia ser feito utilizando o seguinte comando:

```
db.monumentos.insert({name: "Amizade Sirio-Libanesa", nature: "Artistica", protected: "sim", partyName: "Prefeitura de São Paulo", rightType: "Propriedade Publica", address: "Praca Ragueb Chohfim, nº36, Centro Historico de São Paulo, cep: 1022030", dimension: "3D", type: "Escultorico", purpose: "Memorativa", specie: "Grupo escultórico", authorName: "Ettore Ximenes", materialEmployed: "Bronze e Granito"})
```

Observa-se que não é necessário definir os tipos dos campos, apenas informar o nome do campo (nome do monumento, natureza, se o monumento é tombado etc.) e seu respectivo valor. Uma característica importante presente no paradigma NoSQL é que caso seja necessário inserir outros monumentos na coleção db.monumentos, estes não precisam ter os mesmos campos do primeiro monumento inserido. Cada um dos monumentos inseridos na coleção pode possuir atributos diferentes.

5 CONSIDERAÇÕES FINAIS

Foi verificado que a implementação utilizando um banco de dados relacional é facilitada pela existência de ferramentas cases que convertem o modelo conceitual em scripts SQL de forma automática. No entanto, o uso de um banco de dados relacional esbarra em um dos principais objetivos do LADM: fornecer uma base de dados que possa ser compartilhada por diferentes instituições. Ao precisar de acessos simultâneos ao sistema, um banco de dados relacional apresenta uma redução de seu desempenho, já que esta ação provocaria uma alta concorrência no sistema, aumentando assim o tempo de resposta das consultas.

Além disso, devido ao fato de os dados serem organizados de forma fortemente rígida com várias tabelas, os processos de consulta exigem que várias referências sejam inseridas no comando de seleção para se obter a informação completa sobre uma instância, cujos dados estão em tabelas distintas.

Com relação à implementação utilizando programação orientada a objetos, foi verificado que classes convencionais, abstratas e relacionamentos onde ocorrem herança de características e métodos são facilmente implementadas em Java a partir do das palavras-chave definidas na linguagem. A implementação em Java ainda não está conectada a nenhum banco de dados. Logo, para que um grande número de monumentos fosse cadastrado, seria necessário conectar o programa desenvolvido a um banco de dados e implementar uma interface de inserção dos dados no sistema, de modo a facilitar e otimizar a interação com um usuário. Neste caso, existe uma correspondência entre classes, no código, e tabelas, no banco de dados.

Apesar do modelo não ter sido de fato implementado em um banco de dados NoSQL, percebe-se, a partir da análise realizada, que a inserção de informações neste tipo de banco de dados é simplificada e mais flexível, devido ao modelo ser livre de esquema. Essa característica provoca um aumento de seu desempenho, diminuindo drasticamente o tempo de retorno das consultas.

Para trabalhos futuros, recomenda-se que o modelo seja implementado em um banco de dados NoSQL, a conexão do programa implementado em Java a um banco de dados e a realização de testes para verificar as vantagens e desvantagens do uso destas diferentes ferramentas na administração territorial.

REFERÊNCIAS

GUEDES, G. **UML 2: Uma Abordagem Prática**. 2nd ed. São Paulo: Novatec, p.19-29; 101-117, 2011.

IBGE. **Monumentos históricos e artísticos do Brasil : normas de classificação para fins estatísticos**. Rio de Janeiro: Seção de Estatísticas Culturais, 1959. Disponível em: https://biblioteca.ibge.gov.br/biblioteca-catalogo.html?id=281652&view=detalhes. Acesso em: 08 set. 2021.

ISO/TC211. ISO 19152 - **Land Administration Domain Model (LADM)**. ISO/TC211, 2012.

LEMMEN, C., van OOSTEROM, P. AND KALANTARI, M. Towards a New Working Item Proposal for Edition II of LADM**. In:** 7th International Fig Workshop On The Land Administration Domain Model, Zagreb, Croatia, 2018.

LÓSCIO, B. F.; OLIVEIRA, H. R.; PONTES, J. C. S. NoSQL no desenvolvimento de aplicações Web colaborativas. **In***:* VIII Simpósio Brasileiro de Sistemas Colaborativos, v. 10, n. 1, p. 11, 2011.

SINTES, A. **Aprenda Programação Orientada a Objetos em 21 dias**. Tradução: Tortello, J. E. N. Makron Books. São Paulo: Pearson Education do Brasil, 2002.