Viviana Elizabeth Romero Noguera

Extensão de uma álgebra ER para execução de consultas em bancos de dados NoSQL orientados a documentos

Brasil

2018, Abril

Viviana Elizabeth Romero Noguera

Extensão de uma álgebra ER para execução de consultas em bancos de dados NoSQL orientados a documentos

Exame de Dissertação apresentado ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.

Universidade Federal de São Carlos – UFSCar

Departamento de Computação

Programa de Pós-Graduação

Orientador: Daniel Lucrédio

Brasil

2018, Abril

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado da candidata Viviana Elizabeth Romero Noguera, realizada em 02/04/2018:

Prof. Dr. Daniel Lucrédio
UFSCar

Prof. Dr. Valter Vieira de Camargo
UFSCar

Prof. Dr. Ronaldo dos Santos Mello
UFSC

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Ronaldo dos Santos Mello e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

Prof. Dr. Daniel Lucrédio

A meus pais Eulalia e Ángel.

Agradecimentos

A Deus por me abençoar com a vida, força, fé e saúde para que eu pudesse concluir mais esta importante etapa da minha vida.

Ao meu orientador, professor Dr. Daniel Lucrédio, pela orientação, pelos conhecimentos compartilhados, por ter compreendido minhas dificuldades e ajudado a realizar este trabalho.

Aos professores: Dr. Valter Vieira de Camargo e Dr. Ronaldo dos Santos Mello por contribuírem de forma significativa com sugestões de melhoria e por aceitarem participar da defesa realizada.

A meus pais, Eulalia e Ángel pelo apoio incondicional. Também por todo o sacrifício, em todos os aspectos, que fizeram para que eu pudesse alcançar meus objetivos.

A minha irmã, Sofia, por acreditar que este grande sonho um dia se realizaria.

A meu namorado, por demonstrar tamanho companheirismo em todos os momentos difíceis. Pelo apoio, força e compreensão.

Ao Departamento de Computação da UFSCar, por me oferecer uma oportunidade para obter o título de mestre.

A Organização dos Estados Americanos - OEA e o Grupo Coimbra de Universidades Brasileiras, ao CNPq, pelo auxílio financeiro que tornou possível o desenvolvimento deste trabalho.

Aos professores do Programa de Pós Graduação em Ciência da Computação da UFSCar, que de alguma forma contribuíram para minha formação.

Aos meu amigos e colegas de mestrado os quais compartilhei alegrias e dificuldades deste percurso.

A meus amigos do Paraguai, pelo apoio e conselhos.

Enfim, agradeço a todos que contribuíram de forma direta ou indiretamente para a obtenção do título de mestre em Ciência da Computação.

Resumo

Bancos de dados não relacionais, também chamados de NoSQL ("Not Only SQL"), surgiram da necessidade de contar com novos mecanismos de armazenamento, recuperação e processamento de grandes volumes de dados produzidos principalmente a partir da chegada das aplicações Web 2.0. Existem diferentes modelos de dados não relacionais, cada um com suas próprias características e aplicabilidade. Existem também centenas de sistemas gerenciadores de bancos de dados não relacionais, cada um com uma linguagem de consulta diferente e sua própria API. Devido à falta de um padrão para as linguagens de manipulação de dados (DML) nesse tipo de bancos de dados, torna-se um desafio para o Engenheiro de Software lidar com o aprendizado e o desenvolvimento das consultas. Além disso, por priorizar o desempenho, bancos de dados não relacionais frequentemente adotam esquemas desnormalizados, gerando uma grande diversidade nas estruturas dos dados armazenados, o que também dificulta o trabalho de desenvolvimento e reutilização das consultas. Frente a esses dois desafios, este trabalho teve como objetivo a definição de uma abordagem genérica para consultas, independente do tipo do banco de dados e também independente do esquema no qual os dados são armazenados. Para isso, estendeu-se uma álgebra existente, baseada em Entidades e Relacionamentos para manipulação de dados, com uma semântica operacional dirigida por modelos (*Model-Driven Engineering* - MDE). Mais especificamente, foram desenvolvidos metamodelos para que o Engenheiro de Software possa: (i) modelar as entidades persistentes (Entidade-Relacionamento); (ii) modelar consultas sobre as entidades e relacionamentos conforme a álgebra sendo utilizada; e (iii) modelar o mapeamento entre as entidades e os tipos de documentos armazenados. Com base nessas especificações, foi implementado um algoritmo que gera automaticamente consultas JavaScript para serem executadas no banco de dados MongoDB, de acordo com o modelo de mapeamento. Caso o esquema do banco precise ser modificado ao longo do ciclo de vida, apenas o modelo do mapeamento (modelo "iii") precisa ser modificado. O modelo conceitual e as consultas (modelos "i" e "ii") permanecem completamente independentes do esquema de armazenamento, podendo ser desenvolvidos e reutilizados mais facilmente. Foram realizadas duas avaliações, utilizando um exemplo de um sistema acadêmico e um sistema real, buscando testar exaustivamente as diferentes possibilidades de mapeamento entre entidades e documentos. Os resultados obtidos com esses sistemas demonstram ser possível que as consultas geradas retornem sempre os mesmos resultados, independente do mapeamento, alcançando assim o objetivo deste trabalho.

Palavras-chaves: NoSQL, Bancos de dados, Consultas, Entidade Relacionamento, Mapeamento, MDE.

Abstract

Non-relational databases, also called as NoSQL ("Not Only SQL"), have emerged from the need for new mechanisms for storing, retrieving and processing for large amounts of data produced mainly from the coming of Web 2.0 applications. There are different models of non-relational data, each with its own characteristics and applicability. There are also hundreds of non-relational database management systems, each with a different query language,and its own API. Due to the lack of a standard for data manipulation languages (DML) in this type of database, it is a challenge for the Software Engineer to deal with the learning and development of queries.In addition, by prioritizing performance, non-relational databases often adopt denormalized schemas, generating a large diversity in stored data structures, which also makes it difficult to develop and reuse queries. Faced with these two challenges, this work aimed to define a generic approach to queries, independent of the database type and also independent of the schema in which the data is stored. For this, extended an existent algebra, based on Entities and Relationships for data manipulation, with an operational semantics driven by models (*Model-Driven Engineering* - MDE). More specifically, metamodels have been developed so that the Software Engineer can: (i) model the persistent entities (Entity-Relationship); (ii) model queries about the entities and relationships according to the algebra being used; and (iii) model the mapping between the entities and the types of documents stored. Based on these specifications, an algorithm was implemented that automatically generates JavaScript queries to be executed in the MongoDB database, according to the mapping model. If the database schema needs to be modified throughout the lifecycle, only the mapping model (model "iii") needs to be modified. The conceptual model and the queries ("i" and "ii" models) remain completely independent of the storage schema and can be developed and reused more easily. Two evaluations were carried out, using an example of an academic system and a real system, trying to exhaustively test the different possibilities of mapping between entities and documents. The results obtained with these systems show that it is possible that the generated queries always return the same results, independent of the mapping, thus reaching the objective of this work.

Key-words: NoSQL, Database, Queries, Entity Relationship, Mapping, MDE.

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas e Siglas

ACID	*Atomicity, Consistency, Isolation, Duration / Atomicidade, Consistência, Isolamento, Durabilidade*
API	*Application Programming Interface / Interface de programação de aplicativos APT- Automatically Programmed Tool / Ferramenta automaticamente programada*
ASCII	*American Standard Code for Information Interchange / Código Padrão Americano para o Intercâmbio de Informação*
BD	*Banco de dados*
BASE	*Basically Available, Soft state, Eventual Consistency / Basicamente Disponível, Estado Leve, Consistência eventual*
BLOB	*Binary Large OBject / Objeto Binário Grande*
BNF	*Backus- Naur Form / Formalismo Backus-Naur*
BQL	*Bridge Query Language*
CAP	*Consistency, Availability, Tolerance of network partition / Consistencia, Disponibilidade, Tolerância a partição da rede*
CIM	*Computation Independent Model / Modelo Independente de Computação*
CRUD	*Create, Read, Update e Delete / Criar, Ler, Atualizar, Remover*
DDL	*Data Definition Language / Linguagem de Definição de Dados*
DML	*Data Manipulation Language / Linguagem de Manipulação de Dados*
DSL	*Domain Specific Language / Linguagem de domínio específico*
DSML	*Domain Specific Modeling Language / Linguagem de modelagem de domínio específico*
EBNF	*Extended Backus–Naur form / Formalismo de Backus-Naur Estendido*
EMF	*Eclipse Modeling Framework*
EOL	*Epsilon Object Language*

ER	*Entidade Relacionamento*
GLC	*Gramática Livre de Contexto*
HQL	*Hibernate Query Language / Linguagem de consulta de Hibernate*
HTTP	*Hypertext Transfer Protocol / Protocolo de Transferência de Hipertexto*
JET	*Java Emitter Templates*
JMI	*Java Metadata Inferface*
JSON	*JavaScript Object Notation / Notação de Objetos JavaScript*
M2T	*Modelo to text / Modelo para texto*
MDA	*Model Driven Architecture / Arquitetura Dirigida a Modelos*
MDE	*Model Driven Engineering / Engenharia Dirigida a Modelos*
MDSD	*Model Driven Software Development / Desenvolvimento de Software Orientado a Modelos*
MOF	*Model Driven Engineering / Meta Object Facility*
NoSQL	*Not Only Structured Query Language / Não apenas Linguagem de Consulta Estructurada*
OCL	*Object Constraint Language / Linguagem de Restrição de Objeto*
OMG	*Object Management Group*
PDF	*Portable Document Format / Formato Portátil de Documento*
PIM	*Platform Independent Model / Modelo Independente de Plataforma*
PSM	*Platform Specific Model/ Modelo Específico de Plataforma*
QODM	*Query Oriented Data Modeling / Modelagem de Dados Orientada a Consulta*
REST	*Representational State Transfer / Transferência de Estado Representacional*
RS	*Revisão Sistemática*
SGBD	*Sistema de Gerenciamento de Banco de Dados*
SGBDR	*Sistema Gerenciador de Banco de Dados Relacional*

SOS	*Save Our Systems*
SQL	*Structured Query Language / Linguagem de Consulta Estruturada*
UML	*Unified Modeling Language / Linguagem de Modelagem Unificada*
XML	*eXtensible Markup Language / Linguagem de marcação extensible*

Sumário

1 Introdução

Bancos de dados relacionais têm sido as principais tecnologias de armazenamento de dados durante anos. No entanto, alterações nas exigências para o processamento de dados causaram o surgimento de novos mecanismos de armazenamento de dados, recuperação e processamento (KARNITIS; ARNICANS, 2015).

O advento das aplicações Web 2.0 provocou o aumento da quantidade de conteúdo gerado pelo usuário, sendo essencial que as aplicações que gerenciam esses dados tolerem falhas, disponibilidade e escalabilidade. Com o objetivo de atender essas necessidades surgiram os sistemas de gerenciamento de bancos de dados (SGBDs) NoSQL (SCAVUZZO; NITTO; CERI, 2014).

SGBDs NoSQL são bancos de dados não-relacionais (LI; MA; CHEN, 2014). Os SGBDs NoSQL são projetados para atender aos requisitos de desempenho e escalabilidade que não podem ser abordados por bases de dados relacionais tradicionais, porque está ficando caro ou impossível obter e processar grandes volumes de dados rapidamente, em virtude do aumento de dados que exigem mais recursos computacionais (SADALAGE; FOWLER, 2013). Por esse motivo, são principalmente utilizados em aplicativos baseados na Web (CHANDRA, 2015), apesar de não serem dedicados exclusivamente a esse domínio.

Os bancos de dados NoSQL atuam sem um esquema, permitindo que sejam adicionados, livremente, campos aos registros do banco de dados, sem ter de definir primeiro quaisquer mudanças na estrutura. Além disso, os bancos de dados NoSQL não utilizam o modelo relacional, tem uma boa execução em *cluster* que permite replicação e distribuição de dados. Outra característica é que a maioria dos SGBDs NoSQL utilizados atualmente tem código aberto (SADALAGE; FOWLER, 2013), (LI; MA; CHEN, 2014).

Modelos de dados de bancos de dados NoSQL não têm estrita restrição na estrutura de dados e são usados para armazenar dados semi-estruturados (LI; MA; CHEN, 2014). De acordo com suas características, as bases de dados NoSQL usualmente são divididas em quatro modelos de dados principais (SCHMID; GALICZ; REINHARDT, 2015):

- Chave-valor: este tipo de banco de dados NoSQL usa um esquema simples baseado em pares chave-valor;
- Colunar: os dados são organizados em colunas ao invés de linhas;
- Documentos: os dados não são armazenados em tabelas, mas em documentos; e

- Grafos: os dados são armazenados como um grafo ou uma árvore que liga os diferentes aspectos dos dados.

Dentre esses modelos, os bancos de dados orientados a documentos lidam com coleções de objetos representados em formatos como XML, YAML, BSON ou JSON. Cada documento é composto por um conjunto de campos (aninhado) e está associado a um identificador exclusivo, para fins de indexação e recuperação (ATZENI; BUGIOTTI; ROSSI, 2012). Os documentos não precisam respeitar necessariamente um esquema. Normalmente, todos os documentos de uma coleção têm um propósito relacionado (STANESCU; BREZOVAN; BURDESCU, 2016). MongoDB é o banco de dados de documentos de código-aberto mais popular do mundo NoSQL (RANKING, 2018). Está concebido para ser capaz de enfrentar novos desafios, tais como escalabilidade, alta disponibilidade e flexibilidade para lidar com dados (KAUR; RANI, 2013). Por estes motivos, este modelo de dados NoSQL foi escolhido para fins deste trabalho.

1.1 Motivação e Caracterização do Problema

Hoje existem mais de 100 implementações de SGBDs NoSQL diferentes. A maioria se enquadra em um dos quatro modelos de dados, sendo que existe muita heterogeneidade entre os modelos (DHARMASIRI; GOONETILLAKE, 2013). Devido a essa razão, trabalhar com vários sistemas NoSQL torna-se difícil (STONEBRAKER, 2010), induzindo vários problemas ao desenvolvimento, implementação e migração de aplicativos de armazenamento de dados múltiplo. Com efeito, como NoSQL envolve tecnologias relativamente recentes, a falta de padrão é uma grande preocupação para organizações interessadas em adotar qualquer um destes sistemas (STONEBRAKER, 2011). Aplicações e dados não são portáveis e habilidades e conhecimentos adquiridos em um sistema específico não são reutilizáveis em outro. Dessa maneira os desenvolvedores devem estar familiarizados com diferentes APIs quando estão codificando suas aplicações (ATZENI; BUGIOTTI; ROSSI, 2012), (SELLAMI; BHIRI; DEFUDE, 2016).

Além da falta de padrão, a falta de um esquema predefinido pode também dificultar o desenvolvimento. Enquanto no modelo relacional existe um conjunto de diretrizes de mapeamento (ELMASRI et al., 2010) que garantem certa homogeneidade na recuperação dos dados, nos modelos não relacionais não há nenhuma garantia. Esse aspecto desnormalizado de bancos de dados NoSQL faz que o desempenho de consulta seja altamente dependente de caminhos de acesso (CURÉ et al., 2011). Por esse motivo existem diferentes formas de mapeamento do modelo conceitual para o modelo físico NoSQL, cada uma buscando maximizar o desempenho das consultas.

Para ilustrar melhor o problema que se buscou resolver, na Figura 1 é mostrado um exemplo simples de mapeamento entre um modelo ER e documentos armazenados no

MongoDB, para o domínio de oficinas de veículos. Neste exemplo, a entidade "Pessoa" foi mapeada para um tipo de documento específico ("DocTypePessoa"), onde seus dados são armazenados como elementos de um documento do tipo JSON. O campo "_id" é obrigatório para documentos no MongoDB. A entidade "Carro" também foi mapeada para um tipo de documento específico ("DocTypeCarro"). Já o relacionamento "Dirige" aparece como um campo multivalorado (denominado "carros") dentro do tipo de documento para armazenar pessoas. Neste campo são armazenados, em uma lista, os identificadores de todos os carros que aquela pessoa dirige, e o atributo do relacionamento. Os identificadores da entidade "Carro" dentro do *array* "carros" funcionam como uma referência a outros documentos.

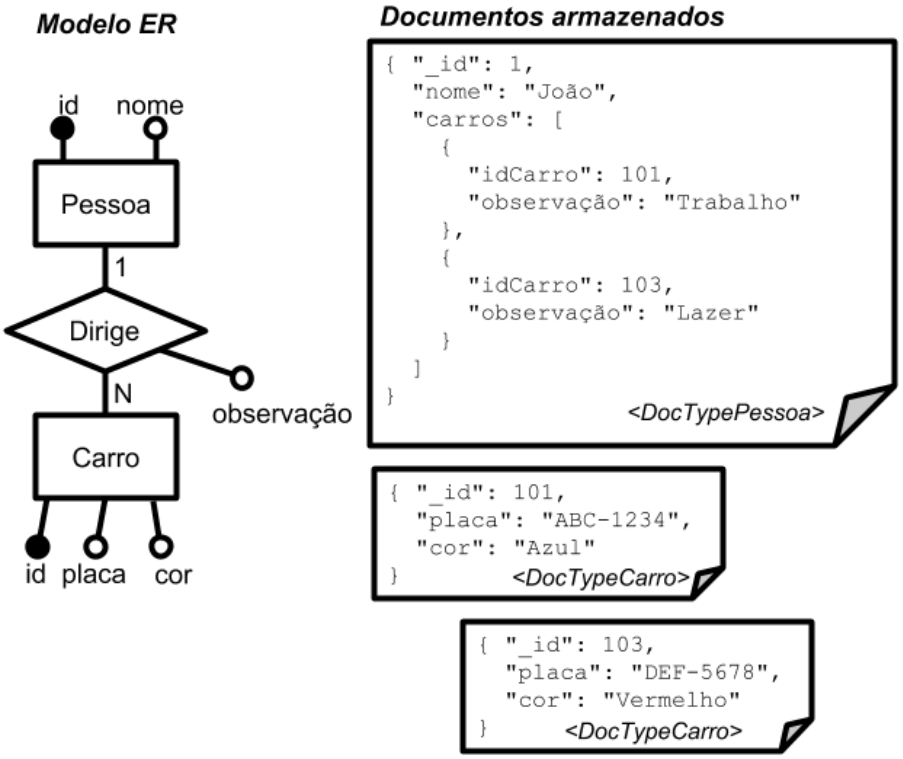

Figura 1 – Exemplo de mapeamento ER para MongoDB com referências a documentos.

Considere a seguinte consulta Q1: *"Encontrar todos os dados das pessoas junto com os dados dos respectivos carros que elas dirigem"*.

No MongoDB, para esse mapeamento em particular, é necessário percorrer diferentes tipos de documentos, já que os dados sendo requisitados estão separados. Há pelo menos duas possibilidades de se implementar essa consulta. A Listagem 1.1 mostra uma implementação utilizando cursor (método *forEach*). Nesta implementação primeiro é necessário acessar o documento *DocTypePessoa*, percorrer com o método *forEach*, recuperar os atributos da entidade 1 "Pessoa" e inserir no documento "DocTypeFinal" (linhas 1-5), em seguida com o método "*forEach*"(linha 6), percorrer o campo "carros" de tipo "*array*" (que contêm o atributo do relacionamento "Dirige" e o identificador da entidade

2 “Carro”). Posteriormente, é executada a junção entre o documento *DocTypeCarro* e o identificador do array “carros” pelos atributos “*_id*” e “dados1._id” (linha 7). Depois os dados são recuperados e atualizados pelo identificador da entidade 1 no documento “DocTypeFinal” (linhas 8-19). O documento “DocTypeFinal” contêm os dados requisitados na consulta Q1.

```
1  db.DocTypePessoa.find().forEach(function(dados) {
2    db.DocTypeFinal.insert({
3      _id: dados._id,
4      nome: dados.nome
5    });
6    dados.carros.forEach(function(dados1){
7     db.DocTypeCarro.find({'_id': dados1._id}).forEach(function(
     dados2){
8       db.DocTypeFinal.update({'_id':dados._id},
9         { $addToSet: {
10          'carros': {
11             _id: dados2._id,
12             placa: dados2.placa,
13             cor: dados2.cor
14             observacão: dados1.observacão
15          }
16        }});
17      });
18   });
19 });
```

Listagem 1.1 – Consulta Q1 implementada para o mapeamento da Figura 1 utilizando cursor.

A Listagem 1.2 apresenta um exemplo de resultado da consulta Q1, uma vez executada a consulta mostrada na Listagem 1.1.

```
1  {
2    "_id": 1,
3    "nome": "João",
4    "carros":
5    [
6      {
7          "_id": 101,
8          "placa": "ABC-1234",
9          "cor": "Azul",
10         "observacão": "Trabalho"
11     },
```

```
12      {
13          "_id": 103,
14          "placa": "DEF-5678",
15          "cor": "Vermelho",
16          "observacão": "Lazer"
17      }
18    ]
19 }
```

Listagem 1.2 – Exemplo de resultado da consulta Q1 implementada para o mapeamento da Figura 1 utilizando cursor.

A Listagem 1.3 mostra uma implementação da mesma consulta Q1 mas utilizando agregação (método *aggregate*). O MongoDB fornece o método *aggregate*() no *shell* mongo e o comando *aggregate* para *pipeline* de agregação. O *pipeline* de agregação é um framework para agregação de dados modelado no conceito de *pipelines* de processamento de dados. Os documentos inserem um *pipeline* em várias etapas que transforma os documentos em resultados agregados.

Na implementação da Listagem1.3 é necessário executar o método *aggregate* sobre o documento *DocTypePessoa* (linha 1). O operador $unwind (linha 3) desconstrói um campo de matriz dos documentos de entrada para exibir um documento para cada elemento, neste exemplo, a matriz "carros" presente dentro do documento *DocTypePessoa*.

O operador $lookup (linhas 4 e 11) executa *left outer join* para uma coleção no mesmo banco de dados. Para cada documento de entrada (neste exemplo o documento *DocTypePessoa*), a etapa $lookup adiciona um novo campo de matriz cujos elementos são os documentos correspondentes a coleção especificada na sintaxe *from* (linha 6), (neste exemplo *DocTypeCarro*). A etapa $lookup passa esses documentos para a próxima etapa. Posteriormente, é executado novamente o operador $unwind, mas desta vez sobre a matriz "dados_completo_carros" (linha 12). A continuação o operador $group (linha 13-22) agrupa os documentos pelos campos *_id* (identificador do documento *DocTypePessoa*), nome e "dados". Os documentos de saída contêm um campo id (linha 13) que conta com uma chave distinta por grupo. O operador $push (linha 18) adiciona um valor específico a uma matriz. Neste exemplo dentro do campo "dados" são adicionados os atributos correspondentes a entidade "Carro" e o relacionamento "Dirige". O operador $out obtém os documentos retornados pelo *pipeline* de agregação e os grava na coleção "DocTypeFinal" (linha 23).

```
1 db.DocTypePessoa.aggregate(
2    [
3     {$unwind:``$carros''},
4     {$lookup:
```

```
5         {
6           from: ‘‘DocTypeCarro’’,
7           localField: ‘‘carros.idCarro’’,
8           foreignField: ‘‘_id’’,
9           as: ‘‘dados_completo_carros’’
10        }
11    },
12    {$unwind:‘‘$dados_completo_carros’’},
13    {$group: {_id:
14        {
15                ‘‘_id": ‘‘$_id’’,
16                nome: ‘‘$nome’’},
17                dados: {
18                    $push:  {
19                    Carro: ‘‘$dados_completo_carros’’,
20                    Dirige: ‘‘$carros.observacão’’
21                } }
22    }},
23    { $out: ‘‘DocTypeFinal’’ }
24   ]
25 )
```

Listagem 1.3 – Consulta Q1 implementada para o mapeamento da Figura 1 utilizando o método *aggregate*

A Listagem 1.4 apresenta um exemplo de resultado da consulta Q1 utilizando o método *aggregate*. O primeiro campo "*_id*" corresponde a chave distinta gerada pelo operador $group. Ele contêm os atributos da entidade "Pessoa". O segundo campo "dados" contêm os atributos agrupados por relacionamento e entidade 2, associados a entidade 1.

```
1  {
2      "_id" : {
3          "_id" : 1.0,
4          "nome" : "João"
5      },
6      "dados" : [
7          {
8              "Carro" : {
9                  "_id" : 101,
10                 "placa" : "ABC-1234",
11                 "cor" : "Azul"
12             },
13             "Dirige" : {
```

```
                "observacão" : "Trabalho"
            }
        },
        {
            "Carro" : {
                "_id" : 103,
                "placa" : "DEF-5678",
                "cor" : "Vermelho"
            },
            "Dirige" : {
                "observacão" : "Lazer"
            }
        }
    ]
}
```

Listagem 1.4 – Exemplo de resultado da consulta Q1 implementada para o mapeamento da Figura 1 utilizando o método *aggregate*.

Com o mapeamento da Figura 1, torna-se necessário percorrer e juntar dois documentos separados, o que dependendo do volume de dados armazenado pode se mostrar muito custoso. Um mapeamento alternativo, que pode melhorar o desempenho de Q1, é mostrado na Figura 2. Nesta figura, a entidade "Carro" é mapeada para seu próprio tipo de documento específico ("DocTypeCarro"), mas também aparece de forma embutida - e duplicada - no tipo de documento dedicado à entidade "Pessoa".

Como resultado, a implementação da consulta fica mais simples, já que os dados estão estruturados da forma exata em que são necessários para atender Q1. A Listagem 1.5 mostra um exemplo de implementação, que consiste na utilização do método *find*, que seleciona todos os documentos da coleção "DocTypePessoa". Porém, a diferença da consulta especificada na Listagem 1.1, neste exemplo de mapeamento, não é necessário executar outra operação sobre o documento "DocTypePessoa", uma vez que os dados requisitados na consulta Q1 estão presentes nesse documento.

```
db.DocTypePessoa.find();
```

Listagem 1.5 – Consulta Q1 implementada para o mapeamento da Figura 2.

Na Listagem 1.6 é apresentada o exemplo de resultado da execução da consulta da Listagem 1.5 com o método *find*.

```
{
  "_id": 1,
  "nome": "João",
  "carros":
```

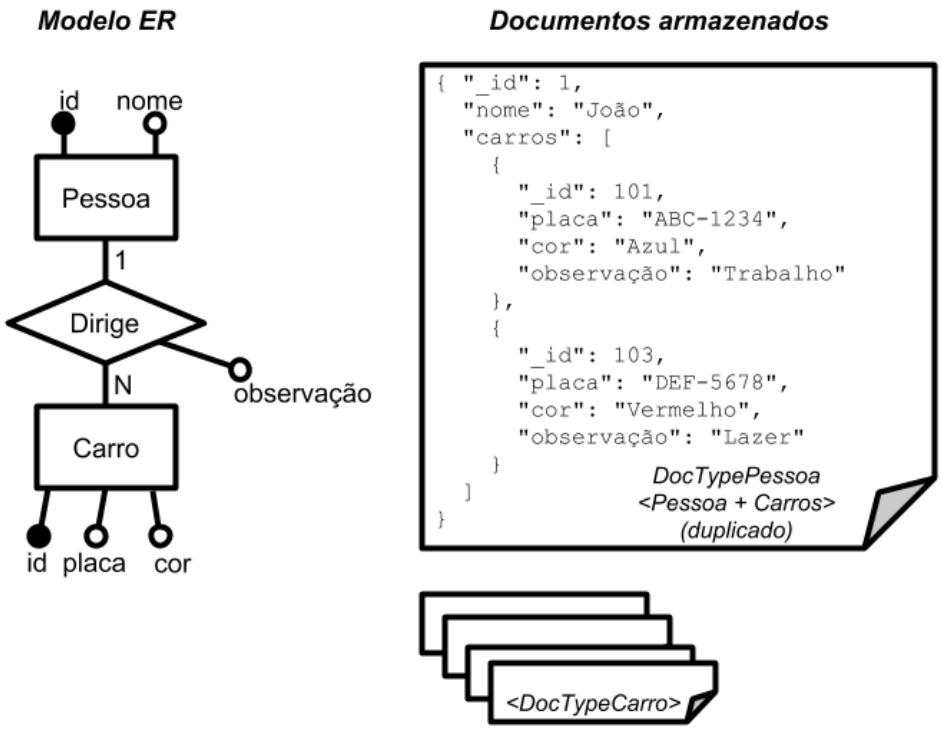

Figura 2 – Exemplo de mapeamento ER para MongoDB com documentos embutidos.

```
  [
    {
        "_id": 101,
        "placa": "ABC-1234",
        "cor": "Azul",
        "observacão": "Trabalho"
    },
    {
        "_id": 103,
        "placa": "DEF-5678",
        "cor": "Vermelho",
        "observacão": "Lazer"
    }
  ]
}
```

Listagem 1.6 – Exemplo de resultado da consulta Q1 implementada para o mapeamento da Figura 2 utilizando o método *find*.

A Figura 3 resume o problema, ilustrando o impacto dos diferentes mapeamentos e a necessidade de trabalho manual para cada situação.

Como se pode perceber, projetar o esquema de armazenamento NoSQL pode se

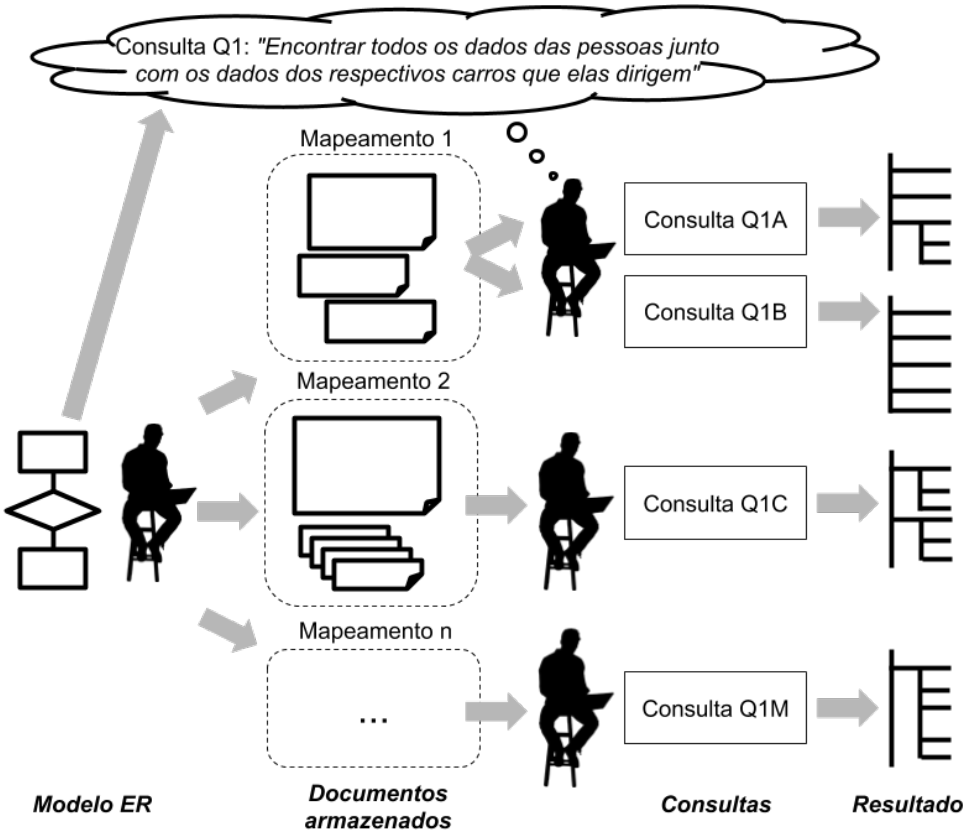

Figura 3 – Resumo do problema sendo abordado nesta pesquisa.

tornar uma tarefa difícil, pois para um mesmo modelo conceitual ER, dependendo das consultas e dos requisitos de desempenho, gera-se uma necessidade de criar diferentes mapeamentos, com duplicações, documentos embutidos e outras decisões que tem sério impacto no desenvolvimento. Do ponto de vista do desenvolvimento das consultas, que é o tema desta pesquisa, isso também tem impacto. Como se pode observar nas Listagens 1.1, 1.3 e 1.5, a implementação das consultas e até mesmo os resultados podem mudar conforme o mapeamento adotado, exigindo estudos cuidadosos no projeto inicial.

Mas o problema não se restringe ao desenvolvimento inicial. É bastante plausível assumir, neste cenário, que o esquema pode variar ao longo do ciclo de vida de software, à medida que os requisitos são refinados ou mesmo novos requisitos são identificados. Caso aconteça, o impacto é potencializado pelo fato de que será exigido retrabalho para atualizar as consultas.

1.2 Objetivo

A literatura contempla trabalhos que buscam solucionar parte desses problemas, seguindo normalmente dois caminhos:

1. existem trabalhos que buscam oferecer um padrão para a escrita de consultas de forma independente do tipo de banco NoSQL, seja utilizando linguagens ou APIs. No entanto, a maioria se limita a consultas simples, não investigando soluções para consultas complexas (ATZENI; BUGIOTTI; ROSSI, 2012), (SELLAMI; BHIRI; DEFUDE, 2014), (LI; MA; CHEN, 2014);

2. também existem trabalhos que fazem o mapeamento entre um modelo conceitual independente de SGBD NoSQL para os diferentes modelos de dados NoSQL (BUGIOTTI et al., 2013), (FIORAVANTI et al., 2016), (LIMA; MELLO, 2015). Dessa forma, é possível que o desenvolvedor trabalhe na definição de dados sem precisar se preocupar muito com os detalhes do SGBD, ou qual modelo NoSQL será utilizado.

Este trabalho procura unir esses dois caminhos para uma solução mais completa e abrangente, propondo uma abordagem que:

- permita a especificação de consultas de forma independente do tipo de banco de dados NoSQL utilizado;
- possibilite consultas não triviais, tais como junções entre entidades; e que
- as consultas gerem sempre a mesma estrutura no resultado final, independente do mapeamento entre o modelo conceitual e o modelo físico NoSQL.

O trabalho de Parent e Spaccapietra (1984) apresenta uma álgebra sobre modelos Entidade-Relacionamento, incluindo operadores de seleção e junção, contemplando assim o primeiro e segundo itens da solução. Tal álgebra permite que o engenheiro de software possa trabalhar em um nível de abstração onde os detalhes do modelo físico, tais como o tipo do banco de dados ou o esquema de armazenamento utilizado, não influenciam o código das consultas. Em outras palavras, o engenheiro de software cria modelos das entidades e relacionamentos e cria as consultas sobre tais modelos.

Porém a semântica da álgebra de Parent e Spaccapietra (1984) é definida apenas de maneira teórica, o que significa que é necessário implementá-la para cada caso específico.

Assim, neste trabalho a álgebra de Parent e Spaccapietra (1984) foi estendida de forma a proporcionar uma semântica operacional focada no modelo de documentos conforme implementado pelo MongoDB. Foram implementadas transformações que automaticamente traduzem as consultas especificadas para modelos físicos do MongoDB. Além disso, as transformações levam em consideração as diferentes possibilidades de mapeamento físico, gerando sempre resultados consistentes com a álgebra de Parent e Spaccapietra (1984).

Para concretizar essa semântica, foram adotados conceitos e técnicas do desenvolvimento de software dirigido por modelos (*Model-Driven Engineering* - MDE), que tem como premissa exatamente o que se pretendeu alcançar nesta pesquisa. A ideia em torno do MDE aplicado ao desenvolvimento de sistemas é identificar as abstrações que permitam criar sistemas, independentemente das tecnologias das plataformas (BRUNELIERE; CABOT; JOUAULT, 2010) (neste caso os diferentes tipos de bancos e esquemas NoSQL). Desta maneira, os engenheiros de software podem trabalhar em um nível mais alto de abstração, em modelos que melhor captam a essência e a lógica da aplicação (neste caso, o modelo ER e a álgebra ER). Consequentemente, é possível reduzir as tarefas repetitivas de codificação, que ficam a cargo de transformações automatizadas (neste caso, transformações que geram consultas JavaScript para o MongoDB). Além disso, o MDE possibilita que o desenvolvedor fique protegido das complexidades requeridas para implementação nas diferentes plataformas (FRANCE; RUMPE, 2007) (neste caso, as diferentes possibilidades de mapeamento entre modelo conceitual e físico).

1.3 Metodologia de pesquisa

Para alcançar o objetivo proposto, este projeto de pesquisa se dividiu em fases: (i) uma fase teórica, que envolveu o levantamento bibliográfico e trabalhos correlatos; (ii) uma fase de construção, responsável pelo planejamento e desenvolvimento da extensão da álgebra ER; e (iii) uma fase de avaliação, responsável por analisar sua funcionalidade.

Na fase teórica (i), foi realizada uma revisão sistemática de literatura de acordo com o processo proposto por Kitchenham e Charters (2007) a fim de conhecer o estado-da-arte da pesquisa. Esta fase gerou o Capítulo 2 deste documento, e o principal resultado observado foi que o tema de pesquisa é relevante e vem sendo investigado pela comunidade acadêmica, mas que poucos trabalhos se aprofundam na questão de consultas complexas.

A fase de construção (ii) consistiu da aplicação dos conceitos e técnicas da MDE. Assim, foram criados 3 metamodelos: dois a partir dos elementos do Modelo de Entidade Relacionamento e do banco de dados orientado a documentos, e um terceiro metamodelo que atua de intermediário entre os dois metamodelos descritos anteriormente. É por meio desse metamodelo intermediário que o Engenheiro de Software define o mapeamento entre as Entidades e Relacionamentos e os tipos de documentos a serem armazenados. Posteriormente, foi desenvolvida uma Gramática Livre de Contexto (GLC) para dar uma sintaxe concreta à álgebra de Parent e Spaccapietra (1984), permitindo assim sua utilização na prática. Por fim, foi implementado um algoritmo que converte uma consulta inicial especificada com as regras da GLC definida, para consultas em JavaScript, levando em consideração o mapeamento do Modelo ER para o esquema MongoDB. Neste trabalho, o termo esquema MongoDB é utilizado como referência à estrutura de como os

dados são armazenados. Como outros SGBDs NoSQL, o MongoDB não implementa de fato um esquema, porém o termo é utilizado com este propósito no manual do MongoDB (MONGODB, 2018).

A fase de avaliação (iii) envolveu dois estudos: um exemplo de sistema acadêmico e um sistema real. O objetivo foi verificar se, para uma mesma consulta e com diferentes mapeamentos, o resultado se mantêm consistente com a semântica original da álgebra de Parent e Spaccapietra (1984). Neste sentido, a avaliação foi bem sucedida, pois para todos os casos analisados os resultados foram positivos.

1.4 Limitações

Inicialmente, pretendia-se implementar completamente a semântica prevista na álgebra ER de Parent e Spaccapietra (1984), incluindo consultas simples a entidades, projeções, junções de duas entidades, junções de três ou mais entidades, funções de agregação e consultas aninhadas.

No entanto, e conforme já alertado por outros pesquisadores (SELLAMI; BHIRI; DEFUDE, 2014), percebeu-se durante esta pesquisa que o problema não tem solução trivial. Constatou-se que a quantidade de combinações possíveis de mapeamentos, mesmo para relacionamentos simples, é muito grande. Neste trabalho foram investigadas 26 possibilidades diferentes de mapeamento para relacionamento com cardinalidade 1-1, 6 para cardinalidade 1-N e duas possibilidades para cardinalidade N-N. Existem ainda outras que ficaram fora do escopo. Constatada essa dificuldade, foi necessário reduzir o escopo da implementação, no seguinte sentido: a gramática livre de contexto e os metamodelos foram definidos completamente. No entanto, o algoritmo de geração das consultas ficou restrito à junção de duas entidades e parte da projeção. Tal escolha se justifica para um trabalho de mestrado, já que essa parte do problema é justamente aquela em que há maior complexidade, devido às várias combinações possíveis de mapeamento das entidades e relacionamentos, com as diferentes cardinalidades, para documentos, considerando, por exemplo, a existência de documentos aninhados, duplicação e outros tipos de desnormalização.

Como consequência, a avaliação também se limitou à projeção de atributos da consulta e junção entre duas entidades, o que novamente, não foi uma tarefa trivial dada a grande variedade de combinações possíveis.

Outra limitação do trabalho é a preocupação com o desempenho. Neste momento, o foco foi em garantir a corretude das consultas e a consistência dos resultados com o que foi especificado. Assim, as consultas geradas não serão necessariamente as mais otimizadas para cada caso. No entanto, uma vez que se tenha a abordagem completa, é possível, além de otimizar as consultas geradas por meio de análise de código e dos modelos, gerar

sugestões de mapeamento de forma a produzir o melhor desempenho possível para as consultas planejadas.

1.5 Organização da Dissertação

Este primeiro capítulo apresentou o contexto sobre o qual esse trabalho de mestrado foi desenvolvido, os objetivos desta pesquisa, limitações e também a metodologia utilizada durante seu desenvolvimento.

No Capítulo 2 traz-se a revisão bibliográfica, a fim de apresentar tanto o atual estado da prática, quanto o atual estado da arte, sobre o contexto trabalhado no escopo desta dissertação.

No Capítulo 3 apresenta-se a extensão da álgebra ER. Neste capítulo são definidos os metamodelos e o algoritmo de conversão de consultas para a linguagem JavaScript.

No Capítulo 4 apresenta-se dois estudos realizados para avaliar o resultado das consultas executadas em MongoDB.

Por fim, as contribuições e as conclusões referentes a esta dissertação são apresentadas pelo Capítulo 5, juntamente com sugestões para trabalhos futuros.

2 Revisão Bibliográfica

Este capítulo descreve os principais conceitos relacionados a esta dissertacão de mestrado. Primeiramente, é definido o conceito de bancos de dados não relacionais. Em seguida, são apresentados os diferentes modelos de dados NoSQL. Por fim, são apresentados os conceitos de desenvolvimento de software dirigido por modelos. Na Seção 2.3 discute-se o *estado-da-arte*: são apresentados e discutidos trabalhos acadêmicos que possuem proposta relacionada à desta pesquisa, e que constituem avanço ainda não consolidado na indústria.

Os trabalhos acadêmicos aqui apresentados foram encontrados nas plataformas digitais *ACM Digital Library*[1], *Scopus*[2] e *IEEE Xplore*[3] após buscas com as palavras chave: *mapping, query* e *mongodb* e selecionados baseados em sua relevância com o contexto e com a solução proposta neste trabalho. Essa revisão da literatura, realizada de forma semi-sistemática, foi feita durantes os anos 2016 até o fim de 2017.

2.1 Bancos de dados não relacional

O termo "NoSQL" fez sua primeira aparição no final da década de 1990 com o nome de um banco de dados relacional de código aberto (STROZZI, 2010). Liderado por Carlo Strozzi, esse banco de dados armazena suas tabelas sob a forma de arquivos ASCII. O nome NoSQL vem do fato de que o banco de dados não utiliza SQL como uma linguagem de consulta. Em vez disso, ele é manipulado por meio de *shell scripts* (SADALAGE; FOWLER, 2013). NoSQL é uma abreviação da frase "*Not Only SQL*" ou seja, "Não Apenas SQL" e demonstra uma tendência da comunidade em encontrar alternativas para complementar a linguagem SQL.

Johan Oskarsson, um desenvolvedor de software de Londres, organizou uma reunião em 2009, em São Francisco, com o objetivo de descobrir mais sobre armazenamentos alternativos de dados. Ele necessitava um nome curto e fácil de lembrar para a reunião, de maneira que uma busca que utilizasse esse nome encontrasse rapidamente a reunião. O termo "NoSQL" é o resultado dessa reunião. Oskarsson não esperava que o nome se tornaria uma tendência tecnológica como um todo, dado que na época só pensava em dar um nome para a reunião (SADALAGE; FOWLER, 2013).

As primeiras implementações foram desenvolvidas pelas empresas como o Google e Amazon, os armazenamentos respectivamente são Bigtable (CHANG et al., 2008) e

[1] http://dl.acm.org/
[2] http://scopus.org
[3] http://ieeexplore.ieee.org/Xplore/home.jsp

Dynamo (DECANDIA et al., 2007). Os bancos de dados NoSQL atuam sem um esquema, permitindo que sejam adicionados, livremente, campos aos registros do banco de dados, sem ter de definir primeiro quaisquer mudanças na estrutura (SADALAGE; FOWLER, 2013).

Bancos de dados NoSQL a princípio não são compatíveis com ACID, mas eles mantêm o teorema CAP (GILBERT; LYNCH, 2002) e princípios BASE (PRITCHETT, 2008). A propriedade ACID garante que uma série de operações terá sucesso dentro de uma transação, geralmente emprega bloqueios de leitura e escrita, para proteger seus dados não confirmados de serem vistos e modificados por alguma outra transação (FADOUA; AMEL, 2016), (YALAMANCHI; GAWLICK, 2009).

Em 2000, o professor Eric Brewer apresentou o famoso teorema CAP que quer dizer: consistência, disponibilidade, tolerância a partição da rede. A idéia central do teorema CAP é que um sistema distribuído não pode atender às três necessidades do setor simultaneamente, mas só dois (HAN et al., 2011), como mostrado na Fig. 4. Consistência significa que, sempre que os dados são gravados, todas as operações de leitura que ocorrem após a gravação devem retornar a mais recente versão dos dados. Disponibilidade significa que podemos sempre esperar que cada operação termine em uma resposta desejada. Tolerância a partições significa que o banco de dados ainda pode ser lido e escrito quando partes dele são completamente inacessíveis (POKORNY, 2013), (GILBERT; LYNCH, 2002).

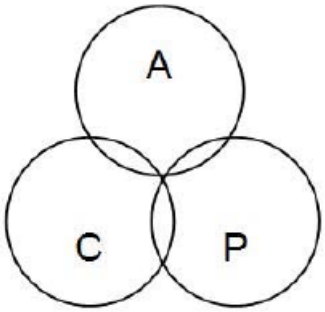

Figura 4 – O Teorema CAP (WANG; TANG, 2012)

O modelo BASE é completamente diferente do modelo ACID. BASE é a abreviatura de basicamente disponível, estado leve, e consistência eventual. O sistema funciona basicamente o tempo todo (basicamente disponível), não tem de ser consistente durante todo o tempo (estado leve), mas o sistema de armazenamento garante que se novas atualizações são feitas para o objeto, eventualmente todos os acessos retornarão o último valor atualizado (POKORNY, 2013), (WANG; TANG, 2012).

2.1.1 Modelos de dados

Sobre o modelo de dados, bancos de dados NoSQL podem geralmente ser divididos em chave-valor, documento, grafo e orientado a coluna. A Tabela 1 ilustra exemplos de

NoSQLs e seus modelos de dados. Muitos bancos de dados não se encaixam exatamente nas categorias; por exemplo, o OrientDB[4] é considerado tanto um banco de dados de documentos quanto de grafos (SADALAGE; FOWLER, 2013). Em OrientDB todos os vértices e arestas são documentos.

Tabela 1 – Modelos de Dados (SADALAGE; FOWLER, 2013)

Modelo de dados	**Exemplos de bancos de dados**
Chave-valor	BerkeleyDB
	LevelDB
	Memcached
	Project Voldemort
	Redis
	Riak
Documentos	CouchDB
	MongoDB
	OrientDB
	RavenDB
	Terrastore
Família de Colunas	Amazon SimpleDB
	Cassandra
	HBase
	Hypertable
Grafos	FlockDB
	HyperGraphDB
	Infinite Graph
	Neo4J
	OrientDB

Ao utilizar qualquer armazenamento de dados NoSQL, há uma necessidade de entender como os seus recursos se comparam com os armazenamentos de dados padrão de SGBDR habituais. O principal motivo é entender quais recursos estão faltando e como a arquitetura do aplicativo precisa mudar para utilizar melhor os recursos de um armazenamento de dados NoSQL (SADALAGE; FOWLER, 2013). Nas Seções 2.1.1.1, 2.1.1.2 e 2.1.1.3 apresenta-se exemplos de comparações do modelo NoSQL com o modelo relacional. Mapear uma consulta semelhante a SQL a uma consulta de banco de dados de grafo não explorará os recursos do banco de dados de grafo (KOLEV et al., 2016), por esse motivo não fazemos a comparação com o SGBDR, mas mostramos um exemplo simples do modelo de grafos.

Nas seções a seguir são apresentados mais detalhes sobre os principais modelos de dados. Nessas seções são também mostradas comparações com os elementos do SGBD Oracle. Tais comparações são feitas para facilitar o entendimento dos conceitos NoSQL, assim como se apresenta no livro de Sadalage e Fowler (2013).

[4] http://orientdb.com/orientdb/

2.1.1.1 Chave-valor

Os dados armazenados neste tipo de modelo são constituídos por duas partes: uma *string* que representa a chave, e os dados a serem armazenados, que representam o valor, assim criando um par "chave-valor" (NAYAK; PORIYA; POOJARY, 2013). Uma chave identifica unicamente um valor e este valor pode ser estruturado ou completamente desestruturado, texto, JSON, XML e assim por diante. A chave é definida pelo programador (POKORNY, 2013), e é utilizada principalmente para o acesso pela chave primária (SADALAGE; FOWLER, 2013).

Os mais populares sistemas que utilizam este tipo de armazenamento são o Riak, Redis, Voldemort e DynamoDB (PEREIRA; OLIVEIRA; RODRIGUES, 2015). Apresenta-se na Tabela 2 a relação entre os principais elementos do SGBD Oracle e seus correspondentes no SGBD Riak.

Tabela 2 – Relação entre os elementos dos SGBDs Oracle e Riak (SADALAGE; FOWLER, 2013)

Oracle	**Riak**
Instância de banco de dados	Cluster Riak
Tabela	Bucket
Linha	Chave-valor
Rowid	Chave

Uma instância de banco de dados Oracle é um conjunto de estruturas de memória que gerenciam arquivos de banco de dados. O modo de operação padrão do Riak é trabalhar como um *cluster* composto por múltiplos nodos, ou seja, vários *hosts* de dados conectados. Cada *host* no *cluster* executa uma única instância do Riak, conhecida como um nodo Riak. Um *Bucket* é usada para definir um *keyspace* para armazenar objetos no Riak. *Buckets* são comparados a tabelas em bancos de dados Oracle.

Um servidor Oracle atribui cada linha em cada tabela com um único *Rowid* para identificar a linha na tabela. Riak fornece uma interface baseada em HTTP REST e REST é usada para mapear recursos para URLs e interagir com eles usando o conjunto de expressões CRUD: *POST*, *GET*, *PUT* e *DELETE*, que significam criar, ler, atualizar e excluir, respectivamente (BARON et al., 2016).

Na Listagem 2.1 apresenta-se um exemplo de Baron et al. (2016) sobre operação de criação (*PUT*) relacionada a objetos e armazenadas com HTTP. Primeiro especifica-se o endereço do *bucket* (linha 1). Em seguida, com a expressão *PUT* (linha 2) é atualizado o *bucket* chamado "carro", com a chave "bmw", no *Header* (linha 3) especifica-se o tipo de conteúdo, que neste caso é JSON. No *Body* (linhas 4 a 7) descreve-se como valor as propriedades do objeto carro.

```
1 http://SERVER:PORT/riak/BUCKET/KEY
```

```
PUT http://localhost:8091/riak/carro/bmw
Header: "Content-Type: application/json"
Body: ‘{
"modelo" : "520",
"ano": "2014"
}’
```

Listagem 2.1 – Exemplo de expressão PUT no Riak (BARON et al., 2016)

2.1.1.2 Documentos

Nos armazenamentos orientados a documento, um documento pode compreender valores escalares, listas ou mesmo documentos aninhados; não há esquema para documentos, e cada documento pode ter seus próprios atributos, definidos em tempo de execução (POKORNY, 2013). No entanto, MongoDB fornece validação de esquema durante atualizações e inserções desde a versão 3.6 (MONGODB, 2018). Os documentos armazenados nestes sistemas são normalmente de tipos bastante comuns como XML, JSON, PDF, entre outras (NAYAK; PORIYA; POOJARY, 2013). Esses documentos são estruturas de dados na forma de árvores hierárquicas e autodescritivas, constituídas de mapas, coleções e valores escalares (SADALAGE; FOWLER, 2013).

Características como modelo de dados persistente, replicação de documentos, distribuição automática entre servidores e estrutura de índice podem ser aplicadas ao modelo de armazenamento de documentos (MATHEW; KUMAR, 2015). Entre armazenamentos de documentos, MongoDB é um dos mais adotados, oferecendo uma interface de programação rica para manipular tanto documentos inteiros como campos individuais simples (ATZENI; BUGIOTTI; ROSSI, 2012). Apresenta-se na Tabela 3 a relação entre os principais elementos do SGBD Oracle e seus correspondentes no SGBD MongoDB.

Tabela 3 – Relação entre os elementos dos SGBDs Oracle - MongoDB (SADALAGE; FOWLER, 2013)

Oracle	**MongoDB**
Instância de banco de dados	Instância MongoDB
Esquema	Banco de dados
Tabela	Coleção
Linha	Documento
Rowid	_id
Junção	DBRef

Uma instância MongoDB aloja vários bancos de dados. Como mencionado na Seção 2.1.1.1, uma instância de banco de dados Oracle é um conjunto de estruturas que gerenciam arquivos de banco de dados. Um esquema no Oracle é representado por uma

coleção de objetos tais como: tabelas, sequencias, índices e outros objetos. Um esquema no Oracle é associada a um banco de dados no MongoDB.

MongoDB armazena em bancos de dados coleções de documentos. DBRefs são referências de um documento para outro usando o valor do campo *_id* do primeiro documento, nome da coleção e, opcionalmente, seu nome de banco de dados. Ao incluir esses nomes, DBRefs permitem que documentos localizados em várias coleções sejam mais facilmente vinculados com documentos de uma única coleção. DBRefs é um método para relacionar documentos no MongoDB. Na Listagem 2.2 apresenta-se um exemplo de recuperação de dados de uma coleção chamada "carro" em MongoDB no formato JSON. O conteúdo de um documento é especificado com chaves abertas e fechadas (linhas 1, 8, 9, 17). Dentro das chaves são recuperados os campos e os valores separados por dois pontos (linhas 2, 3, 10, 11). Um valor pode conter uma matriz de documentos que é representado por colchetes (`[]`) (linhas 4, 7, 12, 16), dentro da matriz os campos e valores são retornados entre virgulas (linhas 5, 6, 13, 14, 15).

```
1  {
2    "_id" : 1.0,
3    "marca" : "bmw",
4    "outros dados" : [{
5      "modelo" : "520",
6      "ano" : "2014"
7    }]
8  }
9  {
10   "_id" : 2.0,
11   "marca" : "toyota",
12   "outros dados" : [{
13     "modelo" : "155",
14     "ano" : "1998",
15     "cor" : "preto"
16   }]
17 }
```

Listagem 2.2 – Exemplo de armazenamento de uma coleção em MongoDB

Na Listagem 2.3 apresenta-se a sintaxe para retornar os documentos de uma coleção chamada "carro". É preciso indicar o nome da coleção entre aspas simples dentro do parêntese do operador *getCollection*. O comando *find* do MongoDB é similar ao *select* na linguagem SQL. Dentro do parêntese desse comando, é possível indicar expressões condicionais que identificam os documentos a retornar na consulta.

```
1 db.getCollection('carro').find({})
```

Listagem 2.3 – Exemplo de sintaxe de recuperação de dados em MongoDB

2.1.1.3 Colunar

Semelhante ao gerenciamento de banco de dados relacional, armazenamentos de dados de colunas também armazenam tabelas, mas como colunas de dados em oposição a linhas como em SGBDRs (KAUR; RANI, 2015). Os dados são modelados em forma de pares chave-valor e armazenados em um arquivo de sistema distribuído. O sistema atribui a cada valor um *timestamp* para fins de consistência de dados, resolver conflitos de gravação, lidar com dados desatualizados e outras ações. Contudo, a combinação <chave, nome da coluna, timestamp> representa as coordenadas do valor (DEHDOUH et al., 2014), (SADALAGE; FOWLER, 2013).

Os bancos de dados orientados a colunas são adequados para aplicações de mineração de dados e análise, onde o método de armazenamento é ideal para as operações comuns realizadas nos dados (NAYAK; PORIYA; POOJARY, 2013). Os sistemas NoSQL mais populares deste tipo são o HBase, Cassandra e HyperTable (PEREIRA; OLIVEIRA; RODRIGUES, 2015). Apresenta-se na Tabela 4 a relação entre os principais elementos do SGBD Oracle e seus correspondentes no SGBD Cassandra.

Tabela 4 – Relação entre os elementos dos SGBDs Oracle - Cassandra

Oracle	**Cassandra**
Instância de banco de dados	Cluster
Esquema	Keyspace
Tabela	Família de colunas
Linha	Linha
Coluna (a mesma para todas as linhas)	Coluna (podem ser diferentes por linha)

Cada família de colunas pode ser comparada a um contêiner de linhas em uma tabela de Oracle, onde a chave identifica a linha, que é constituída de múltiplas colunas. A diferença com Cassandra é que as linhas não têm de ser as mesmas colunas, e novas colunas podem ser adicionadas a qualquer linha, a qualquer momento, sem a obrigatoriedade de ter de adicioná-las às outras linhas também (SADALAGE; FOWLER, 2013).

No armazenamento de bancos de dados relacional os dados são armazenados conforme Tabela 5. Já no banco de dados orientado a coluna o armazenamento de dados se da conforme a Tabela 6.

Tabela 5 – Armazenamento no banco de dados relacional

bmw	520	2014	null
toyota	115	1998	preto

2.1.1.4 Grafos

Bancos de dados de grafo modelam o banco de dados como uma estrutura de rede contendo nós e arestas que relacionam nós para representar relação entre eles. Com-

Tabela 6 – Armazenamento no banco de dados orientado a colunas

bmw
520
2014

toyota
115
1998
preto

parando com Modelo Entidade-Relacionamento, um nó corresponde a uma entidade, a propriedade de um nó corresponde um atributo e a relação entre entidades corresponde a uma aresta (KAUR; RANI, 2013). Os bancos de dados de grafos são compatíveis com as propriedades ACID e oferecem suporte a *rollback*. Podem ser usados para uma variedade de aplicações como aplicativos de redes sociais, software de recomendação, bioinformática, gerenciamento de conteúdo, segurança e controle de acesso, entre outros (NAYAK; PORIYA; POOJARY, 2013).

Bancos de dados de grafos garantem consistência por meio de transações. Eles não permitem relacionamentos pendentes: o nó inicial e o nó final sempre têm de existir e nós somente podem ser excluídos se não tiverem relacionamentos associados a eles (SADALAGE; FOWLER, 2013). As soluções mais populares representativas deste tipo de bases de dados NoSQL são o sistema Neo4J, Titan e OrientDB (PEREIRA; OLIVEIRA; RODRIGUES, 2015).

Na Figura 5 apresenta-se um exemplo de estrutura de grafo. O nó chamado Motorista, na verdade, é um nó com a propriedade nome configurada como Alice. Também vemos que as arestas tem propriedades que permitem organizar os nós; por exemplo, os nós Motorista e Carro1 têm uma aresta que os conecta com um relacionamento do tipo possui (Alice possui o Carro1).

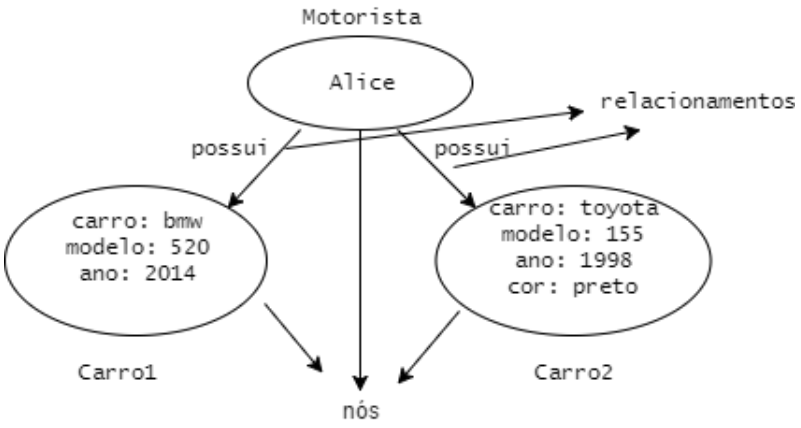

Figura 5 – Exemplo de estrutura de grafo

Na Listagem 2.4 ilustra um trecho de código na linguagem Cypher, utilizada pelo banco de dados Neo4j, para criação dos nós “Motorista”, “Carro1” e “Carro2”, por meio de comandos *CREATE*.

```
1 CREATE (a:Motorista { nome: 'Alice'})
2 CREATE (b:Carro1 { carro: 'bmw', modelo:250, ano:2014})
3 CREATE (c:Carro2 { carro: 'toyota', modelo:155, ano:1998, cor:
    preto })
```

Listagem 2.4 – Criação de nós no Neo4J utilizando a linguagem Cypher

A Listagem 2.5 mostra um trecho de código que estabelece um relacionamento entre os nós "Motorista" e "Carro1". O comando *MATCH* (linha 1) busca um padrão no grafo, semelhante ao comando *SELECT* da linguagem SQL. O comando *WHERE* (linha 2) usa predicados para filtrar elementos. O comando *CREATE* efetivamente cria a aresta entre os nós (linha 3). O relacionamento é representado por `->`, `<-` ou `-`. Além disso, pode-se incluir um rótulo (tipo) no nó, por exemplo: `-[:TYPE]->`. O comando *RETURN* (linha 4) retorna os resultados.

```
1 MATCH (a:Motorista),(b:Carro1)
2 WHERE a.nome = 'Alice' AND b.carro = 'bmw'
3 CREATE (a)-[r:RELTYPE { name: a.nome + '->' + b.carro }]->(b)
4 RETURN r
```

Listagem 2.5 – Criação de relacionamentos no Neo4J utilizando a linguagem Cypher

2.2 Desenvolvimento de software dirigido por modelos

Apesar da evolução das técnicas e ferramentas de desenvolvimento de sistemas, preocupações como modelagem, reúso, produtividade, manutenção, documentação, validação, otimização, portabilidade e interoperabilidade ainda são recorrentes no desenvolvimento de software. O conceito de desenvolvimento dirigido por modelos surgiu com o objetivo de ajudar a resolver esses problemas (KLEPPE; WARMER; BAST, 2003). Desenvolvimento de software dirigido por modelos (MDE) fornece uma base organizada para criação de métodos, isto é, linguagens específicas, para engenharia de domínios específicos. Também fornece técnicas para análise e transformação que possibilitem a automatização de tarefas de engenharia e sua integração em processos de software personalizados (WENDE et al., 2011).

Uma das preocupações mais importantes na elaboração da solução de MDE é a especificação de uma linguagem de modelagem que permite que os produtos de software necessários sejam representados no nível conceitual sem ambigüidade. Atualmente, existem duas opções que são as mais utilizadas para a definição dessas linguagens de modelagem: a especificação de uma linguagem de modelagem específica de domínio (DSML) ou a personalização de UML (GIACHETTI; MARIN; PASTOR, 2009). Uma linguagem de domínio específico (DSL) é uma linguagem de programação ou linguagem de especificação executável que oferece, através de notações e abstrações apropriadas, um poder expressivo

centrado e geralmente restrito a um determinado domínio do problema (DEURSEN et al., 2000). Mernik, Heering e Sloane (2005) citam os exemplos da linguagem APT (*Automatically Programmed Tool*) um padrão internacional para a programação de máquinas-ferramentas controladas numericamente (ROSS, 1978), que data de 1957-58, e da mais famosa BNF, ou *Backus-Naur Form*, linguagem para especificação de gramática criada em 1959. Uma DSML é uma DSL voltada para modelagem.

2.2.1 Principais elementos do MDE

Apesar de todos os esforços e pesquisas em relação ao MDE, ainda assim a automatização das transformações não é uma tarefa simples (LUCRÉDIO, 2009). A Figura 6 mostra os principais elementos necessários para essa abordagem, e como eles são combinados (SILVA et al., 2013):

- Ferramentas de Modelagem: utilizadas para produzir modelos que descrevem conceitos de domínio. Os modelos criados por essas ferramentas precisam seguir regras de semântica, uma vez que serão interpretados pelo computador. Normalmente é utilizada o DSL.
- Ferramentas para definir transformações: são utilizadas para gerar outros modelos e código fonte a partir de modelos recebidos como entrada. Através delas são construídas as regras de mapeamento.
- Modelos: servem de entrada para transformações e representam o conhecimento do domínio da aplicação.
- Mecanismos para executar as transformações: executam as transformações seguindo as regras de mapeamento do modelo para modelo ou modelo para código. Mantêm informações de rastreabilidade, possibilitando muitas vezes saber o origem de cada elemento gerado, seja modelo ou código fonte.
- Outros modelos e código fonte: são resultantes do processo de transformação.(SILVA et al., 2013)

2.2.2 Metamodelagem

As principais abordagens MDE baseiam-se no conceito de metamodelagem, que dá suporte a diferentes linguagens de programação e ajuda a garantir que os modelos construídos estejam completos e semanticamente corretos. Além disso, possibilita a definição e execução de transformações (LUCRÉDIO, 2009). Um metamodelo trata-se de um modelo que fornece base para construir outro modelo. A diferença entre modelagem e metamodelagem é subjetiva. Apesar de ambos serem modelos, um é especificado utilizando-se o

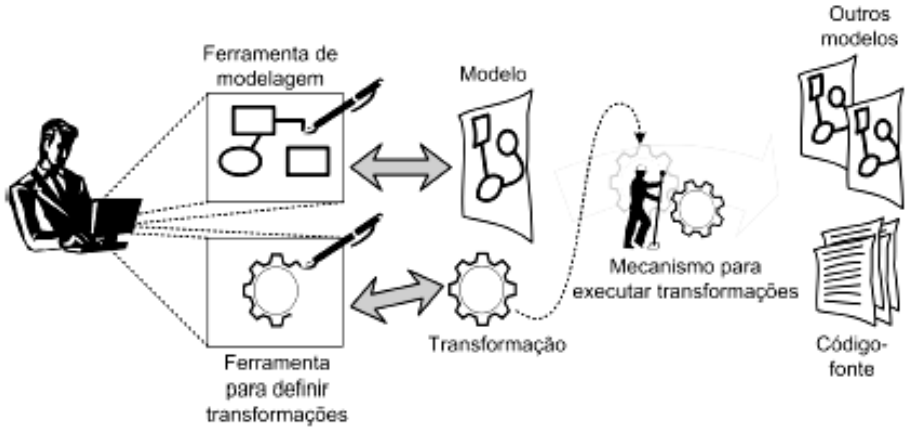

Figura 6 – Principais elementos do MDE (LUCRÉDIO, 2009)

outro, ou seja, um modelo deve estar em conformidade com o metamodelo (GRONBACK, 2009). A Figura 7 apresenta a arquitetura de metamodelagem comumente utilizada pela comunidade de desenvolvimento.

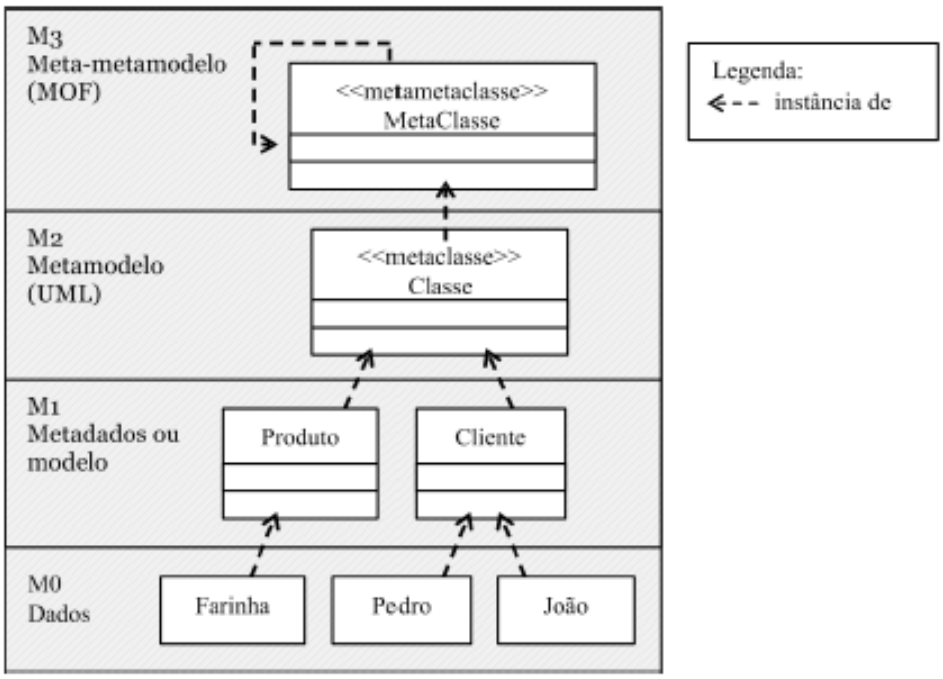

Figura 7 – Arquitetura de metamodelagem (LUCRÉDIO, 2009)

- Nível M3: a camada de meta-metamodelo é utilizada para definir linguagens de modelagem (como a UML). Normalmente, o meta-metamodelo é uma instancia de sí próprio.

- Nível M2: a camada de metamodelo é utilizada para especificação de modelos. A especificação UML é um exemplo de metamodelo.

- Nível M1: a camada de metadados ou modelo eleva o nível de abstração, são dados que descrevem os dados. É nessa camada que são descritos os conceitos de domínio.

- Nível M0: corresponde aos dados que instanciam os conceitos de domínio.

2.2.3 Principais abordagens da indústria para MDE

Há na indústria diversas abordagens para o MDE. Dentre todas essas, a mais conhecida é *Model-Driven Architecture* (MDA). MDA enfatiza modelos e transformações entre eles para alcançar interoperabilidade entre sistemas, reusando produtos intermediários. Essa abordagem provê um framework conceitual e um conjunto de padrões para expressar modelos e relações entre eles, para que assim seja possível uma transformação automática de modelos. A ideia central por trás do MDA é o aumento do nível de abstração, a fim de separar as preocupações entre negócios e plataformas (YANG; LIU; WANG, 2012).

Na MDA, transformações são utilizadas para transformar um modelo em outro, sucessivamente, até o código-fonte. O metamodelo utilizado nesta abordagem é o *Meta Object Facility* (MOF). O MOF consiste em um padrão orientado a objetos que permite a definição de classes com atributos e relacionamentos, sendo bastante semelhante ao diagrama de classes UML. Também define uma interface padrão de acesso aos dados dos modelos, oferecendo métodos para manipulação dos dados e dos metadados (LUCRÉDIO, 2009). O aumento do nível de abstração da MDA nos traz conceitos como CIM (*Computation Independent Model* - Modelo Independente de Computação), PIM (*Platform Independent Model* - Modelo Independente de Plataforma) e PSM (*Platform-Specific Model* - Modelo Específico de Plataforma). CIM descreve somente requisitos de usuários, sendo independente de plataforma e de computação. Seu principal foco é descrever problemas de dominio, por esse motivo, é também conhecido como Modelo de Domínio. O CIM pode ser mapeado tanto para um PIM, quanto para um PSM. PIM envolve a computação e busca soluções, porém não leva em consideração plataformas. Sua função é descrever e analisar subdomínios de uma aplicação. Um PSM é baseado em um PIM e leva em consideração a plataforma. O principal objetivo é descrever o *design* do subdomínio de uma aplicação (YANG; LIU; WANG, 2012).

A empresa *Sun Microsystems*, comprada pela Oracle em 2009, também fornece meios de se trabalhar com MDE. A Sun, com auxílio do seu ambiente de desenvolvimento NetBeans[5], busca oferecer ao desenvolvedor a opção de criar sistemas simultaneamente no modelo e código-fonte. A empresa oferece ferramentas que seguem os padrões OMG (SILVA et al., 2013).

[5] http://netbeans.org

Outra abordagem muito importante é o conjunto de ferramentas *Eclipse Modeling Project*, destacando-se o *Eclipse Modeling Framework* (EMF). O EMF permite a manipulação de modelos segundo seu correspondente metamodelo, seguindo um metamodelo denominado *Ecore*, ao invés de MOF. O *Ecore* surgiu como implementação MOF, mas evoluiu para uma forma mais eficiente, a partir da experiencia obtida após alguns anos na construção de ferramentas. O EMF não segue fielmente o padrão JMI, mais sim utiliza um subconjunto de mapeamentos de metamodelo para Java otimizados para manipulação em memória, o que torna mais eficiente e mais simples de ser utilizado (MOORE et al., 2004).

Há também os demais projetos do *Eclipse Modeling Project*, incluindo algumas ferramentas que apoiam o MDE e o desenvolvimento de DSLs, que podem ser integradas ao ambiente de desenvolvimento Eclipse (SILVA et al., 2013). Um exemplo é Xtext, que é uma ferramenta para desenvolvimento de gramática DSL usando uma linguagem EBNF, extensão da BNF. Pode-se também citar as ferramentas M2T (*Modelo to text - Modelo para texto*) para transformação e geração de código, das quais se destacam o JET *(Java Emitter Templates)* que é um motor de *template* tradicional que é usado por EMF e Xpand que é um motor de *template* de complemento para DSLs (GRONBACK, 2009).

2.3 Estado da Arte

São apresentados alguns trabalhos que possuem relação com o trabalho desenvolvido nesta pesquisa. São apresentados os trabalhos: Framework SOS de Atzeni, Bugiotti e Rossi (2012), API REST ODBAPI de Sellami, Bhiri e Defude (2014), Modelo de dados único de Sellami, Bhiri e Defude (2016), Modelagem de dados orientada a consulta de Li, Ma e Chen (2014), Linguagem BQL de Curé et al. (2011) e Álgebra ER de Parent e Spaccapietra (1984) que empregam abordagens que visam aliviar as consequências da heterogeneidade de interfaces por os vários sistemas NoSQL. Esses trabalhos são apresentados e discutidos em detalhes, por se aproximar bastante dos objetivos desta proposta. Em seguida são apresentadas outras abordagens relacionadas.

2.3.1 Framework SOS (ATZENI; BUGIOTTI; ROSSI, 2012)

Atendendo o problema da heterogeneidade das linguagens e as interfaces que oferecem os vários sistemas Atzeni, Bugiotti e Rossi (2012) desenvolveram o framework SOS (*Save Our Systems*), um ambiente de programação onde os bancos de dados não-relacionais podem ser definidos, consultados e acessados por uma aplicação. O ambiente é baseado em uma abordagem de metamodelos, no sentido de que as interfaces específicas de cada sistema são mapeadas para um modelo comum. A aplicação foi implementada para três SGBDs dentro da família de NoSQL: Redis, MongoDB e HBase.

A Figura 8 apresenta a arquitetura da plataforma SOS, que é organizada nos seguintes módulos: a interface comum (*Common Interface*), que permite o acesso aos dados armazenados em diferentes sistemas de gestão de dados, é definida com operações muito básicas e gerais: *PUT*, *GET* e *DELETE* que estão baseadas nos identificadores dos objetos. A meta-camada (*Meta Layer*) representa uma definição genérica dos modelos de dados construídos com base em três principais construtores: *Struct, Set e Attribute*, os quais permitem gerenciar coleções de objetos. Os controladores (*Handler*) geram as chamadas apropriadas para sistema do banco de dados específico.

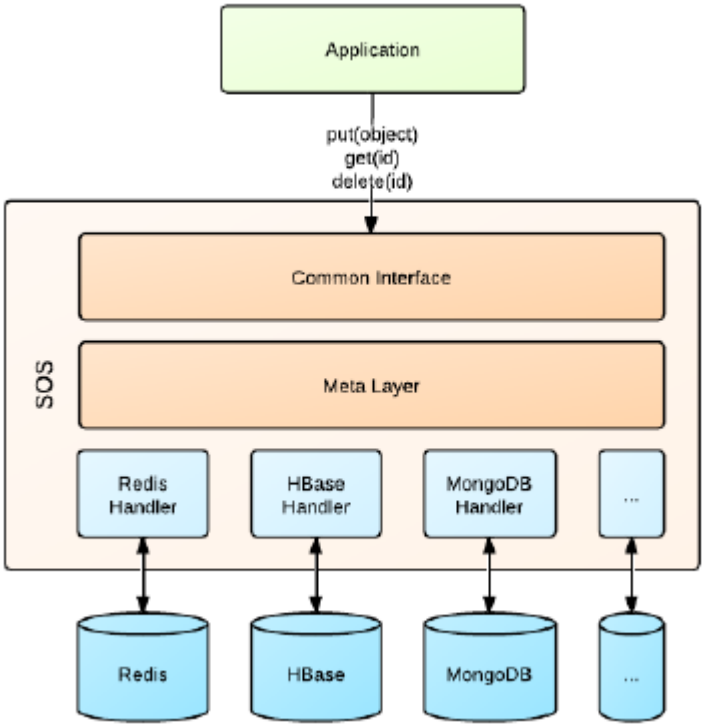

Figura 8 – Arquitetura da plataforma SOS (ATZENI; BUGIOTTI; ROSSI, 2012)

Para a definição da plataforma de SOS que caracterizou uma API Java, se expõe os seguintes métodos correspondentes às operações básicas definidas na interface comum:

- `void put (String collection, String ID, Object o)`
- `void delete (String collection, String ID)`
- `Object get (String collection, String ID)`
- `Set <Object>get (Query q)`

Esses métodos lidam com objetos arbitrários Java e são responsáveis por suas especificações em sistemas NoSQL. Na versão atual da ferramenta, foi implementada a

meta-camada em JSON. Os autores apresentam também um exemplo do esquema de dados acerca da definição de uma versão simplificada do Twitter, aplicativo popular de rede social. A implementação se baseia no mapeamento de "*following*" entre a meta-camada e o formato JSON. Cada solicitação do aplicativo é codificada em termos de operações de bancos de dados NoSQL nativas e com o objeto JSON é dada uma representação adequada, estruturada, específica para o SGBD utilizado.

O aplicativo pode ser implementado por meio de um pequeno número de classes. Cada uma das classes é implementada usando um ou mais objetos de um SGBD diferente: MongoDB para usuários, Redis para *tweets* e HBase para os *followships*. O código para o método principal, *sendTweet()*, da classe que manipula os *tweets*, é ilustrado na Listagem 2.6.

```
1  NonRelationalManager tweetsDB =
2          new RedisDbNonRelationalManager();
3  NonRelationalManager followshipsDB =
4          new HBaseNonRelationalManager(); ...
5  public void sendTweet(Tweet tweet)
6  {
7      // ADD TWEET TO THE SET OF ALL TWEETS
8      tweetsDB.put("tweets", tweet.getId(),tweet);
9      // ADD TWEET TO THE TWEETS SENT BY THE USER
10     Set<Long> sentTweets =
11             tweetsDB.get("sentTweets", tweet.getAuthor());
12     sentTweets.add(tweet.getId());
13     tweetsDB.put("sentTweets",tweet.getAuthor(),sentTweets);
14     //NOTIFY FOLLOWERS
15     Set<Long> followers =
16             followshipsDB.get("followers",tweet.getAuthor());
17     for (Long followerId : followers)
18     {
19     Set<Long> unreadTweets =
20             tweetsDB.get("unreadTweets", followerId);
21         unreadTweets.add(tweet.getId());
22     }
23         tweetsDB.put("unreadTweets", followerId, unreadTweets);
24 }
```

Listagem 2.6 – Método principal sendTweet (ATZENI; BUGIOTTI; ROSSI, 2012)

O código da Listagem 2.6 mostra os objetos de dois bancos de dados de interesse, *tweetsDB* (linha 1) e *followshipsDB* (linha 3). Ambos os objetos são do tipo *NonRelationalManager* (classe principal que contêm as operações *PUT*, *GET* e *DELETE*). Cada

objeto é instanciado para um SGBD diferente, neste caso Redis (linha 2) e HBase (linha 4). Em seguida, as operações que envolvem os *tweets* são especificadas em uma maneira muito simples, em termos de operações *PUT* e *GET* sobre os objetos "DB" (linhas 7-23). É interessante notar que o código se refere aos sistemas específicos apenas na inicialização dos objetos *tweetsDB* e *followshipsDB*. Assim, seria possível substituir um sistema por outro simplesmente alterando o construtor para esses objetos.

2.3.2 API REST ODBAPI (SELLAMI; BHIRI; DEFUDE, 2014)

Sellami, Bhiri e Defude (2014) apresentam uma API REST chamada ODBAPI (*OPEN-PaaS-DataBase API*) que permite que um aplicativo instalado na plataforma *OpenPaaS* interagir e executar operações CRUD em armazenamentos de dados relacionais e armazenamentos de dados NoSQL. *OpenPaaS* é uma plataforma que trata registros de computação na nuvem. A Tabela 7 apresenta uma comparação entre os diferentes conceitos utilizados em cada SGBD que são usados no contexto do projeto *OpenPaaS*. Foi estabelecido um modelo de recursos que define os diferentes recursos de destino por ODBAPI: *Environment, Database, EntitySet, Entity e Attribute*. Para implementação da API foram consideradas três categorias de armazenamento de dados por meio dos seguintes SGBDs: MySQL, Riak e CouchDB.

Tabela 7 – Comparação de conceitos utilizados por ODBAPI (SELLAMI; BHIRI; DEFUDE, 2014)

Relational concepts	CouchDB concepts	Riak concepts	ODBAPI concepts
Database	Environment	Environment	Database
Table	Database	Database	Entity Set
Row	Document	Key/value	Entity
Column	Field	Value	Attribute

A Figura 9 expõe uma visão geral da ODBAPI. Esta API é projetada para fornecer uma camada de abstração e interação sem interrupções com armazenamentos de dados introduzidos em um ambiente na nuvem. A Figura 9 é dividida em quatro partes que apresenta-se a seguir a partir da direita para a esquerda. Em primeiro lugar, estão todos os armazenamentos de dados estabelecidos (relacional e não relacional) com os quais um desenvolvedor pode interagir. Em segundo lugar, encontramos a API proprietária e o *driver* de cada SGBD suportado pela ODBAPI. A terceira parte representa a implementação da ODBAPI. Ela contêm uma implementação específica de cada armazenamento de dados. Em quarto e último lugar estão as diferentes operações que a ODBAPI oferece para o usuário. Cada operação é garantida por um método do REST (por exemplo, *GET, PUT* entre outros). Na especificação são definidos dois tipos de operações, a primeira é dedicada a obter metainformação sobre os recursos usando método GET REST, a segunda

representa as operações de CRUD executadas. No entanto, essas operações não funcionam corretamente sem especificar o armazenamento de dados de destino. Por este motivo, deve ser adicionado um parâmetro de cabeçalho ao parâmetro do HTTP. Foi definido o parâmetro *database-type* que permite especificar o tipo de armazenamento de dados por uma consulta REST.

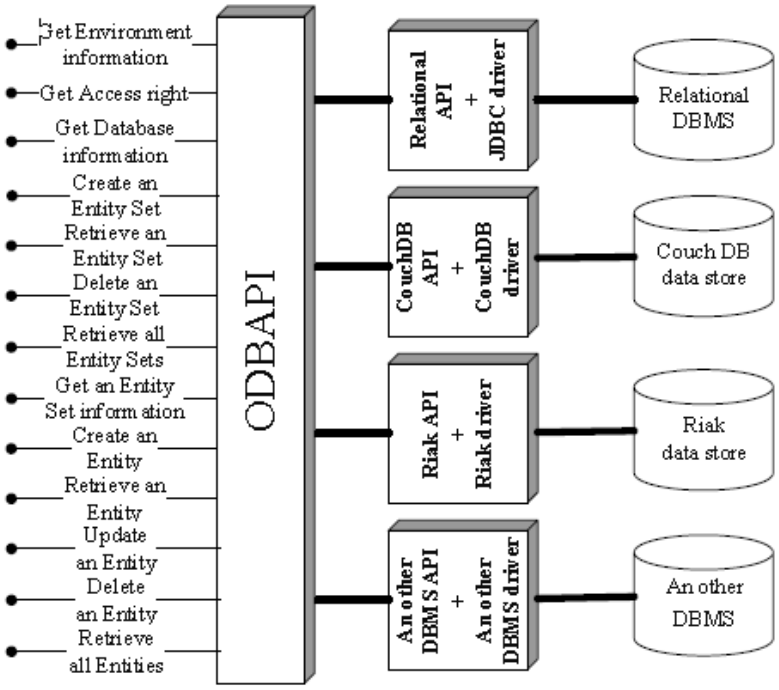

Figura 9 – Uma visão geral de ODBAPI (SELLAMI; BHIRI; DEFUDE, 2014)

A Listagem 2.7 apresenta um exemplo de solicitação HTTP referente à recuperação de uma Entidade do tipo documento, e a resposta obtida. A solicitação consiste na especificação do tipo de método HTTP (*GET*) seguindo-se o recurso de destino (*/odbapi/Person/111*) (linha 1). A solicitação também deve incluir o tipo de SGBD, neste caso o couchDB (linha 2) e o formato da resposta, neste caso JSON (linha 3). A resposta recebida inclui um código de status HTTP (linha 4), e o tipo de conteúdo retornado (linha 5). Em seguida, o documento referente à entidade é apresentado (linhas 6 a 9).

```
1 > GET /odbapi/person/111
2 > Database-type: database/couchDB
3 > Accept: application/json
4 < HTTP/1.1 200 OK
5 < Content-Type: application/json
6 < {
7 < entityID: "111",
8 < name: "personA"
9 < }
```

Listagem 2.7 – Recuperação de entidade com solicitação HTTP

2.3.3 Modelo de dados único (SELLAMI; BHIRI; DEFUDE, 2016)

Sellami, Bhiri e Defude (2016) oferecem um modelo de dados único para interagir com armazenamentos de dados heterogêneos, relacional e NoSQL. Baseado no modelo de dados único, consultas podem ser expressas utilizando uma API REST chamada ODBAPI (*OPEN-PaaS-DataBase API*) que permite que um aplicativo instalado na plataforma *OpenPaaS* interagir e executar operações CRUD. Propõem também um armazenamento de dados virtuais (*Virtual Data Stores* - VDS), que age como um mediador e interage com armazenamentos de dados suportados por ODBAPI. Este componente de tempo de execução suporta a execução de consultas simples e complexas sobre armazenamentos de dados heterogêneos.

Na Figura 10 apresenta-se uma visão geral da abordagem proposta. Os principais componentes são:

- Modelos de dados unificado: modelo de dados que abstrai os modelos de armazenamento de dados integrados subjacentes e fornece uma visão comum e unificada para definir consultas. Este componente separa as consultas em subconsultas para interagir com linguagem específica de cada armazenamento de dados.

- Serviços API REST: com base no modelo de dados unificado, foi definido um modelo de recurso sobre a API REST, chamada ODBAPI, permitindo interagir com os armazenamentos de dados de uma forma única e uniforme. O modelo de dados permite expressar as consultas e ODBAPI interagir com os armazenamentos.

- Armazenamento de dados virtuais (*Virtual Data Store*): os serviços REST do *wrapper* permitem a execução de consultas simples sobre os armazenamentos de dados. No entanto, eles não são destinados a executar consultas complexas (como junção e união). Os VDSs permitem aos desenvolvedores expressar suas consultas de junção em vários armazenamentos de dados de forma declarativa e assumir a carga de suas execuções. Foi definido o banco de dados CouchDB como VDS por contar com recursos de junção.

- Componentes dedicados a detecção (*discovery*) e desenvolvimento (*deployment*): responsáveis por encontrar ambientes em nuvem apropriadas e implantar várias aplicações de armazenamento de dados sobre eles, respectivamente.

O modelo de dados proposto unifica os diferentes conceitos dos armazenamentos de dados existentes, na Tabela 8 mostra-se uma comparação entre os diferentes conceitos utilizados em três categorias de armazenamento de dados que são comumente utilizados no ambiente da nuvem.

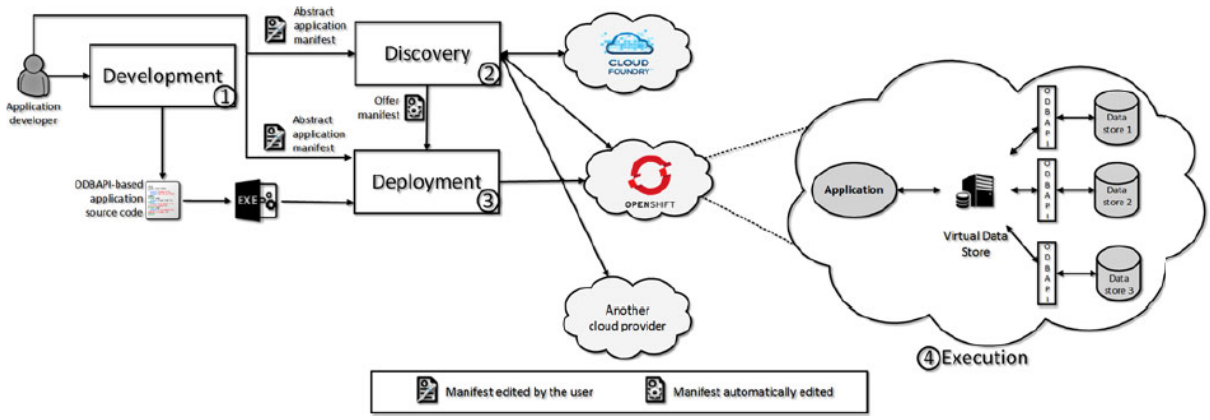

Figura 10 – Uma visão geral da abordagem proposta por (SELLAMI; BHIRI; DEFUDE, 2016)

Tabela 8 – Comparação de conceitos utilizados nos diferentes armazenamentos de dados (SELLAMI; BHIRI; DEFUDE, 2016)

Relational concepts	MongoDB concepts	Riak concepts	Unifying data model concepts
Database	Database	Environment	Database
Table	Collection	Database	Entity Set
Row	Document	Key/value	Entity
Column	Field	Value	Attribute

Foram definidos três armazenamentos de dados com as seguintes características: (*Database*)− > conjunto de entidades (*Entity Set*) -> entidades (*Entity*)-> {atributos (Attribute) }:

- Armazenamento relacional MySQL: personDB->person-> person->{personId, personName, personCountry, personAffiliation}
- Armazenamento de documentos MongoDB: dblpDB->dblp-> dblp-> {title, author, conference, year}
- Armazenamento chave-valor Riak: rankDB->conferenceRanking-> conferenceRanking->{conference, rank}

A partir de uma consulta que envolve diferentes armazenamentos de dados, é realizada uma avaliação e otimização de execução de consulta com base no VDS. É considerada a seguinte consulta de junção: Retornar a afiliação e o nome dos autores tendo pelo menos um artigo publicado em uma conferência classificada como “A”. Essa consulta associa três conjuntos de entidades que são *dblp, person e conferenceRanking*. A Listagem 2.8 apresenta um exemplo de solicitação HTTP referente à recuperação de atributos do tipo documento. A solicitação consiste na especificação do tipo de método HTTP (*POST*)

seguindo-se o recurso de destino (*/odbapi/joinquery*) (linha 1). A solicitação também deve incluir o tipo de SGBD, neste caso o VirtualDataStore (linha 2) e o formato da resposta, neste caso JSON (linha 3). Em seguida, o usuário expressa sua consulta como uma simples expressão *select-from-where* no formato JSON e envia para o VDS usando ODBAPI (linhas 4-20).

```
1 > POST /odbapi/joinquery
2 > Database-Type: database/VirtualDataStore
3 > Accept: application/json
4 {
5 ‘‘select’’: [
6 ‘‘person.personName’’,
7 ‘‘person.personAffiliation’’
8 ],
9 ‘‘from’’: [
10 ‘‘person’’,
11 ‘‘dblp’’,
12 ‘‘ConferenceRanking’’
13 ],
14 ‘‘where’’: [
15 ‘‘person.personName in dblp.author’’,
16 ‘‘dblp.conference = ConferenceRanking.
17 conference’’,
18 ‘‘ConferenceRanking.Rank=A’’
19 ]
20 }
```

Listagem 2.8 – Consulta envolvendo join (SELLAMI; BHIRI; DEFUDE, 2016)

A VDS constrói a arvore algébrica sem qualquer otimização. Então, é otimizado usando regras de reescrita algébrica para privilegiar primeiro a execução de operações unárias nos armazenamentos de dados integrados, a fim de reduzir o tamanho dos conjuntos de entidades transferidos. Uma vez que recebe uma consulta complexa, a VDS constrói um plano de execução de consulta ideal, composto por subconsultas ao nível da fontes de dados de destino, operações de conversão (*conversion*), de envio (*shipping*) e uma consulta final recombina os resultados parciais. Além das operações algébricas, um plano de execução de consulta pode usar duas novas outras operações *ship* e *convert*, que permite respetivamente, a transferência de conjunto de resultados e a conversão de conjunto de resultados (ver Figura 11). Para cada consulta, a API proposta reescreve-a na linguagem de consulta proprietária do armazenamento de dados. Em seguida, converte o resultado para o formato JSON antes de responder à aplicação. No trabalho não fica claro o formato dos dados retornados pelas subconsultas feitas na linguagem nativa dos

bancos NoSQL, nem a maneira em que o VDS consegue realizar a operação *join* entre os diferentes armazenamentos de dados relacional e não relacional.

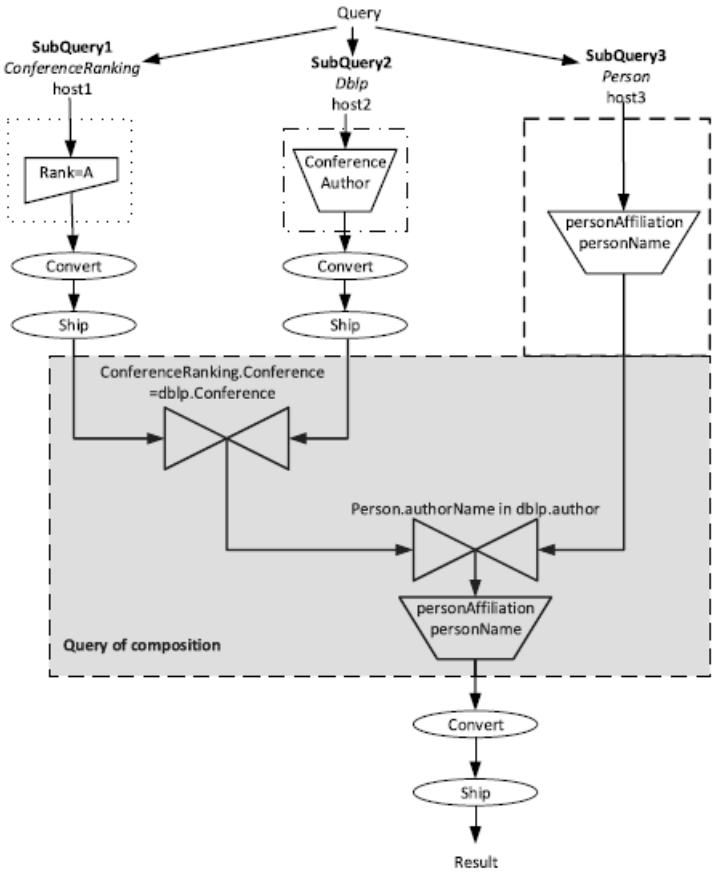

Figura 11 – Plano de execução de consulta (SELLAMI; BHIRI; DEFUDE, 2016)

2.3.4 Modelagem de dados orientada a consulta (LI; MA; CHEN, 2014)

Uma metodologia para modelagem de dados orientada a consulta para bancos de dados NoSQL é proposta por Li, Ma e Chen (2014). Eles apresentam um framework da abordagem QODM - *Query Oriented Data Modeling* mostrada na Figura 12, que inclui três fases: (i) descrever a estrutura dos dados armazenados e os requisitos da consulta de dados da aplicação; (ii) com base nos dados armazenados de requisitos de estrutura e de consulta de dados, a *QODM Analyzer* gera o modelo de dados para bancos de dados NoSQL; e (iii) com base no modelo de dados, o *QODM Analyzer* gera o esquema de dados para bancos de dados NoSQL.

Para definir a estrutura de dados armazenados, é analisado o domínio do problema (*Problem Domain*) ou PD na Figura 12, definindo as entidades e os relacionamentos e descritos com diagramas de classes UML. Os requisitos da consulta de dados são representados por um par de tuplas <Ck, Ct>. Uma classe na Ck indica que é parte da chave da consulta, uma classe no Ct significa que uma classe é uma parte do resultado da consulta. Para gerar o modelo de dados o analisador irá mapear cada consulta em uma entidade

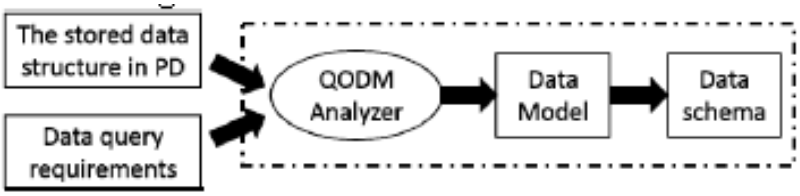

Figura 12 – Abordagem de modelagem de dados orientada a consulta (LI; MA; CHEN, 2014)

agregada ou uma entidade de índice. A partir destes diagramas são gerados os esquemas de dados em documentos JSON, independentes da plataforma dos diferentes bancos de dados NoSQL.

Foi definido o metamodelo do esquema de dados que é apresentado na Figura 13. O esquema de dados deve ser um modelo independente da plataforma, dado que existem vários tipos de bancos de dados NoSQL. Como mostrado na Figura 13, cada entidade do modelo é representado por um elemento *Entity*. Um elemento *Entity* possui vários elementos *Field*. Um elemento *Field* representa uma propriedade do *Entity*. Cada elemento *Field* tem uma propriedade *Name* e um elemento *Value*. Existem três tipos de elementos *Value*: *EntityValue, BasicValue e ListValue*. O elemento *BasicValue* representa o elemento *Value* que é um valor de tipo básico, como *string, numeric, float*, entre outros. O elemento *BasicValue* é representado por uma *string*. O elemento *EntityValue* representa o valor do elemento *Value* como uma entidade (*Entity*). O elemento *ListValue* representa o valor do elemento *Value* como uma lista.

Com base no metamodelo, é exibida a estrutura do documento do esquema de dados, conforme representada na Listagem 2.9.

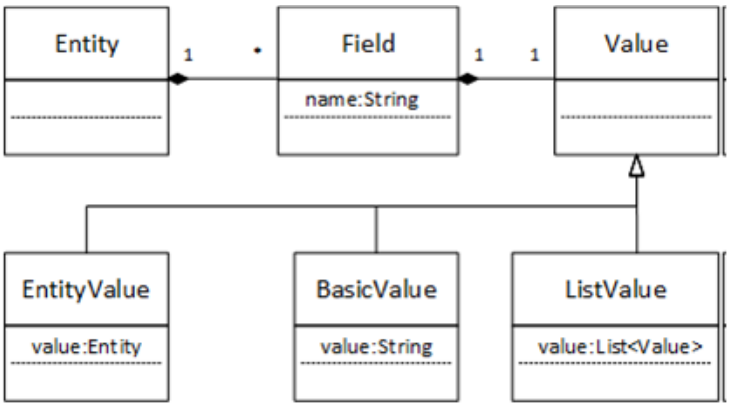

Figura 13 – Metamodelo do esquema da dados QODM (LI; MA; CHEN, 2014)

```
1 //Entity
2 {
3     "<Field.name>": "<BasicValue.value>",
4     "<Field.name>": "<EntityValue.value>",
5     "<Field.name>": [<ListValue.value.get(0)>,
```

```
    <ListValue.value.get(1)>,
    <ListValue.value.get(2)>
    ...]
}
```

Listagem 2.9 – Estrutura do esquema de dados QODM (LI; MA; CHEN, 2014)

Para avaliar a disponibilidade e independência de plataforma da abordagem QODM, foi usado um aplicativo de armazenamento de *e-mail* chamado *"ElasticInbox"*. O aplicativo mostra seu modelo de dados e os esquemas de dados em Apache Cassandra. Eles podem ser usados para avaliar o modelo de dados e os esquemas de dados que são gerados pela abordagem QODM. Os pesquisadores extraíram três requisitos de consulta de dados do código fonte do *"ElasticInbox"* e o esquema de dados de uma delas está representada na Listagem 2.10. As entidades *MailboxID* e *LabelID* possuem um valor do tipo *BasicValue*, e a entidade chamada *Messages* possui um valor do tipo *ListValue*. Com base na comparação entre o esquema de dados gerado e os dados reais do esquema de "ElasticInbox", foi avaliada a disponibilidade da abordagem QODM.

```
"MailboxID": <mailboxid>
"LabelID": <labelid>
"messages": [
<messageID0>,
<messageID1>,
...]
```

Listagem 2.10 – Esquema de dados ElasticInbox (LI; MA; CHEN, 2014)

2.3.5 Linguagem BQL (CURÉ et al., 2011)

O trabalho de Curé et al. (2011), investiga a eficiência das consultas sobre bancos de dados relacionais e NoSQL, com abordagem na solução de mapeamento baseado em caminhos de acesso. Foi proposta uma linguagem de consulta BQL - *Bridge Query Language* que permite a transformação de uma consulta SQL definida sobre um banco de dados relacional para a consulta executada sobre um determinado banco de dados NoSQL. A linguagem faz a tradução do SQL para BQL e a partir de BQL para a linguagem de consulta suportada por cada armazenamento NoSQL.

Com relação ao mapeamento baseado em caminhos de acesso, é definida a seguinte especificação: RelationT(a,b,c) $\overleftarrow{a}$ EntityS(<key;value>), onde RelationT e EntityS denotam a relação entre conjunto de coleções, família de colunas ou relações do banco de dados NoSQL. Um caminho de acesso com o atributo 'a' é definido quando a entidade de destino (bancos de dados NoSQL) oferece um acesso eficiente, usando uma chave ou um índice, a uma coleção, conjunto de colunas ou tupla. A arquitetura geral do proces-

samento de consulta é apresentada na Figura 14. Primeramente, um usuario grava uma consulta SQL sobre um esquema relacional. Posteriormente, a consulta SQL será traduzida na linguagem de consulta interna BQL usando as declarações de mapeamento. Então para cada consulta BQL, é realizada uma segunda transformação, desta vez para gerar uma consulta no sistema de banco de dados NoSQL. Assim, para cada implementação de sistemas NoSQL, é definida um conjunto de regras para tradução da consulta BQL.

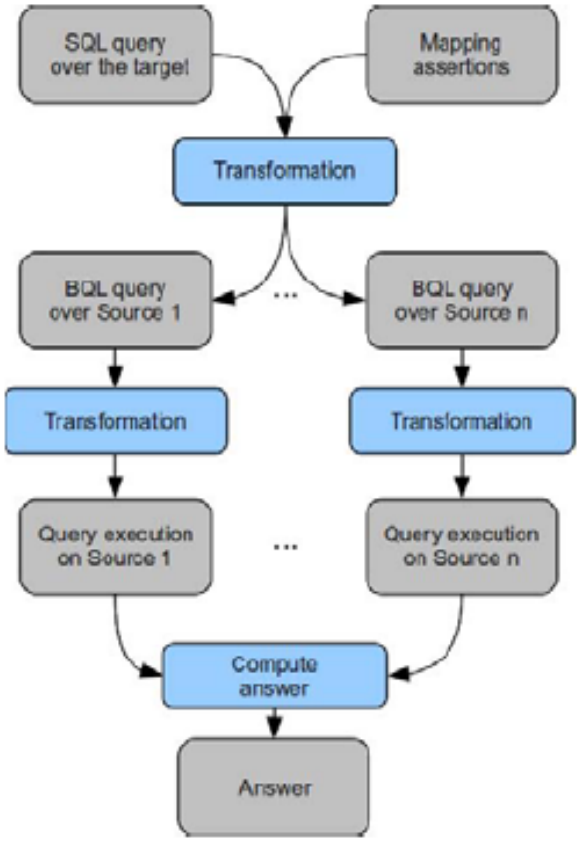

Figura 14 – Arquitetura de processamento de consulta com abordagem BQL (CURÉ et al., 2011)

São definidas declarações de mapeamento para traduzir uma consulta na linguagem SQL para linguagem BQL. Na Tabela 9 são exibidos os seguintes mapeamentos: o mapeamento número 1 representa o caminho de acesso por ("*"), isso considerando que todos os atributos da coleção drugD estão indexados. A coleção drugD está armazenada em um banco de dados orientado a documentos. Neste mapeamento, o atributo "c" não está mapeado para qualquer atributo de origem, desde que as informações não estejam disponível no banco de dados que contenha a coleção. O mapeamento número 3 mostra a utilização de varias entidades num único mapeamento. Intuitivamente, esta consulta acessa uma determinada entrada de família de coluna drugNameC identificada pelo nome do medicamento ($\overleftarrow{\mathtt{n}}$) e itera seus identificadores de medicamentos, representadas por "i", que são chaves (usando expressão 'IN KEY') então ele usa esses identificadores para

acessar na família de coluna drugC.

Foram extraídas semelhanças entre consultas expressas sobre bancos de dados NoSQL e implementados padrões em *templates* BQL e associadas a métodos Java com o propósito de realizar o mapeamento da consulta BQL para o banco de dados destino. A linguagem BQL contêm um conjunto de palavras reservadas cuja semântica inclui a instrução *GET* que permite definir os valores necessários para o resultado da consulta.

Tabela 9 – Mapeamento de consultas SQL para BQL (CURÉ et al., 2011)

1. `drug(i,l,n,c,p)` $\overleftarrow{*}$ `drugD`(<PKEY AS `i`; name AS `n`, lab AS `l`, price AS `p`>
2. `drug(i,n,l,c,p)` $\overleftarrow{i}$ `drugC`(<PKEY AS `i`; name AS `n`, lab AS `l`, compo as `c`, price AS `p`>)
3. `drug(i,n,l,c,p)` $\overleftarrow{n}$ `drugNameC`(<PKEY AS `n`; FOREACH id AS `i` IN KEY>, `drugC`(<id, lab AS `l`, compo as `c`,price AS `p`>)
4. `therapDrug(i,t)` $\overleftarrow{i}$ `drugD`(<PKEY AS `i`; FOREACH the AS `t` IN `Therap`>)
5. `therapDrug(i,t)` $\overleftarrow{t}$ `therapD`(<PKEY AS `t`; FOREACH id AS `i` IN `Drugs`>)
6. `therapDrug(i,t)` $\overleftarrow{i}$ `DrugC`(<PKEY AS `i`; FOREACH the AS `t` IN `Therap`>)
7. `therapDrug(i,t)` $\overleftarrow{t}$ `therapD`(<PKEY AS `t`; FOREACH id AS `i` IN KEY >)

Apresenta-se na Listagem 2.11 um exemplo de consulta expressa em SQL (linhas 1-4) e traduzida para bancos de dados de documentos (linhas 7-9) e colunar (linhas 11-13). A consulta acessa uma única tabela através de sua chave primária.

```
1  SQL:
2  SELECT drugName, price
3  FROM drug
4  WHERE drugId=3295935;
5
6  /*docDB's BQL: */
7  ans(drugName, price) = docDB.drugD.get({PKEY= 3295935},{name,
8  price})
9  Resposta: (Advil, 1.88)
10 /*colDB's BQL: */
11 ans(drugName, price) = colDB.drugC.get({PKEY= 3295935},{name,
12 price})
13 Resposta: (Advil, 2.05)
```

Listagem 2.11 – Exemplo de consulta SQL mapeada na linguagem BQL (CURÉ et al., 2011)

2.3.6 Álgebra ER (PARENT; SPACCAPIETRA, 1984)

O trabalho de Parent e Spaccapietra (1984) propõe uma definição de um conjunto de operadores algébricos a serem aplicados em um banco de dados de Entidade-

Relacionamento. A álgebra ER inclui os operadores básicos: união, diferença de conjuntos, produto cartesiano, projeção e seleção. Támbem é estudada a operação de junção ("*relationship join*"). O resultado de uma operação deve poder servir como operando para qualquer um dos operadores, de modo que qualquer expressão desejada pode ser formulada usando uma combinação apropriada de operações.

Entramos em detalhes com a operação de junção, que foi utilizada para proporcionar uma semântica operacional focada no modelo de documentos.

Suponha que: $E_1, E_2, ..., E_n$ são tipos de entidades associados através de um relacionamento R (ver Figura 15):

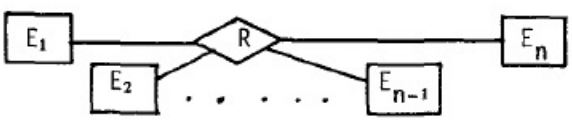

Figura 15 – Relacionamento R, *relationship join* (PARENT; SPACCAPIETRA, 1984)

O *relationship join* de $E_1, ..., E_n$ através de R define um novo tipo de entidade E: $E = E_1 *_R (E_2, E_3, ..., E_n)$.

Uma ocorrência de E consiste em uma ocorrência de E1 juntamente com todas as ocorrências (se houver) de R, com as entidades associadas $E_2, .., E_n$, nas quais a corrente de E1 está envolvida. Em outras palavras, E é um novo tipo de entidade com dois atributos:

- um atributo complexo, obrigatório, monovalorado que agrupa todos os atributos de E1,
- um atributo complexo, opcional, multivalorado agrupando n atributos complexos, obrigatórios e monovalentes, onde cada um desses atributos n, respectivamente, agrupa todos os atributos de $R, E_2, ..., E_n$.

É apresentado o seguinte exemplo extraído do trabalho de Parent e Spaccapietra (1984): suponha que o conteúdo do banco de dados de exemplo - representa os seguintes fatos:

- uma pessoa chamada SMITH tem um contrato 247 com uma empresa chamada LIFE-INS para o seu carro com número 1111 e outro contrato (número 3412) com a mesma empresa do carro número 2222;
- uma pessoa chamada CHEN tem um número de contrato 81 com LLOYD para o número de carro 1234. Então, um novo tipo de entidade PC pode ser construído para armazenar dados sobre pessoas junto com seus carros e companhia de seguros (ver Figura 16), onde () denota valores para atributos complexos e {} denota atributos multivalorados.

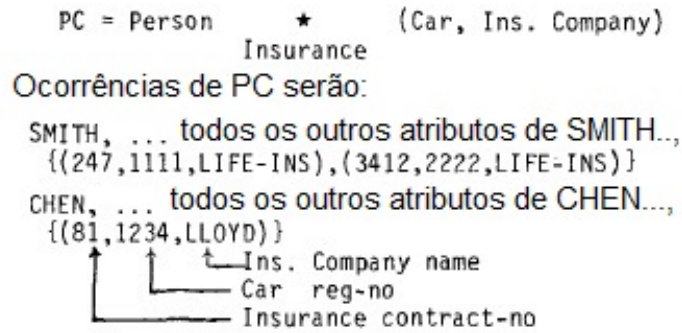

Figura 16 – Exemplo *relationship join* (PARENT; SPACCAPIETRA, 1984)

A composição de *relationship joins* (e de outras operações também) é possível porque o resultado da operação é estabelecido como um novo tipo de entidade (Entidade Computada) que herda os relacionamentos que associam as entidades "internas" (aqueles que aparecem dentro da fórmula que define o novo tipo de entidade) para entidades "externas" (aquelas que não estão envolvidas na fórmula).

A facilidade oferecida por este operador de junção é fundamental para a manipulação de dados, pois permite que a estrutura do resultado desejada seja independente das estruturas existentes gravadas no esquema.

2.3.7 Outras abordagens

As pesquisas apresentadas até este ponto tem uma proposta de facilitar o trabalho do desenvolvedor com diferentes SGBDs relacionais e não relacionais. No geral, todos oferecem suporte para que o desenvolvedor possa realizar a modelagem de dados (LI; MA; CHEN, 2014) ou a manipulação e consulta dos dados de maneira uniforme, seja por meio de uma linguagem de consulta genérica (CURÉ et al., 2011), (KOLEV et al., 2016) ou APIs (ATZENI; BUGIOTTI; ROSSI, 2012), (SELLAMI; BHIRI; DEFUDE, 2014).

No entanto, nestes trabalhos não são exploradas questões referentes a consultas complexas. A maior parte dos exemplos envolve consultas simples, com uma única entidade e recuperação com base no valor de uma chave. Conforme Sellami, Bhiri e Defude (2016), mais investigação é necessária para possibilitar que o desenvolvedor seja capaz de especificar consultas complexas que possam ser executadas em diferentes SGBDs relacionais e não-relacionais.

O trabalho de Parent e Spaccapietra (1984) é o único desses que investiga de forma mais aprofundada a execução de consultas complexas em um nível mais conceitual e com semântica completa. No entanto, a semântica é definida de forma teórica apenas, o que exige trabalho de implementação caso a álgebra venha a ser utilizada.

Também seguindo nessa linha, Kolev et al. (2016) apresentam uma linguagem se-

melhante ao SQL, chamada CloudMdsQL - *Cloud Multidatastore Query Language*, capaz de consultar armazenamentos de dados heterogêneos, relacional e NoSQL, dentro de uma única consulta principal contendo tabelas aninhadas. Estas tabelas aninhadas são apresentadas na forma de subconsulta com expressões nativas dos diferentes armazenamentos, e uma instrução *select* principal é processada e sua correção semântica é verificada. Por exemplo, na Listagem 2.12 é apresentada uma consulta CloudMdsQL simples que contêm duas subconsultas, definidas pelas expressões de tabela denominadas T1 e T2, e dirigidos respectivamente contra os armazenamento de dados DB1, um banco de dados SQL (linha 1) e DB2, um banco de dados MongoDB (linhas 2-5) e a instrução *select* principal (linha 6-8). A definição de uma consulta em expressão nativa (consultas na linguagem de SGBDs NoSQL) deve ser expressa com símbolos de chaves ({*, *}).

```
1 T1(x int, y int)@DB1 = ( SELECT x, y FROM A )
2 T2(x int, z string)@DB2 = {*
3 db.B.find( {$lt: {x, 10}}, {x:1, z:1, _id:0} )
4 *}
5
6 SELECT T1.x, T2.z
7 FROM T1, T2
8 WHERE T1.x = T2.x AND T1.y <= 3
```

Listagem 2.12 – Consultas en diferentes armazenamentos de dados - CloudMdsQL (KOLEV et al., 2016)

A proposta de tratamento de consultas complexas sobre grandes modelos também foi estudada por Daniel, Sunyé e Cabot (2016). Eles propõem a *Mogwai*, um framework que converte consultas de modelo escritas em OCL para Gremlin, uma linguagem de consulta compatível com vários bancos de dados NoSQL orientados a grafos. As expressões Gremlin são geradas dentro do próprio banco de dados, ignorando limitações das APIs existentes e melhorando o desempenho da execução de consultas.

Essas pesquisas (KOLEV et al., 2016; SELLAMI; BHIRI; DEFUDE, 2016; DANIEL; SUNYÉ; CABOT, 2016) possuem o mesmo objetivo deste trabalho, pois exploram o tema de consultas complexas envolvendo diferentes tipos de armazenamento. No entanto, existem algumas diferenças. Os estudos de Kolev et al. (2016) e Sellami, Bhiri e Defude (2016) oferecem suporte às consultas complexas em tempo de execução, não utilizam o modelo conceitual, não levam em consideração as diferentes formas de armazenamento de dados no NoSQL, nem fazem referencia ao uso de cardinalidades. O trabalho de Sellami, Bhiri e Defude (2016) não suporta expresões *group by* e *like*, *inner queries*, disjuncão e negação na cláusula *where*. Já o trabalho de Kolev et al. (2016) permite otimizar a execução da consulta reescrevendo consultas de acordo com as junções e as seleções solicitadas. O otimizador de consultas leva em consideração a disponibilidade e o tipo de índices nos

atributos de junção da relação do lado direito no armazenamento de dados. A otimização é feita a partir de funções de custos definidos pelo usuario.

Neste trabalho, pretende-se utilizar uma abordagem generativa, ou seja, as consultas complexas são transformadas em consultas nativas, para somente então serem executadas, levando em consideração as diferentes formas de mapeamento de um modelo ER para o esquema MongoDB.

Já a pesquisa de Daniel, Sunyé e Cabot (2016) tem uma limitação na abordagem devido a que realiza a conversão de consultas para uma linguagem própria de bancos de dados orientados a grafos, sem levar em conta os demais linguagens e modelos NoSQL.

2.3.8 Considerações finais

Nas Tabelas 10 e 11 é apresentada uma síntese dos trabalhos correlatos apresentados neste capítulo, bem como as características da abordagem proposta.

Este capítulo apresentou conceitos relacionados a bancos de dados não relacionais, desenvolvimento dirigido por modelo, bem como os principais elementos do MDE. Tais conceitos são necessários para entendimento de tópico de pesquisa abordado neste trabalho.

Entre os trabalhos apresentados, alguns deles recuperam dados do armazenamento NoSQL a partir de chaves primárias e a abordagem de consultas complexas é pouco explorada.

A maior dificuldade apontada pelos trabalhos envolvendo modelos de dados é a heterogeneidade dos bancos NoSQL. Existem diversos bancos de dados não relacionais, onde cada um tem linguagem de consulta própria, tipos de armazenamentos, estratégias, ferramentas, modelo de dados e outras características que tornam o processo de integração de bancos de dados uma atividade complexa (DHARMASIRI; GOONETILLAKE, 2013). É necessário levar em consideração que mesmo quando um único tipo de banco NoSQL é utilizado, existe mais de uma estratégia diferente para se organizar os dados. Muitos estudos focam na descrição de metodologias para migração de um banco relacional para bancos NoSQL, propondo e definindo arquiteturas que descrevem a transformação do modelo de dados relacional para um equivalente em bancos NoSQL, a maioria dos casos para bancos de dados NoSQL orientados a chave-valor, orientados a famílias de colunas e orientado a documentos.

A análise dos dados permite concluir que, nenhum trabalho recuperado nesta revisão investiga o assunto das diferentes formas de mapeamento de um modelo conceitual para o modelo físico NoSQL.

Tabela 10 – Trabalhos correlatos e abordagem proposta

Abordagem	**Modelos NoSQL**	**Entrada de dados**	**Consulta**	**Mapeamento**
Atzeni, Bugiotti e Rossi(2012)	Chave-valor, documentos, colunar	Interface comum	Simple	Utiliza métodos para operações básicas (PUT, GET, DELETE)
Sellami, Bhiri e Defude (2014)	Chave-valor, documentos	Modelo de recursos ODBAPI	Simple	Operações garantidas pelo método REST(PUT, GET, DELETE)
Sellami, Bhiri e Defude (2016)	Chave-valor, documentos	Modelo de dados único	Simple; Complexa (junção e união, não suporta group by, like, inner join)	Consultas expressadas em formato JSON
Li, Ma e Chen (2014)	Chave-valor, documentos, colunar, grafos	Estrutura dos dados armazenados e requisitos da consulta de dados	Simple	Cada consulta é mapeada em uma entidade agregada ou uma entidade de índice
Curé et al.(2011)	Documentos, colunar	Consulta SQL	Simple; Complexa (abordam subconsultas, não abordam junções)	Regras de mapeamento de consultas de SQL para BQL
Kolev et al (2016)	Documentos, grafos	Consulta SQL	Complexa (aborda junções)	Consulta principal com linguagem SQL contendo tabelas aninhadas, estas são apresentadas com subconsultas com expressões nativas do banco

Tabela 11 – Trabalhos correlatos e abordagem proposta - continuação

Abordagem	**Modelos NoSQL**	**Entrada de dados**	**Consulta**	**Mapeamento**
Daniel, Sunyé e Cabot (2016)	Grafos	Consulta OCL	Complexa	Tradução de consultas usando transformação modelo para modelo
Abordagem proposta	Documentos	Modelo ER	Simple; Complexa	Mapeamento utilizando técnicas de desenvolvimento de software dirigido por modelos; Consultas são independentes do esquema do banco de dados. Qualquer esquema pode ser escolhido, que o algoritmo transforma a consulta em linguagem JavaScript;

3 Extensão de uma álgebra ER para execução de consultas em bancos de dados NoSQL orientados a documentos

Conforme discutido no Capítulo 1, o objetivo deste trabalho foi facilitar o trabalho do Engenheiro de Software no desenvolvimento de operações de manipulação de dados em um cenário com centenas de tecnologias diferentes para armazenamento. Em particular, buscou-se uma forma de possibilitar que o Engenheiro de Software consiga trabalhar em um nível mais alto de abstração no desenvolvimento de consultas, considerando-se um banco de dados orientado a documentos (MongoDB) onde os dados podem ser estruturados de diferentes formas.

A Figura 17 ilustra a solução desenvolvida nesta pesquisa, retomando o cenário da Figura 3 do Capítulo 1. A ideia foi possibilitar que o Engenheiro de Software consiga especificar consultas de forma independente do banco de dados utilizado, e que os resultados obtidos sejam sempre os mesmos, em termos de estrutura, independente do esquema no qual os dados estão armazenados. Dessa forma, ele pode trabalhar nas consultas em um alto nível de abstração, próximo ao modelo conceitual ER.

Conforme já discutido no Capítulo 1, a solução se baseou na álgebra de Parent e Spaccapietra (1984), que define de forma teórica as operações de manipulação de dados sobre entidades e relacionamentos. Nesta pesquisa, foi desenvolvida uma semântica operacional para a operação de consulta da referida álgebra.

A semântica operacional consiste de uma forma concreta de traduzir as operações especificadas na álgebra ER para uma implementação executável. A ideia original era implementar a álgebra completa, no entanto, devido à alta complexidade do problema e às restrições de tempo, foi implementada somente a semântica de consultas de junções entre duas entidades e parte da projeção, para o MongoDB, considerando diferentes possibilidades de mapeamentos.

O resultado do desenvolvimento desta semântica consiste de três partes:

1. Uma gramática livre de contexto (GLC) foi desenvolvida para formalizar a sintaxe concreta da linguagem de consultas, uma vez que a álgebra de Parent e Spaccapietra (1984) não faz nenhuma definição neste sentido. Tal formalização é necessária para viabilizar o processamento das consultas e posterior geração de código;

2. Um conjunto de metamodelos foi desenvolvido para que o Engenheiro de Software

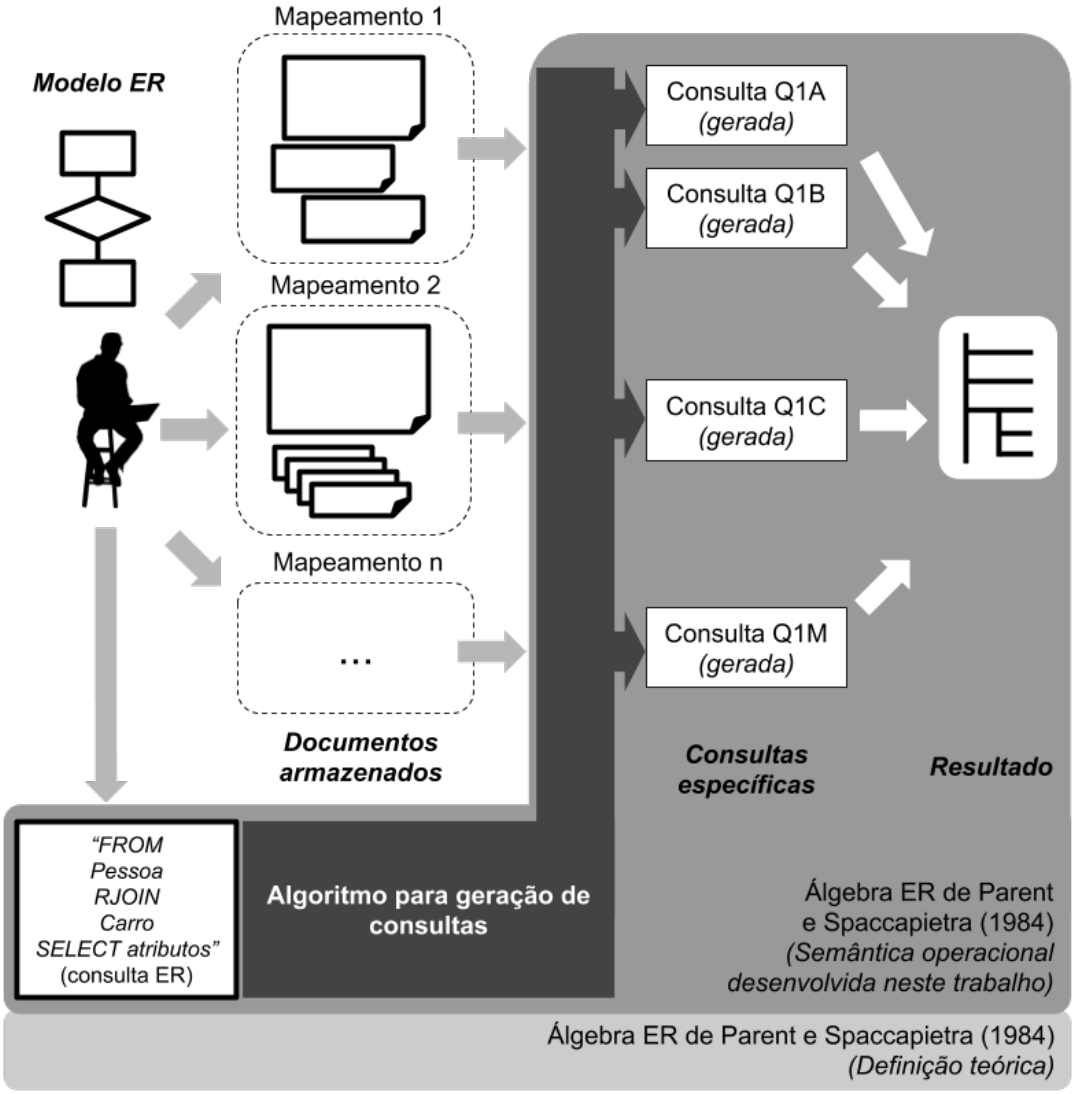

Figura 17 – Visão geral da solução desenvolvida nesta pesquisa.

consiga especificar, formalmente: um modelo conceitual (ER), um modelo que representa o esquema (ESQ) de armazenamento dos documentos no MongoDB, e um modelo intermediário, que conecta os elementos dos outros dois modelos, formalizando assim o mapeamento (MAP) entre entidades e relacionamentos com os documentos armazenados;

3. Foi desenvolvido um algoritmo que lê uma consulta Q, especificada conforme as regras da GLC definida, lê os modelos ER, ESQ e MAP, e produz código JavaScript que executa, no MongoDB, a consulta Q, considerando as especificidades do mapeamento MAP. O algoritmo também cuida para que as consultas geradas produzam resultados sempre consistentes entre si, e com a definição da álgebra de Parent e Spaccapietra (1984). Isso é feito por meio de uma estratégia de implementação que analisa todas as combinações possíveis de documentos mapeados às entidades e rela-

cionamentos. Dessa forma, o Engenheiro de Software tem a garantia que o resultado da consulta será sempre o mesmo, independente do mapeamento MAP.

Nas próximas seções é apresentado um detalhamento sobre cada uma dessas partes desenvolvidas.

3.1 Gramática Livre de Contexto - ERQS (*Entity-Relationship Query Syntax)*

A gramática livre de contexto é utilizada para definir, sintaticamente, a linguagem. Foi criada a partir das cláusulas básicas que compõem as consultas SQL (ELMASRI et al., 2010). Foi utilizado o Formalismo de Backus-Naur Estendido (também conhecido como EBNF) para definir a GLC.

G = (V, $\sum$, P, S)

V = { VALUE, NUMERIC }

$\sum$ = { query, entity, simpleEntity, entityName, entityNickname, rjoin, relationshipName, relationshipNickname, listOfEntities, select, simpleAttribute, listOfAttributes, aggregationFunction, alias, description, where, expressionList, arithmeticExpression, logicalExpression, otherExpression, groupby, having, orderby }

S: query

P: Conjunto de regras de produção, mostradas na Listagem 3.1 em formato compatível com a ferramenta ANTLR[1]. ANTLR é amplamente utilizado para construir linguagens, ferramentas e estruturas. As regras são descritas a seguir.

Nas linhas 1-2 descreve-se expressões regulares para identificadores válidos das variáveis (valor - *VALUE*) e nas linhas 4-5 para números (*NUMERIC*). Na linha 3 é definida uma expressão regular para interpretação apropriada dos espaços em branco (*WHITE_SPACE*) como delimitadores visuais, não geradores de *tokens* (palavra-chave "skip"), dos demais elementos da linguagem.

Continuando com a descrição das regras de produção, na linha 6 da Listagem 3.1, são definidos os elementos da cláusula *query*, que é a regra de produção associada ao símbolo inicial da gramática. Portanto esta é a cláusula principal. São strings terminais 'from', 'select', 'where', 'group by', 'having' e 'order by'; entidade (*entity*) atua como variável.

Nas linhas 7-8, entidade (*entity*) pode ser uma entidade simples (*simpleEntity*) ou uma entidade (*entity*) seguida da variável rjoin string terminal '(', uma lista de entidades

[1] http://www.antlr.org/

(*listOfEntities*) e a string terminal ‘)’.

Na linha 9, entidade simples (*simpleEntity*) é composta por um nome da entidade (*entityName*) e um alias (*entityNickname*) e faz referência a uma única entidade.

Nas linhas 10-12, uma lista de entidades (*listOfEntities*) possui variáveis: entidade (*entity*), na sequência entidade (*entity*) string terminal ‘,’ e lista de entidades (*listOfEntities*) e na sequência entidade string terminal ‘,’ lista de entidades rjoin string terminal ‘(’ entidade e string terminal ‘)’.

Na linha 13, a varíavel rjoin é composta por uma string terminal ‘rjoin’ seguida de um nome do relacionamento (*relationshipName*) e o alias do relacionamento (*relationshipNickname*). A string terminal ‘rjoin’ foi definida para representar junção (*join*) entre entidades.

Nas linhas 14-28, são descritas regras sobre a cláusula *select*: atributos simples (*simpleAttribute*), lista de atributos (*listOfAttributes*) e funcões de agregação (*aggregationFunction*).

Nas linhas 29-53, são definidas regras sobre a cláusula *where*: lista de expressões (*expressionList*), este último incluindo atributos simples (*simpleAttribute*), expressões aritméticas (*arithmeticExpression*), expressões lógicas (*logicalExpression*) e outras expressões (*otherExpression*).

Na linhas 54-57 são descritas regras sobre as cláusulas *groupby*, *having* e *orderby*.

```
1  NAME : (‘a’..‘z’|‘A’..‘Z’)(‘a’..‘z’|‘A’..‘Z’ ‘0’..‘9’|‘_’)*;
2  INTEGER: (‘0’..‘9’)+ ;
3  WHITE_SPACE : (‘ ’ | ‘\t’ | ‘\n’ | ‘\r’ ) -> skip;
4  VALUE: NAME;
5  NUMERIC: INTEGER;
6  query : ‘from’ entity ‘select’ select (‘where’)? (where)? (‘group
       by’)? (groupby)? (‘having’)? (having)? (‘order by’)? (orderby)
     ?;
7  entity : simpleEntity |
8           entity rjoin ‘(’ listOfEntities ‘)’;
9  simpleEntity : entityName=NAME entityNickname=NAME;
10 listOfEntities : entity |
11                  entity ‘,’ listOfEntities |
12                  entity ‘,’ listOfEntities rjoin ‘(’entity‘)’;
13 rjoin: ‘rjoin’ relationshipName = NAME relationshipNickname =
     NAME;
14 select : simpleAttribute |
15          listOfAttributes |
16          aggregationFunction (‘,’ aggregationFunction)* |
```

```
17          listOfAttributes (',' aggregationFunction)* ;
18 simpleAttribute:  entityNickname=NAME '.' attribute=NAME |
19                   relationshipName = NAME '.'
     relationshipNickname = NAME;
20 listOfAttributes:  simpleAttribute |
21                    simpleAttribute ',' listOfAttributes;
22 alias: description=NAME;
23 aggregationFunction: 'avg' '(' simpleAttribute ')' alias? |
24                  'max' '(' simpleAttribute ')' alias? |
25                  'min' '(' simpleAttribute ')'  alias? |
26                  'sum' '(' simpleAttribute ')' alias? |
27                  'count' '(' simpleAttribute ')' alias? |
28                  'count' '(*)' alias? ;
29 where: expressionList;
30 expressionList: simpleAttribute arithmeticExpression VALUE?
                     NUMERIC? (logicalExpression expressionList)* |
31                '(' simpleAttribute otherExpression ')'
32                '(logicalExpression expressionList)* |
33                 otherExpression (logicalExpression
34                 expressionList)* ;
35 arithmeticExpression : '=' |
36                  '<>' |
37                  '>=' |
38                  '<='|
39                  '>' |
40                  '<' |
41                  'like' |
42                  'is not null' |
43                  'is null' ;
44 otherExpression:  'between' NUMERIC 'and' NUMERIC |
45              'not in' '(' query ')' |
46              'not in' '(' NUMERIC (',' NUMERIC)* ')' |
47              'not in' '(' VALUE (',' VALUE)* ')' |
48              'in' '(' query ')' |
49              'in' '(' NUMERIC (',' NUMERIC)* ')' |
50              'in' '(' VALUE (',' VALUE)* ')' |
51              'not exists' '(' query ')' |
52              'exists' '(' query ')' ;
53 logicalExpression: 'and' | 'or';
54 groupby: listOfAttributes;
55 having: aggregationFunction arithmeticExpression NUMERIC |
            expressionList ;
```

```
56 orderby: listOfAttributes 'asc' (',' orderby)* |
57          listOfAttributes 'desc' (',' orderby)*;
```

Listagem 3.1 – Gramática Livre de Contexto ERQS (*Entity-Relationship Query Syntax*)

Na Listagem 3.2 é apresentado um exemplo de consulta especificada na GLC definida. Na linha 1 na cláusula *from* é detalhada a junção (*rjoin drives d*) entre as entidades *person* e *car*; sendo *person* o nome da primeira entidade, *p* o alias, *car* o nome da segunda entidade e *c* o alias. Na linha 2 expõe a cláusula *select* com a lista de atributos a retornar pela consulta, sendo identificadas pelo alias de cada entidade ou relacionamento.

```
1 from person p rjoin drives d (car c)
2 select p.id, p.name, p.address, c.plate, d.observation
```

Listagem 3.2 – Exemplo de consulta especificada na GLC

Na Figura 18 apresenta-se a árvore de análise sintática da consulta exposta na Listagem 3.2. A raiz da árvore é a cláusula *query*.

Foram feitos testes com várias consultas reais para testar a expressividade da gramática. Do ponto de vista sintático, os testes não revelaram nenhum problema, e a gramática parece atender a operação de recuperação de dados prevista na álgebra ER. No entanto, como não foi feita a implementação completa da semântica, pode ser que falte algum elemento sintático referente a essa parte não implementada, e que não foi percebida até o momento.

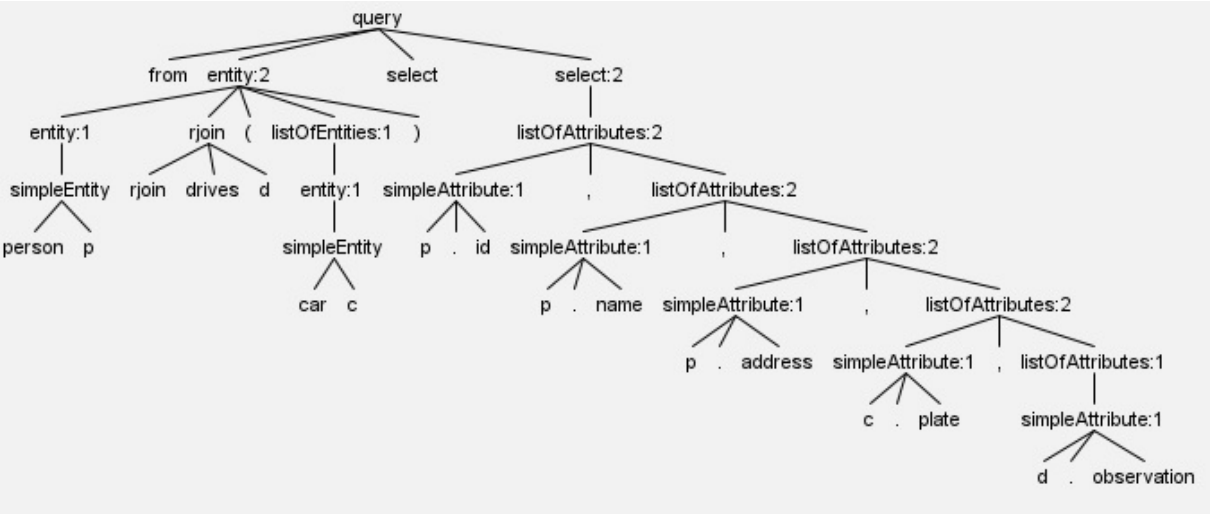

Figura 18 – Árvore de análise sintática

3.2 Metamodelos

Um metamodelo é uma estrutura similar a um diagrama de classes, e possui elementos como classes, atributos, associações e agregações (SILVA et al., 2013). Foram

definidos três metamodelos de modo a capturar os pontos comuns e variáveis do dominio, e definir a estrutura dos modelos que podem ser criados.

Para criação dos metamodelos foi utilizado *Eclipse Modeling Framework* (*EMF*). O *EMF* é um framework integrado ao ambiente de desenvolvimento *Eclipse*, que gera ferramentas baseadas em modelos estruturados para geração de código. Os metamodelos são criados usando a ferramenta *Ecore EMF*, que gera um arquivo XMI com base nas construções *Ecore* (ROSA et al., 2013).

3.2.1 Modelo Entidade-Relacionamento

A Figura 19 apresenta o metamodelo do Modelo Entidade - Relacionamento. Para esta pesquisa foi considerado o Modelo ER clássico do Chen (1976). No metamodelo projetado, um Modelo ER (*ERModel*) é composto por Entidade (*Entity*) e Relacionamento (*Relationship*). Cada *Entity* além do nome (*name*), possui Atributos (*Attribute*) e estes podem ser Compostos (*CompoundAtt*) ou Simples (*SimpleAtt*). Atributos (*Attribute*) possuem nome (*name*), identificador (*identifier*) e indicação se é atributo multivalorado (*multivalued*). O Relacionamento contêm ao menos dois *RelationshipEnd*, que são entidades relacionadas, cada uma com cardinalidade (*Cardinality*) 1 (ONE) ou N (MANY). Isso gera possibilidades de combinações binárias 1-1 (*OneToOne*), 1-N (*OneToMany*), N-1 (*ManyToOne*), N-N (*ManyToMany*) ou outras combinações em casos ternários, quaternários, e outros.

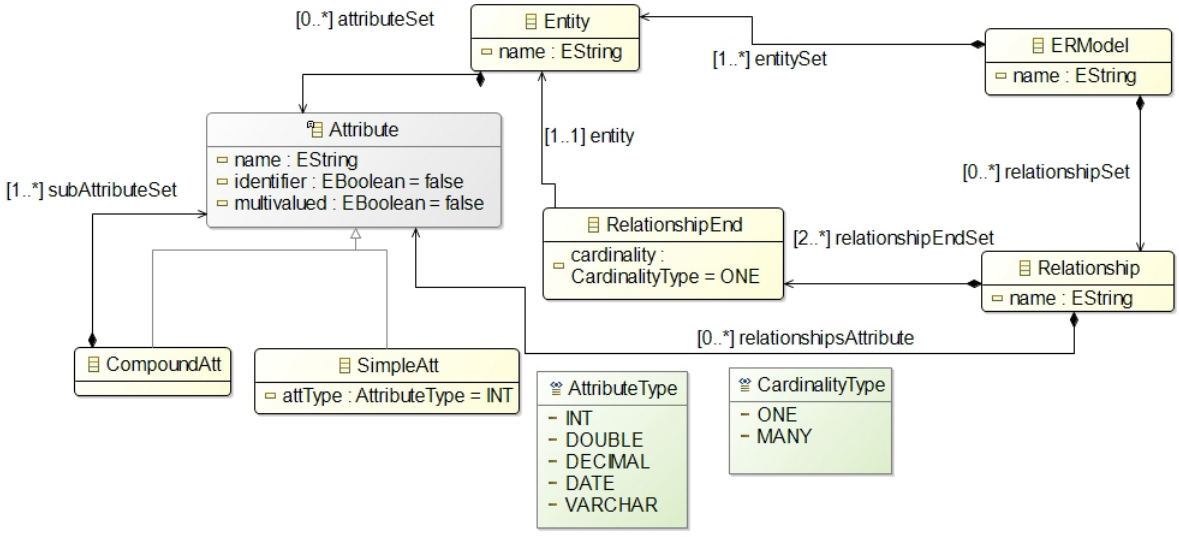

Figura 19 – Metamodelo Modelo Entidade - Relacionamento

3.2.2 Esquema de banco de dados MongoDB

A Figura 20 apresenta o metamodelo do esquema de banco de dados MongoDB. No metamodelo exibido, um esquema MongoDB (*MongoSchema*) contêm um ou mais tipos de documentos (*DocumentType*). Cada *DocumentType*, além do nome (*name*), compõe-se de campos (*Field*), podendo ser do tipo Embutido (*EmbeddedField*) ou Atômico (*AtomicField*). Um campo embutido contêm outros tipos de documentos (*DocumentType*).

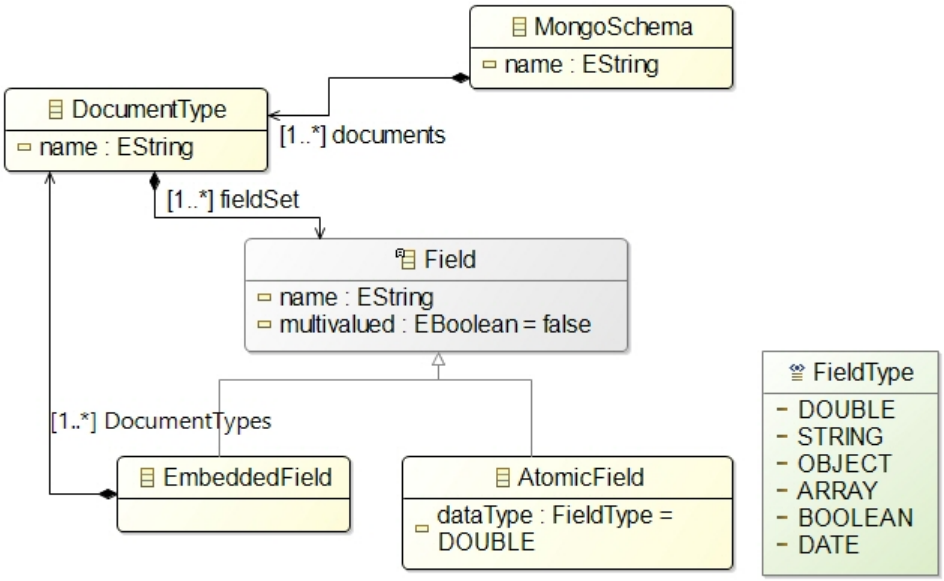

Figura 20 – Metamodelo do esquema de banco de dados MongoDB

3.2.3 ERtoMongoMapping

A Figura 21 expõe o metamodelo intermediário entre os dois primeiros mencionados na Seção 3.2.1 e 3.2.2. Nesta Figura, os elementos dos outros metamodelos aparecem replicados aqui, pois são referenciados pelo metamodelo intermediário. Na Figura, os elementos replicados dos outros metamodelos são representados em cor sombreada, enquanto os elementos do metamodelo intermediário são representados em cor branca. A classe "*Root*" é um elemento novo criado para conter todos os outros, o que é um requisito do EMF. O *ERtoMongoMapping* contêm *EntityMapping* e *RelationshipMapping*. O *EntityMapping* conta com *AttributeMapping*. O Metamodelo do Modelo ER atua de metamodelo origem (*source*), e o Metamodelo do esquema de banco de dados MongoDB atua como destino (*destination*). O objetivo deste metamodelo é possibilitar a criação de modelos que estabelecem um mapeamento entre entidades, relacionamentos, seus atributos (de um modelo ER) e tipos de documentos, campos atômicos e embutidos (de um esquema MongoDB).

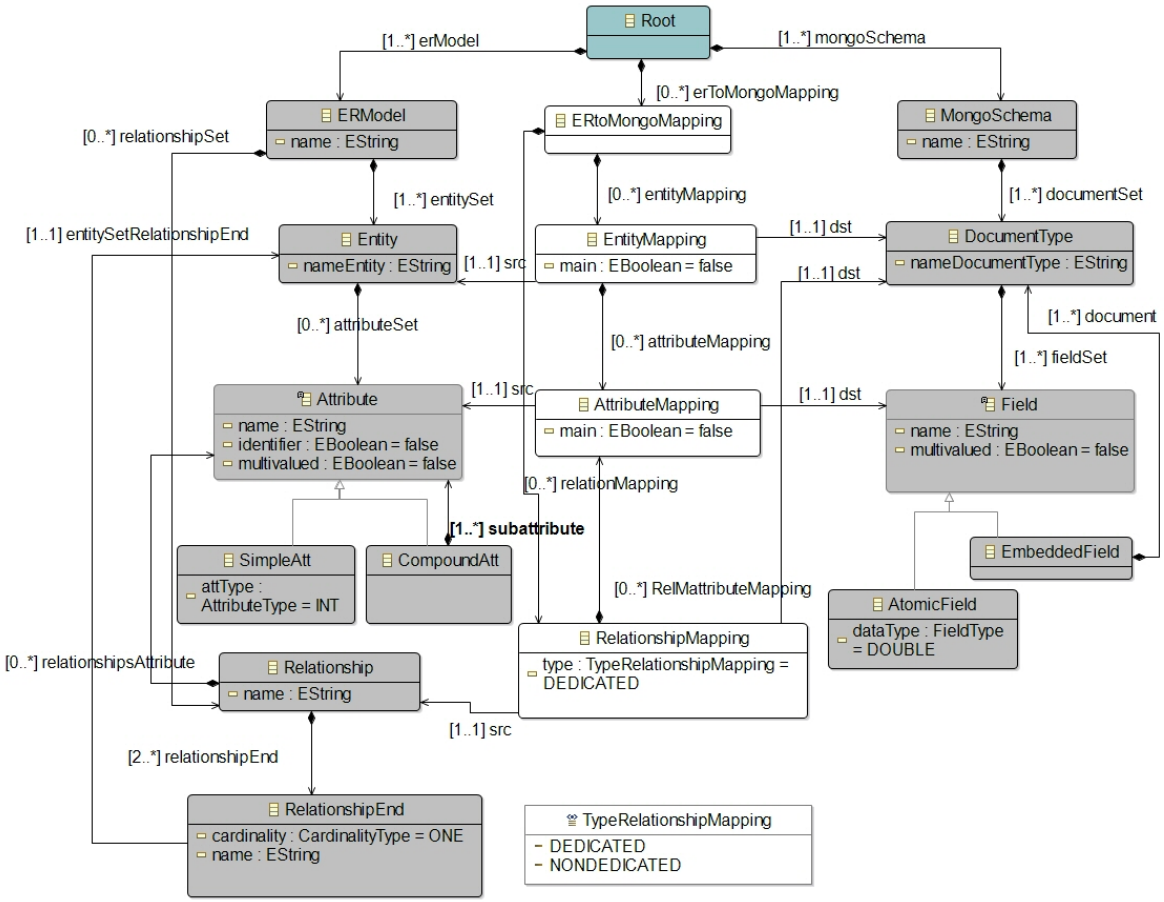

Figura 21 – Metamodelo Intermediario ERtoMongoMapping

As definições a seguir descrevem em mais detalhes como o mapeamento é estabelecido entre elementos do *ERModel* e elementos do *MongoSchema*:

- Existem três tipos de mapeamento: mapeamento de entidade (metaclasse *EntityMapping*), mapeamento de atributo (metaclasse *AttributeMapping*), e mapeamento de relacionamento (metaclasse *RelationshipMapping*);

- Cada mapeamento estabelece uma ligação entre uma e somente uma origem (*src*), que pertence a um modelo ER, e um e somente um destino (*dst*), que pertence a um modelo MongoSchema;

- Um mapeamento de entidade estabelece uma relação entre uma entidade (*Entity*) e um tipo de documento (*DocumentType*).

- Um mapeamento de atributo estabelece uma relação entre um atributo (*Attribute*), que pode ser simples ou composto, e um campo (*Field*), que pode ser atômico ou embutido.

- Um mapeamento de relacionamento estabelece uma relação entre um relacionamento (*Relationship*) e um tipo de documento (*DocumentType*). Isso permite que sejam identificados os locais onde estão mapeados os atributos das entidades relacionadas e do próprio relacionamento.

- Mapeamentos de entidades e relacionamentos podem ser marcados como principais (*main=true*) ou não (*main=false*).

- O mapeamento com *main=true* significa que:

  - Todos os atributos da entidade ou relacionamento estão presentes no tipo de documento (*DocumentType*).
  - Todas as instâncias da entidade estarão armazenadas no *DocumentType*, sem repetição, ou seja, uma consulta do tipo "*findAll*" neste *DocumentType* irá retornar todas as instâncias possíveis da entidade, com todos os atributos possíveis.

- O mapeamento com *main=false* significa que pelo menos uma das seguintes situações é verdadeira:

  - O *DocumentType* não tem todos os atributos da entidade/relacionamento mapeado.
  - Uma consulta do tipo "*findAll*" neste *DocumentType* retorna instâncias duplicadas.
  - Uma consulta do tipo "*findAll*" neste *DocumentType* não retorna todas as instâncias possíveis da entidade/relacionamento.

Além das definições acima, que constam na própria definição do metamodelo, as seguintes restrições acrescentam critérios adicionais que precisam ser respeitados no mapeamento.

- Se uma entidade/relacionamento está mapeada a um tipo de documento, este deve possuir pelo menos um campo mapeado a um atributo desta entidade/relacionamento.

- Se um atributo está mapeado a um campo, este deve pertencer a um tipo de documento mapeado à entidade/relacionamento que contêm o atributo.

- Um campo de um tipo de documento pode ser simples ou embutir um outro tipo de documento.

- Cada tipo de documento deve ter unicamente um campo identificador.

- Se a Entidade no *ERModel* tiver cardinalidade N em um relacionamento, os atributos da Entidade podem ser mapeados num array de campos (*ArrayField*). Este *ArrayField* terá um nome e os campos simples mapeados aos atributos da Entidade.

- Se a Entidade no *ERModel* tiver cardinalidade N em um relacionamento, os atributos pertencentes ao Relacionamento, serão mapeados como campos do *DocumentType* mapeado a Entidade do lado N, mas sempre referenciado ao relacionamento.

Todos os metamodelos construídos foram testados por meio da criação manual, dentro da ferramenta EMF, de vários casos de teste de sistemas reais, envolvendo modelos ER, esquemas MongoDB diversos e mapeamentos considerados válidos. Os testes serviram para refinar e corrigir erros, até um ponto em que os metamodelos foram considerados adequados para a sequência da pesquisa.

Após os testes, o EMF não foi mais diretamente utilizado, exceto como uma referência. Explica-se: a longo prazo, pretende-se utilizar o próprio EMF como mecanismo de criação e gerenciamento de modelos, pois é este seu principal propósito. Pretende-se criar sintaxes visuais completas, sobre os metamodelos EMF, para que o Engenheiro de Software possa criar e manipular modelos visualmente ou textualmente. No entanto, para isso é necessário um estudo mais aprofundado sobre tecnologias de linguagens, tais como Eclipse GMP[2] e Xtext[3], que atuam junto ao EMF. Também é necessário um trabalho de implementação referente à construção das ferramentas de modelagem.

Assim, para esta pesquisa, optou-se por uma implementação mais rápida, para que se pudesse dar início à construção do algoritmo de geração das consultas. Por esse motivo, foi criada uma representação desses metamodelos manualmente na linguagem Java. Não é o ideal, mas com isso pode-se criar rapidamente um arcabouço de testes para o algoritmo.

A Listagem 3.3 mostra um trecho de código com exemplo de modelos especificados diretamente na linguagem Java, para os três metamodelos. As linhas 1-16 contêm definições do modelo ER, com duas entidades (*person* e *car*) e um relacionamento (*drives*) do tipo 1-N. As linhas 18-30 contêm a definição do esquema MongoDB, com um único tipo de documento (*docTypePerson*). Nesta implementação, os mapeamentos aparecem embutidos dentro dos tipos de documentos e campos (*ERMapping* e *FieldMapping* nas linhas 21-26). Nas linhas 27-28 são mapeados o identificador da entidade *car* e o atributo do relacionamento *drives* dentro de um *array*, o qual contêm todos os carros que a pessoa dirige. As linhas 32-37 implementam uma chamada à verificação das restrições descritas nesta seção.

```
1 ERModel erModel = new ERModel();
2
```

[2] http://www.eclipse.org/modeling/gmp/
[3] https://www.eclipse.org/Xtext/

```
Entity person = new Entity("Person");
person.addAttribute(new Attribute(person, "id", "int", true));
person.addAttribute(new Attribute(person, "name", "string", false
    ));
person.addAttribute(new Attribute(person, "address", "string",
    false));
erModel.addERElement(person);

Entity car = new Entity("Car");
// atributos de car ...
erModel.addERElement(car);

Relationship drives = new Relationship("Drives");
drives.addRelationshipEnd(new RelationshipEnd(person, Cardinality
    .One));
drives.addRelationshipEnd(new RelationshipEnd(car, Cardinality.
    Many));
erModel.addERElement(drives);

MongoSchema mongoSchema = new MongoSchema();

DocumentType docTypePerson = new DocumentType("DocTypePerson");
docTypePerson.addERMapping(new ERMapping(person, true));
docTypePerson.addERMapping(new ERMapping(car, false));
docTypePerson.addERMapping(new ERMapping(drives, true));
docTypePerson.addField(new SimpleField(docTypePerson, "_id", "int
    ", new FieldMapping(person.getAttribute("id"))));
docTypePerson.addField(new SimpleField(docTypePerson, "fName", "
    string", new FieldMapping(person.getAttribute("name"))));
docTypePerson.addField(new SimpleField(docTypePerson, "fAddress",
     "string", new FieldMapping(person.getAttribute("address"))));
docTypePerson.addArrayField(new ArrayField ("data_Car", new
    SimpleField(docTypeCar, "fCarId", "int", new FieldMapping(car.
    getAttribute("id")))));
docTypePerson.addArrayField(new ArrayField ("data_Car", new
    SimpleField(docTypeCar, "fObservation", "string", new
    FieldMapping(drives.getAttribute("observation")))));

mongoSchema.addDocumentType(docTypePerson);

List<String> violations = mongoSchema.validate();
if(violations.size() > 0) {
```

```
34      for(String v:violations) {
35          System.out.println(v);
36      }
37  }
```

Listagem 3.3 – Exemplo de modelos criados diretamente em uma estrutura Java

3.3 Estrutura do resultado da consulta

Para facilitar o entendimento do algoritmo, primeiro apresenta-se a estrutura do resultado das consultas a serem geradas.

Tomando como base a Álgebra de Entidade - Relacionamento definida por Parent e Spaccapietra (1984), foi determinada a estrutura do resultado das consultas, de forma a imitar a estrutura teórica definida na álgebra utilizando documentos JSON do MongoDB.

Seguindo o exemplo do Modelo ER definido na Listagem 3.4, uma vez executada a consulta em JavaScript, o resultado tem a estrutura mostrada na Listagem 3.5.

```
1  Person{
2     id : int,
3     name : string,
4     address : string
5  }
6  Car{
7     id : int,
8     plate : string,
9     color : string
10 }
11 Drives (Person, Car) 1:N{
12   observation: string
13 }
```

Listagem 3.4 – Modelo ER, cardinalidade 1-N

O primeiro campo do resultado sempre será um *id* gerado pelo MongoDB (linha 2), o segundo campo, um agrupamento de campos complexos, obrigatórios e monovalorados todos campos da Entidade 1. Este será exibido com o nome "data_" seguido do nome da Entidade 1 (linha 3) e entre chaves são recuperados os campos.

Em seguida é determinado o terceiro campo "data_Join" de tipo *array* (linha 8), que consiste num agrupamento multivalorado, opcional, de campos complexos, obrigatório e monovalorado. Onde cada um desses campos agrupa respectivamente todos os campos do Relacionamento e Entidade 2. Caso não existam campos do Relacionamento e Entidade 2, é inserido o campo "data_Join" com um *array* vazio.

Entende-se por campos complexos aqueles campos definidos por um agrupamento de campos simples ou compostos.

```
{
    "_id" : ObjectId("5a904e68e3c0b516ba3ffb5d"),
    "data_Person" : {
        "_id" : 1.0,
        "fName" : "João",
        "fAddress" : "Alameda dos Crisantemos"
    },
    "data_Join" : [
        {
            "data_Car" : {
                "fCarId" : 101.0,
                "fPlate" : "ABC-1234",
                "fColor" : "Azul"
            },
            "data_Drives" : {
                "fObservation" : "Trabalho"
            }
        },
        {
            "data_Car" : {
                "fCarId" : 103.0,
                "fPlate" : "DEF-5678",
                "fColor" : "Vermelho"
            },
            "data_Drives" : {
                "fObservation" : "Lazer"
            }
        }
    ]
}
```

Listagem 3.5 – Estrutura do resultado da consulta

3.4 Algoritmo de conversão de consultas

Este algoritmo foi implementado na linguagem Java, considerando os 3 metamodelos criados. Ele é capaz de construir a consulta em JavaScript, levando em consideração todas as possíveis combinações de formas de mapeamento do Modelo ER para o esquema

MongoDB. Em outras palavras, ele gera a consulta de maneira a recuperar os dados desde todos os documentos onde foram mapeadas as entidades, relacionamentos e atributos.

No inicio do trabalho, foi descrita uma GLC para ajudar na definição sintática da linguagem. Pretendia-se definir a semântica da linguagem levando em consideração:

- Recuperação de dados de uma entidade;
- Recuperação de dados a partir da junção de duas ou mais entidades;
- Uso de expressões condicionais que identificam os atributos a serem recuperados pela consulta;
- Uso de funções de agregação para resumir informações de várias entidades em uma síntese, tais como `count`, `sum`, `max`, `min`, `avg` na linguagem SQL;
- Uso de consultas aninhadas. Algumas consultas precisam que os valores existentes no banco de dados sejam buscados e depois usados em uma condição de comparação posterior. Tais consultas podem ser formuladas convenientemente usando consultas aninhadas (*subqueries*);
- Uso de cláusulas: `group by`, `having`, `order by`.

No entanto, foi possível implementar um dos items mencionados. A construção da semântica da linguagem foi focada em parte da operação de projeção (*select*) e a operação de junção (*join*) entre duas entidades. Não foi implementado o primeiro item, o mais simples, recuperação de dados de uma entidade, porque consideramos ser trivial. Assim, partimos da junção de duas entidades, onde encontramos dificuldade devido aos diferentes mapeamentos possíveis causados pela desnormalização que é comum em bancos não relacionais. Os demais não foram implementados por falta de tempo, mas são consideravelmente mais simples do que a junção.

Primeiramente, o desenvolvedor deverá definir o Modelo *ERModel* e o Modelo *MongoSchema* que consiste no mapeamento das Entidades/Relacionamento do Modelo ER para o esquema MongoDB, considerando os critérios definidos na Seção 3.2.3; em seguida, especificar a consulta, que deverá apresentar a junção de duas entidades, e por fim a lista de atributos a serem selecionados (*queryAttributes*).

Como já explicado, na prática, para a execução do algoritmo, é preciso criar os modelos diretamente em Java, como ilustrado na Listagem 3.3. Porém, para o texto e as explicações que se seguem, considerou-se que o excesso de código seria pouco legível. Assim, foi empregada uma representação textual mais simplificada e direta, na esperança de facilitar a compreensão dos modelos pelo leitor. Na Listagem 3.6 é apresentado um exemplo do Modelo de Entidade Relacionamento nesta representação.

```
1  Person {
2      id: int,
3      name: string,
4      address: string
5  }
6  DriversLicense {
7      id: int,
8      number: string,
9      date: date
10 }
11 Registration (Person, DriversLicense) 1:1 {
12     observation: string
13 }
```

Listagem 3.6 – Modelo ER em representação textual criada especificamente para esta dissertação. Para a execução do algoritmo, é necessário criar código Java que recria esta estrutura.

Inicialmente é indicado o nome da Entidade. Os atributos são descritos entre chaves e separados por vírgula; se listam os atributos seguido de dois pontos especificando o tipo. A Entidade *Person* (linhas 1-5) tem como atributo identificador *id* do tipo *int* (linha 2), além dos atributos *name* do tipo *string* (linha 3) e *address* do tipo *string* (linha 4). A Entidade *DriversLicense* (linhas 6-10) tem como atributo identificador *id* do tipo *int* (linha 7), além dos atributos *number* do tipo *string* (linha 8) e *date* do tipo *date* (linha 9). O relacionamento entre *Person* e *DriversLicense* nomeado *Registration* (linha 11), de cardinalidade 1:1 possui como atributo *observation* do tipo *string* (linha 12).

A Listagem 3.7 expõe uma opção de mapeamento do Modelo ER para o esquema MongoDB, utilizando os critérios descritos na Seção 3.2.3. Nas linhas 1-6 é definido o tipo de documento *DocTypePerson*, mapeado à entidade *Person*, representado entre colchetes ([ ]) o nome da entidade seguida do (*main=true*) que indica que o documento é o principal e todos os atributos estão presentes em todas as instâncias. Lembrando de que *main* pode ter valores *true* e *false*. Neste caso, o documento é o principal, o que significa que ele é a fonte principal das consultas por esta entidade. Considere uma consulta do tipo "Encontrar todas as pessoas". Se ela for executada neste tipo de documento, há a garantia de que todas as pessoas existentes, com todos os seus atributos, serão encontradas. Nas linhas 3-5 são descritos os campos deste tipo de documento, e os mapeamentos aos atributos das entidades aparecem entre colchetes, seguindo a notação "Nome da entidade"."nome do atributo".

Nas linhas 7-12 é definido o *DocTypeDriversLicense* (mapeado a entidade *DriversLicense*), com os campos mapeados aos atributos (linha 9-11). Este documento é o

principal (*main=true*) e todos os atributos estão presentes.

```
1  DocTypePerson [ Person(main=true) ]
2  {
3      _id      : int        [ Person.id ]
4      fName    : string     [ Person.name ]
5      fAddress: string      [ Person.address ]
6  }
7  DocTypeDriversLicense [ DriversLicense(main=true) ]
8  {
9      _id      : int        [ DriversLicense.id ]
10     fNumber  : string     [ DriversLicense.number ]
11     fDate    : date       [ DriversLicense.date ]
12 }
13 DocTypeRegistration [ Registration(main=true), DriversLicense(
       main=false), Person(main=false) ]
14 {
15     _id                 : int      [ ]
16     observation         : string   [ Registration.observation ]
17     fPersonId           : int      [ Person.id ]
18     fName               : string   [ Person.name ]
19     fAddress            : string   [ Person.address ]
20     fDriversLicenseId   : int      [ DriversLicense.id ]
21     fNumber             : int      [ DriversLicense.number ]
22 }
```

Listagem 3.7 – Mapeamento do Modelo ER para o esquema MongoDB em representação textual criada especificamente para esta dissertação. Para a execução do algoritmo, é necessário criar código Java que recria esta estrutura.

Nas linhas 13-22 é definido o *DocTypeRegistration*. Este mapeamento é mais complexo, pois está simultaneamente ligado ao relacionamento *Registration*, e também às entidades *Person* e *DriversLicense*. É especificado um campo *id* identificador do relacionamento, além dos campos referentes à entidade *Person* e alguns campos de *DriversLicense*. Nota-se que o mapeamento é principal (*main=true*) apenas para o relacionamento *Registration*. Isto significa que este tipo de documento é onde estarão todas as instâncias, completas, dos dados de *Registration*. *Person* e *DriversLicense* não são o mapeamento principal. Isto significa que este tipo de documento não é a fonte principal de armazenamento dos dados dessas entidades. Considere uma consulta do tipo "Encontrar todas as pessoas". Se executada neste tipo de documento, pode ser que algumas pessoas não sejam encontradas (por não ter registro). Essas pessoas estariam armazenadas no *DocTypePerson* (por ser o mapeamento principal) mas não aqui.

Além disso, entidades mapeadas como não-principais (main=false) não necessariamente tem todos os seus atributos mapeados. Neste exemplo de *DocTypeRegistration*, o atributo *date* da entidade *DriversLicense* não se encontra. Com isso, uma consulta executada neste tipo de documento e que procure pelo atributo *date* necessariamente teria que realizar uma junção com o tipo de documento principal da entidade *DriversLicense*. Isso pode ser feito com base nos campos identificadores *fPersonId* e *fDriversLicenseId*, que atuam como referência.

Uma vez especificados os modelos ER, MongoSchema e de mapeamento, é necessário especificar a consulta. A seguir mostra-se um exemplo da consulta: *"from Person p rjoin Registration r (DriversLicense d) select p.name, p.address, r.observation, d.number"*.

A tradução automática da consulta da GLC para o algoritmo ainda não foi implementada, portanto para esta pesquisa realizou-se a implementação manual. No entanto, não é uma tarefa muito complexa, exigindo apenas trabalho de codificação dentro do compilador da GLC.

A Figura 22 mostra a consulta implementada em Java. A linha 21 corresponde à criação dos modelos. A linha 22 apenas imprime os modelos, para conferência. As linhas 24-27 correspondem à obtenção, através do nome, de uma referência Java para as entidades e relacionamentos a serem consultadas, neste caso, "Person", "Registration" e "DriversLicense". As linhas 29-33 correspondem à lista de atributos a serem projetados na consulta (*queryAttributes*). Na linha 35, é feita a chamada à função principal *binaryJoin*.

```
20  public static void main(String[] args) {
21      MappingModel mm = ModelSample1a.getModel();
22      System.out.println(mm.toString());
23
24      MainAlgorithm ma = new MainAlgorithm(mm);
25      Entity person = (Entity) mm.getERModel().findERElement("Person");
26      Relationship registration = (Relationship) mm.getERModel().findERElement("Registration");
27      Entity driversLicense = (Entity) mm.getERModel().findERElement("DriversLicense");
28
29      List<Attribute> queryAttributes = new ArrayList<>();
30      queryAttributes.add(person.getAttribute("name"));
31      queryAttributes.add(person.getAttribute("address"));
32      queryAttributes.add(registration.getAttribute("observation"));
33      queryAttributes.add(driversLicense.getAttribute("number"));
34
35      List<Query> queries = ma.binaryJoin(person, registration, driversLicense, queryAttributes);
```

Figura 22 – Lista queryAttributes, implementado na linguagem Java

Na Listagem 3.8 é apresentado o algoritmo *Query Generator*. O algoritmo utiliza o conceito de Entidade Computada, descrito por Parent e Spaccapietra (1984), como uma entidade que compila os atributos das entidades envolvidas na junção. Aqui, a entidade computada é representada como um documento temporário no MongoDB.

É importante mencionar que o algoritmo pode gerar mais de uma consulta para uma mesma consulta de entrada, visto que ele gera todas as possibilidades de consultas

distintas. Assim, para uma mesma consulta de entrada Q, ele pode gerar N consultas em JavaScript, uma diferente da outra. Uma vez que o algoritmo analisa todas as combinações possíveis de junção dos *DocumentTypes*. O algoritmo descarta aquelas consultas que são repetidas. Caso a consulta não tem resultado garantido, ou são impossíveis, o algoritmo informa com uma mensagem ao momento de gerar a consulta.

Como entrada (linhas 1-2) são requeridos os seguintes elementos: Modelo *ERModel*, Modelo *MongoSchema*, Entidade E1, Entidade E2, Relacionamento R e *queryAttributes*. Nas linhas 4-14 são descritos os códigos com os passos que são executados para verificar a existência de relacionamento e mapeamentos. Caso exista Relacionamento com atributos (linha 15-16), é executado o algoritmo apresentado na Listagem 3.9, caso não exista (linha 17-19), é executado o algoritmo apresentado na Listagem 3.14.

```
1  Entrada: Modelo ERModel, Modelo MongoSchema, Entidade E1,
       Entidade E2, Relacionamento R, queryAttributes
2  Saida: Consulta escritas em JavaScript
3  inicio
4      Verificar existencia de relacionamento R entre E1 e E2
5      Identificar mapeamentos do ERModel no MongoSchema
6      Se existe mapeamento para E1 então
7          Salvar em Lista de docTypesE1
8      fim
9      Se existe mapeamento para R então
10         Salvar em Lista de docTypesR
11     fim
12     Se existe mapeamento para E2 então
13         Salvar em Lista de docTypesE2
14     fim
15     Se existe Lista de docTypesR então
16         Executa Algoritmo para Relacionamento com atributos
17     senão
18         Executa Algoritmo para Relacionamento sem atributos
19     fim
20 fim
```

Listagem 3.8 – Algoritmo *Query Generator*

A Figura 23 representa um fragmento do algoritmo principal implementado na linguagem Java. Nas linhas (40-49) verifica-se a existencia de relacionamento entre Entidade 1 e Entidade 2. Nas linhas (50-52) identifica-se os mapeamentos das entidades e relacionamento do *ERModel* para o *MongoSchema*.

Na Listagem 3.9 é apresentado o algoritmo a ser executado se existe relacionamento com atributos mapeados. Isto é, existe um Relacionamento R, que foi mapeado a um tipo

```java
public class MainAlgorithm {

    MappingModel mappingModel;

    public MainAlgorithm(MappingModel mappingModel) {
        this.mappingModel = mappingModel;
    }

    public List<Query> binaryJoin(Entity e1, Relationship r, Entity e2, List<Attribute> queryAttributes) {
        List<Query> ret = new ArrayList<>();

        if (!(e1 == r.getRelationshipEnds().get(0).getEntity()
                && e2 == r.getRelationshipEnds().get(1).getEntity())
                || (e1 == r.getRelationshipEnds().get(1).getEntity()
                && e2 == r.getRelationshipEnds().get(0).getEntity())) {
            throw new RuntimeException("As entidades do join ("
                    + e1.getName() + ", " + e2.getName() + ")"
                    + " não estão relacionadas por meio de "
                    + " " + r.getName()
            );
        }
        List<DocumentType> docTypesE1 = mappingModel.getMongoSchema().findDocumentTypes(e1);
        List<DocumentType> docTypesR = mappingModel.getMongoSchema().findDocumentTypes(r);
        List<DocumentType> docTypesE2 = mappingModel.getMongoSchema().findDocumentTypes(e2);
```

Figura 23 – Fragmento do Algoritmo principal, implementado na linguagem Java

de documento, com os atributos mapeados aos campos do referido documento.

Como entrada (linhas 1-2) são requeridos os seguintes elementos: lista de *docTypesE1*, lista de *docTypesR*, lista de *docTypesE2*, Entidade E1, Entidade E2, Relacionamento R e *queryAttributes*. Nas linhas (4-6), são percorridas as listas de *DocumentTypes* que contêm os mapeamentos da Entidade 1 (*docTypesE1*), Relacionamento (*docTypesR*) e Entidade 2 (*docTypesE2*).

Na linha 7 da Listagem 3.9, a função *findDocTypeOperation* tem como objetivo encontrar os atributos da Entidade 1 mapeados no *docTypesE1*. Se o *DocumentType* tiver mapeado *main=false*, o algoritmo emite uma advertencia: "A consulta pode não retornar todos os dados, a Entidade 1: Nome Entidade 1 não é principal no documento: *DocType* Nome documento".

Nesta função é gerada uma nova *Entidade Computada* (novo documento com nome *Entidade Computada*) que conterá como primeiro campo do documento, os campos da Entidade 1 mapeados no *docTypesE1*, e o campo 'data_Join' que, caso existam atributos da Entidade 2 e do Relacionamento solicitados no *queryAttributes*, são adicionados em forma agrupada dentro do campo 'data_Join'. Caso o *docTypesE1* só conter atributos da Entidade 1, é adicionado um *array* vazio com nome do campo 'data_Join'. No final da geração do resultado da consulta o campo 'data_Join' incluirá os campos do Relacionamento e da Entidade 2.

```
1 Entrada: Lista de docTypesE1, Lista de docTypesR, Lista de
     docTypesE2,  Entidade E1, Entidade E2, Relacionamento R,
     queryAttributes
2 Salida: Consultas escritas em JavaScript
3 inicio
```

```
4  Para i de docTypesE1[0] ate docTypesE1[N] faca
5   Para j de docTypesR[0] ate docTypesR[N] faca
6    Para x de docTypesE2[0] ate docTypesE2[N] faca
7     Executar findDocTypeOperation (queryAttributes, i, R)
8     Executar completeAttributesOperations(queryAttributes, E1, R)
9
10    Executar joinTwoEntitiesOperation(queryAttributes, E1, R, j)
11    Executar completeAttributesOperations(queryAttributes, R , R)
12
13    Executar joinTwoEntitiesOperation(queryAttributes, E2, R, x)
14    Executar completeAttributesOperations(queryAttributes, E2, R)
15   fimpara
16  fimpara
17 fimpara
18 fim
```

Listagem 3.9 – Algoritmo para Relacionamento com atributos

Levando em consideração o exemplo de mapeamento do Modelo ER para o esquema MongoDB descrito na Listagem 3.7, a função *findDocTypeOperation* gera parcialmente a consulta em JavaScript mostrada na Listagem 3.10. Na Listagem, na linha 1 é accesado no *DocTypePerson* pelo método *find*(), seguido do método *forEach* que itera o cursor para aplicar uma função JavaScript a cada documento a partir do cursor. É inserido um novo *DocumentType* EC, sigla da *Entidade Computada* (linha 2) com os campos da Entidade 1 (linhas 3-7). Como no *DocTypePerson* só existem atributos da Entidade 1, é inserido o campo "data_Join" com um *array* vazío (linha 8). Na linha 9 é encerrado o método *forEach*.

```
1 db.DocTypePerson.find().forEach(function(data){
2    db.EC.insert( {
3       data_Person: {
4          _id: data._id,
5          fName: data.fName,
6          fAddress: data.fAddress,
7       },
8       data_Join: []});
9 });
```

Listagem 3.10 – Consulta inicial em JavaScript

Na linha 8 da Listagem 3.9, a função *completeAttributesOperations* tem como objetivo verificar a existencia de campos da Entidade 2 e do Relacionamento no *docTypesE1*. Se existe atributo de consulta (*queryAttributes*) que ainda não esteja incluido na *Entidade Computada*, procurasse um *DocumentType* principal (*main=true*), verifica-se se existem

campos identificadores comuns entre a *Entidade Computada* e o *DocumentType*, caso existam, é realizada a junção entre documentos e são recuperados e completados os atributos requeridos dentro da *Entidade Computada*, sempre tendo em conta a cardinalidade das entidades de maneira a gerar as consultas na estrutura correta.

Na linha 10 da Listagem 3.9, a função *joinTwoEntitiesOperation* executa a junção entre a *Entidade Computada* e um *DocumentType* principal (*main=true*) da lista de *docTypesR*. Verifica-se se existem campos identificadores comuns entre a *Entidade Computada* e o *DocumentType*, caso existam, é realizada a junção entre documentos e são recuperados e completados os atributos necessários dentro da *Entidade Computada*.

Tanto para operação *completeAttributesOperations* como *joinTwoEntitiesOperation* caso não exista *DocumentType* com (*main=true*), o algoritmo emite uma advertencia: "impossível realizar operação join entre EC e *DocumentType*, pois não há atributos identificadores comuns entre eles".

Levando em consideração o exemplo de mapeamento do modelo ER para o esquema MongoDB descrito na Listagem 3.7, a função *joinTwoEntitiesOperation* adiciona a consulta em JavaScript mostrada na Listagem 3.11 a Listagem 3.10.

Na Listagem 3.11, primeiramente é feito o acesso na *EC* pelo método *find*, em seguida com o método *forEach* percorre-se o documento (linha 2). Posteriormente é realizada uma cópia dos dados existentes no campo "*data_Join*" para mais adiante conseguir sobre escrevê-los (linhas 3-7). A continuação é realizada a junção entre a EC e o *DocTypeRegistration* pelos atributos identificadores (linha 8). Assim, é possível recuperar os dados da Entidade 2 e do Relacionamento (linhas 9-16), e atualizarlos na EC existente (linhas 17-21).

```
1  //add Listagem 3.10
2  db.EC.find().forEach( function(data){
3  var varData = [];
4   data.data_Join.forEach(
5       function(dataCopy) {
6            varData.push(dataCopy);
7       });
8   db.DocTypeRegistration.find({ 'fPersonId': data.data_Person._id
      }).forEach(function(data2) {
9      varData.push( {
10         data_DriversLicense: {
11          fDriversLicenseId: data2.fDriversLicenseId,
12         },
13         data_Registration: {
14          observation: data2.observation,
15       }})
```

```
16      });
17      db.EC.update( {'data_Person._id': data.data_Person._id},
18      { $set: {
19           'data_Join': varData
20      }});
21  });
```

Listagem 3.11 – Consulta gerada depois da execução da função joinTwoEntitiesOperation

As linhas 10-14 da Listagem 3.9 seguem a mesma logica descrita anteriormente, mas levando em consideração, atributos do Relacionamento e da Entidade 2.

A execução da função *completeAttributesOperations* para Entidade 2, gera a consulta apresentada na Listagem 3.12. A consulta JavaScript começa acessando a *EC* pelo método *find*. Após, com o método *forEach* é aplicada uma função JavaScript a cada documento (linha 2). É criada uma variável *novoArray* de tipo *array* que será utilizada para conter os atributos da Entidade 2 (linha 3). Posteriormente, com o método *forEach* é percorrido o campo "data_Join" (linha 4) e por cada elemento é realizada a junção com *DocTypeDriversLicense* (linha 5), para assim recuperar os dados (linhas 6-7) e atualizá-los na *Entidade Computada* (linhas 8-12).

```
1  //add Listagem 3.11
2  db.EC.find().forEach( function(data){
3  var novoArray = [];
4  data.data_Join.forEach(function(data2) {
5      data2.data_DriversLicense = db.DocTypeDriversLicense.findOne
       ({ '_id': data2.data_DriversLicense.fDriversLicenseId });
6          novoArray.push(data2);
7      });
8      db.EC.update( {'data_Person._id': data.data_Person._id},
9      { $set: {
10     'data_Join': novoArray
11     }});
12 });
```

Listagem 3.12 – Consulta gerada depois da execução da função completeAttributesOperations para Entidade 2

O resultado da consulta gerada para o mapeamento 3.7 é apresentada na Listagem 3.13.

```
1  {
2      "_id" : ObjectId("5a87656249cc2ca1169add3a"),
3      "data_Person" : {
4          "fPersonId" : 1.0,
```

```
        "fName" : "João",
        "fAddress" : "Alameda dos Crisantemos"
    },
    "data_Join" : [
        {
            "data_Registration" : {
                "observation" : "Registrado em São Carlos - SP"
            },
            "data_DriversLicense" : {
                "_id" : 100.0,
                "fNumber" : "26726",
                "fDate" : "07/10/2017"
            }
        }
    ]
}
```

Listagem 3.13 – Resultado da consulta sobre o mapeamento 3.7

Na Listagem 3.14 é apresentado o algoritmo a ser executado se não existe relacionamento com atributos mapeado. Isto é, existe um Relacionamento R, que não possui atributos. O algoritmo segue a mesma lógica do algoritmo apresentado na Listagem 3.9 sem considerar os *DocumentTypes* do relacionamento *docTypesR*. Exemplos de mapeamentos onde é executado este algoritmo, são apresentados no Estudo de Caso: Modelo ER PrograduWeb, na Subseção 4.2.1.

```
Entrada: Lista de docTypesE1, Lista de docTypesR, Lista de
   docTypesE2,  Entidade E1, Entidade E2, Relacionamento R,
   queryAttributes
Salida: Consultas escritas em JavaScript
inicio
 Para i de docTypesE1[0] ate docTypesE1[N] faca
   Para x de docTypesE2[0] ate docTypesE2[N] faca
    Executar findDocTypeOperation (queryAttributes, i, R)
    Executar completeAttributesOperations(queryAttributes, E1, R)

    Executar joinTwoEntitiesOperation(queryAttributes, E2, R, x)
    Executar completeAttributesOperations(queryAttributes, E2, R)
   fimpara
 fimpara
fim
```

Listagem 3.14 – Algoritmo para Relacionamento sem atributos

A seguir é apresentado outro exemplo para compreender melhor os passos do algoritmo para geração das consultas em JavaScript. Com este exemplo percebe-se que a execução de duas consultas diferentes em JavaScript para uma mesma consulta de entrada geram o mesmo resultado, em termos de estrutura. A Listagem 3.15 expõe uma opção de mapeamento do Modelo ER representado na Listagem 3.4 para o esquema MongoDB, utilizando os critérios descritos na Seção 3.2.3. Nas linhas 1-6 é definido o tipo de documento *DocTypePerson*, mapeado à entidade *Person*, seguida do (*main=true*) que indica que o documento é o principal e todos os atributos estão presentes em todas as instâncias. Nas linhas 7-16 a entidade *Car* e o relacionamento *Drives* são mapeados para o tipo de documento *DocTypeCar* com *main=true*. A entidade *Person* também é mapeada ao tipo de documento *DocTypeCar* com *main=false*. Considere uma consulta do tipo "Encontrar todas as pessoas com os carros que ela dirige". Se ela for executada no tipo de documento *DocTypeCar*, não há a garantia de que todas as pessoas existentes, com todos os carros que ela dirige, serão encontradas. Nas linhas 9-15 são descritos os campos deste tipo de documento.

```
1  DocTypePerson [ Person (main=true)]
2  {
3     _id      : int [ Person.id ]
4     fName    : string [ Person.name ]
5     fAddress : string [ Person.address ]
6  }
7  DocTypeCar [ Car(main=true), Person(main=false), Drives (main=
      true)]]
8  {
9     _id          : int [ Car.id ]
10    fPlate       : string [ Car.plate ]
11    fColor       : string [ Car.color ]
12    fPersonId    : int [ Person.id ]
13    fName        : string [ Person.name ]
14    fAddress     : string [ Person.address ]
15    fObservation : string [Drives.observation]
16 }
```

Listagem 3.15 – Mapeamento do Modelo ER representado na Listagem 3.4 para o esquema MongoDB, opção OneToMany1a

A seguir mostra-se um exemplo da consulta considerada para execução do algoritmo sobre o mapeamento da Listagem 3.15: *"from Person p rjoin Drives d (Car c) select p.name, p.address, d.observation, c.plate, c.color"*.

A execução do algoritmo para o mapeamento da Listagem 3.15, retornam as consultas apresentadas na Listagem 3.16 e 3.17. Estas consultas não são explicadas em detalhes,

pois seguem o mesmo raciocínio apresentado anteriormente.

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: []
    });
});
db.EC.find().forEach( function(data){
     var varData = [];
     var varData2 = [];
     data.data_Join.forEach(
     function(dataCopy) {
          varData.push(dataCopy);
     });
     db.DocTypeCar.find({'fPersonId': data.data_Person._id }).
   forEach(
     function(data2) {
     varData.push( {
        data_Drives: {
                fObservation: data2.fObservation
        },
        data_Car: {
                _id: data2._id,
                fPlate: data2.fPlate,
                fColor: data2.fColor
        }
    });
});
db.EC.update({'data_Person._id': data.data_Person._id},
{ $set: {
     'data_Join': varData
    } } );
 });
});
```

Listagem 3.16 – Consulta em JavaScript, Mapeamento 1-N, opção OneToMany1a

```
db.DocTypeCar.find().forEach( function(data) {
   db.EC.insert( {
```

```
3       data_Person: {
4          fPersonId: data.fPersonId,
5       },
6       data_Join: []
7     });
8     db.EC.createIndex({'data_Person.fPersonId': 1 }, {unique:
    true } );
9     db.EC.update( {'data_Person.fPersonId':data.fPersonId},
10    {$addToSet: {
11        'data_Join': {
12            data_Drives: {
13                fObservation: data.fObservation
14      },
15            data_Car: {
16                _id: data._id,
17                fPlate: data.fPlate,
18                fColor: data.fColor
19      }
20    }
21  }});
22 });
23 db.EC.find().forEach( function(data){
24    var varData = [];
25    db.DocTypePerson.find({'_id': data.data_Person.fPersonId }).
    forEach(
26      function(data2) {
27      varData.push( {
28         data_Person: {
29          fPersonId: data2._id,
30          fName: data2.fName,
31          fAddress: data2.fAddress,
32       }})
33    }) ;
34 db.EC.updateMany( {'data_Person.fPersonId': data.data_Person.
    fPersonId},
35 { $set: {
36    'data_Person': varData[0].data_Person
37 }})
38 });
```

Listagem 3.17 – Consulta em JavaScript, Mapeamento 1-N, opção OneToMany1b

O resultado da consulta da Listagem 3.16 é apresentada na Listagem 3.18. Já o

resultado da consulta da Listagem 3.17 é apresentada na Listagem 3.19. Percebe-se que a estrutura é a mesma, independente da consulta executada, exceto pelo *id* (linha 2) gerado pelo MongoDB e o nome do atributo da Entidade *Person* (linha 4).

```
1  {
2      "_id" : ObjectId("5acce3e3d08aa5f0ab6e8433"),
3      "data_Person" : {
4          "_id" : 1.0,
5          "fName" : "João",
6          "fAddress" : "Alameda dos Crisantemos"
7      },
8      "data_Join" : [
9          {
10             "data_Drives" : {
11                 "fObservation" : "Trabalho"
12             },
13             "data_Car" : {
14                 "_id" : 101.0,
15                 "fPlate" : "ABC-1234",
16                 "fColor" : "Azul"
17             }
18         },
19         {
20             "data_Drives" : {
21                 "fObservation" : "Lazer"
22             },
23             "data_Car" : {
24                 "_id" : 103.0,
25                 "fPlate" : "DEF-5678",
26                 "fColor" : "Vermelho"
27             }
28         }
29     ]
30 }
```

Listagem 3.18 – Resultado da execução da consulta apresentada na Listagem 3.16

```
1  {
2      "_id" : ObjectId("5acce3e3d08aa5f0ab6e8433"),
3      "data_Person" : {
4          "fPersonId" : 1.0,
5          "fName" : "João",
6          "fAddress" : "Alameda dos Crisantemos"
7      },
```

```
    "data_Join" : [
        {
            "data_Drives" : {
                "fObservation" : "Trabalho"
            },
            "data_Car" : {
                "_id" : 101.0,
                "fPlate" : "ABC-1234",
                "fColor" : "Azul"
            }
        },
        {
            "data_Drives" : {
                "fObservation" : "Lazer"
            },
            "data_Car" : {
                "_id" : 103.0,
                "fPlate" : "DEF-5678",
                "fColor" : "Vermelho"
            }
        }
    ]
}
```

Listagem 3.19 – Resultado da execução da consulta apresentada na Listagem 3.17

Assim, é construída a consulta em base a combinação de todas as possíveis formas de mapeamento do Modelo ER para o esquema MongoDB. Todas as consultas geradas em JavaScript dentro de um mesmo tipo de cardinalidade (1-1, 1-N, N-1, N-N), independente do tipo de mapeamento, retornam a mesma estrutura do resultado da consulta. Excepcionalmente, em alguns casos pode variar só a ordem dos campos da Entidade 2 e do Relacionamento dentro do campo 'data_Join' da *Entidade Computada*.

O algoritmo completo, implementado em Java, está disponibilizado gratuitamente no site *GitHub*[4].

Como se pode perceber, o algoritmo apenas implementa a junção de duas entidades, e apenas parte da projeção.

A projeção não foi implementada completamente, pois os atributos a serem selecionados são utilizados na entrada do algoritmo, e são considerados na decisão sobre quais junções realizar. Porém, as consultas geradas em JavaScript contêm tanto os atributos selecionados como os outros. Ou seja, o algoritmo não desconsidera os atributos que de-

[4] https://github.com/vromero23/query-generator

veriam ser suprimidos. Por esse motivo considera-se que a implementação da projeção foi parcial.

3.5 Considerações finais

Como apresentado durante este capítulo, foi construida uma gramática livre de contexto para definir a sintaxe concreta da linguagem. A construção da semântica da linguagem foi focada em parte da operação de projeção e a operação de junção (*join*) entre duas entidades.

Alem disso, foram tratadas as diferentes formas de mapeamento do Modelo ER para o Modelo MongoDB através de uma abordagem MDE. Mais especificamente, foram desenvolvidos metamodelos, um Metamodelo do Modelo Entidade - Relacionamento, um Metamodelo de esquema MongoDB e um terceiro Metamodelo *ERtoMongoMapping* que atua de intermediário entre os dois primeiros mencionados. Assim, mediante um algoritmo, logrou-se construir consultas de junção entre duas entidades na linguagem JavaScript, levando em consideração todas as possíveis formas de mapeamento. Isto é, são geradas consultas para recuperar os dados de todos os possíveis documentos onde foi mapeado. É importante destacar, que todas as consultas dentro de um mesmo tipo de cardinalidade (1-1, 1-N, N-1, N-N), retornam a mesma estrutura do resultado da consulta.

Algumas restrições na implementação precisaram ser adotadas, devido à falta de tempo:

- não foi feita a tradução automática da GLC para código que executa o algoritmo de geração;
- não foi utilizado o arcabouço EMF para criação e gerenciamento dos modelos;
- não foi implementada a semântica completa da álgebra ER para consultas.

No entanto, tais restrições são menos relevantes do que o que foi alcançado, e podem ser resolvidas em trabalhos futuros.

Por fim, para testar o funcionamento da abordagem desenvolvida, foram realizados alguns estudos. Os mesmos estão detalhadamente explicados no próximo capítulo.

4 Estudos de caso

Dois estudos de caso, envolvendo dois Modelagens ER, são apresentados para avaliar a abordagem desenvolvida. O primeiro Modelo ER é apresentado na Seção 4.1, o mesmo foi definido no trabalho de Parent e Spaccapietra (1984) e foi estendida de forma a proporcionar uma semântica operacional focada no modelo de documentos conforme implementado pelo MongoDB. Apresenta-se na seção 4.2 o segundo Modelo ER, referente a um sistema de controle acadêmico que foi utilizado por muito tempo na UFSCar, denominado PrograduWeb.

Antes de iniciar a discussão sobre a abordagem de cada tipo de representação de modelagem conceitual, é importante destacar alguns conceitos básicos sobre documentos e relacionamentos em um banco de dados orientado a documentos (VERA W. BOAVENTURA, 2015):

- Um documento descreve um conjunto de atributos que possuem suas propriedades organizadas em uma estrutura de chave-valor.
- As informações contidas em um documento são descritas pelo identificador (chave) e o valor associado à chave.
- Diferentes tipos de relações entre documentos são definidos como referências (*References*) e documentos embutidos (*Embedded Documents*).
- Como o NoSQL é um banco de dados não-relacional, os conceitos de normalização, não se aplicam.
- Alguns conceitos de relações entre documentos são semelhantes à modelagem ER, como cardinalidade (*one-to-one*, *one-to-many*, *many-to-many*).

Uma modelagem conceitual simples pode gerar diversas alternativas de representação em uma modelagem física. O algoritmo leva em consideração todas as possíveis combinações de formas de mapeamento entre duas entidades e um relacionamento, do Modelo ER para o esquema MongoDB. Os bancos de dados não relacionais orientado a documentos, permitem representar documentos embutidos em vários níveis, mas para este trabalho foi limitada ao mapeamento de entidade/relacionamento embutidos só no primeiro nível.

Para esses estudos de caso, estavam disponíveis apenas os modelos ER de exemplos, e não os dados. Assim, para possibilitar os testes, foram populados os esquemas do banco de dados orientado a documentos MongoDB com dados gerados através de uma

ferramenta gratuita. É importante mencionar que os dados foram adaptados na mão para as diferentes formas de mapeamento, não sendo utilizada nenhuma ferramenta para auxiliar este processo. Para implementação do algoritmo foi utilizada a linguagem Java e para execução das consultas escritas em JavaScript, foi utilizada *Robomongo*, uma interface gráfica livre com *shell* embutido.

4.1 Estudo de caso: Modelo ER de Parent e Spaccapietra (1984)

Como mencionado anteriormente, o Modelo ER apresentado nesta Seção foi definido pelos autores Parent e Spaccapietra (1984). Foram adicionados alguns atributos nas Entidades/Relacionamentos de maneira a realizar os testes de todas as possíveis formas de mapeamento do Modelo ER para o esquema MongoDB. Na listagem 4.1 observa-se as seguintes entidades: *Person*, *DriversLicense*, *Car*, *Garage* e *InsuranceCompany* e os seguintes relacionamentos: *Registration*, *Drives* e *Repairs*.

Por cada Modelo ER, foram testadas 28 formas de mapeamento para entidades com cardinalidade 1-1. Para entidades com cardinalidade 1-N, foram testadas 6 formas de mapeamento, mesma quantidade para entidades com cardinalidade N-1. Finalmente para entidades com cardinalidade N-N foram testadas duas formas de mapeamento. Nesta Seção são apresentados alguns resultados. Mais resultados são mostrados nos Apêndices A e B.

```
1  Person {
2      id      : int,
3      name    : string,
4      address : string
5  }
6  DriversLicense {
7      id     : int,
8      number: int,
9      date   : date
10 }
11 Car {
12     id    : int,
13     plate: string,
14     color: string
15 }
16 Garage {
17     id      : int,
18     name    : string,
19     address : string
20 }
```

```
Registration (Person, DriversLicense) 1:1 {
    observation: string
}
Drives (Person, Car) 1:N {
    observation: string
}
Repairs (Car, Garage) N:N {
    date: date,
    fix: string
}
```

Listagem 4.1 – Modelo Entidade - Relacionamento extraído do estudo de caso de Parent e Spaccapietra (1984)

4.1.1 Mapeamento com cardinalidade 1-1

Na Listagem 4.2 é apresentada uma das opções de mapeamento para entidades com cardinalidade 1-1. Neste mapeamento, o valor da entidade 1 é *Person*, da entidade 2 é *DriversLicense* e do relacionamento é *Registration*. Foram mapeados documentos principais *DocTypePerson* e *DocTypeDriversLicense* para as entidades. Dentro do *DocTypeRegistration* foram embutidos os atributos das entidades *Person* dentro do campo *data_Person* e *DriversLicense* dentro do campo *data_DriversLicense*, ambos mapeados com *main=false*.

A seguir mostra-se a consulta de entrada para este exemplo: *"from Person p rjoin Registration r (DriversLicense d) select p.name, p.address, r.observation, d.number"*.

```
DocTypePerson [ Person(main=true) ]
{
    _id      : int        [ Person. id ]
    fName    : string     [ Person.name ]
    fAddress: string      [ Person.address ]
}
DocTypeDriversLicense [ DriversLicense(main=true) ]
{
    _id      : int        [ DriversLicense.id ]
    fNumber  : string     [ DriversLicense.number ]
    fDate    : date       [ DriversLicense.date ]
}
DocTypeRegistration [ Registration(main=true), Person(main=false)
   , DriversLicense(main=false) ]
{
    _id             : int        [ ]
    observation     : string     [ Registration.observation ]
```

```
17      data_Person     :
18      {
19          fPersonId   : int        [ Person.id ]
20          fName       : string     [ Person.name ]
21          fAddress    : string     [ Person.address ]
22      }
23      data_DriversLicense :
24      {
25          fDriversLicenseId: int [ DriversLicense.id ]
26      }
27  }
```

Listagem 4.2 – Esquema MongoDB, Mapeamento 1-1

A partir do mapeamento no esquema MongoDB mostrada na Listagem 4.2 e da consulta de entrada, é executado o algoritmo, tendo como resultado as consultas em JavaScript apresentadas na Listagem 4.3 e 4.4.

Na Listagem 4.3 o primeiro acesso é feito no tipo de documento *DocTypePerson* recuperando os dados da entidade 1 *Person* e inserido na *EC* (linhas 1-7) e criado o campo "data_Join" como um *array* vazio (linha 8). Logo é feita a junção entre *EC* e *DocTypeRegistration* para recuperar e atualizar os dados da entidade 2 *DriversLicense* e do relacionamento *Registration* (linhas 10-32). Em seguida, como ainda faltam dados da entidade 2, é completada e atualizada realizando a junção entre *EC* e *DocTypeDriversLicense* (linhas 33-43).

```
1  db.DocTypePerson.find().forEach(function(data){
2   db.EC.insert({
3      data_Person: {
4          _id: data._id,
5          fName: data.fName,
6          fAddress: data.fAddress
7      },
8      data_Join: []});
9  });
10 db.EC.find().forEach(function(data){
11  var varData = [];
12  data.data_Join.forEach(
13   function(dataCopy) {
14      varData.push(dataCopy);
15   });
16   db.DocTypeRegistration.find({'data_Person.fPersonId': data.
      data_Person._id}).forEach(
17     function(data2){
```

```
18      varData.push({
19          data_DriversLicense: {
20              fDriversLicenseId: data2.data_DriversLicense.
    fDriversLicenseId
21          },
22          data_Registration: {
23              observation: data2.observation
24          },
25      })
26      });
27    db.EC.update({'data_Person._id': data.data_Person._id},
28    {
29      $set: {
30          'data_Join': varData
31    }});
32  });
33  db.EC.find().forEach(function(data){
34       var novoArray = [];
35       data.data_Join.forEach(function(data2) {
36          data2.data_DriversLicense = db.DocTypeDriversLicense.
    findOne({'_id': data2.data_DriversLicense.fDriversLicenseId});
37          novoArray.push(data2);
38   });
39   db.EC.update({'data_Person._id': data.data_Person._id},
40   { $set: {
41          'data_Join': novoArray
42   }});
43  });
```

Listagem 4.3 – Consulta em JavaScript, Mapeamento 1-1, Query 1

Na Listagem 4.4 o primeiro acesso é feito no *DocTypeRegistration* recuperando os dados da entidade 1 *Person* e inserido na *EC* (linhas 1-5), e da entidade 2 *DriversLicense* e do relacionamento *Registration* (linhas 6-15). Depois, como faltam dados da entidade 1, é completada e atualizada realizando a junção entre *EC* e *DocTypePerson* (linhas 16-31). Em seguida, é feita a junção entre *EC* e *DocTypeDriversLicense* para completar e atualizar os dados da entidade 2 (linhas 32-43). O algoritmo expõe a seguinte advertência: "A consulta pode não retornar todos os dados, a Entidade 1: *Person* não é principal no documento: *DocTypeRegistration*".

```
1  db.DocTypeRegistration.find().forEach(function(data){
2   db.EC.insert({
3      data_Person: {
```

```
4           fPersonId: data.data_Person.fPersonId
5       },
6       data_Join: [{
7           data_Registration: {
8               observation: data.observation
9           },
10          data_DriversLicense: {
11              fDriversLicenseId: data.data_DriversLicense.
    fDriversLicenseId
12          }
13          }]
14      });
15  });
16  db.EC.find().forEach(function(data){
17   var varData = [];
18   db.DocTypePerson.find({'_id': data.data_Person.fPersonId}).
      forEach(function(data2) {
19      varData.push({
20        data_Person: {
21            fPersonId: data2._id,
22            fName: data2.fName,
23            fAddress: data2.fAddress
24      }})
25   });
26   db.EC.update({'data_Person.fPersonId': data.data_Person.
      fPersonId},
27   {
28      $set: {
29          'data_Person': varData[0].data_Person
30      }})
31   });
32  db.EC.find().forEach( function(data){
33    var novoArray = [];
34    data.data_Join.forEach(function(data2) {
35      data2.data_DriversLicense = db.DocTypeDriversLicense.findOne
      ({'_id': data2.data_DriversLicense.fDriversLicenseId });
36            novoArray.push(data2);
37    });
38    db.EC.update({'data_Person.fPersonId': data.data_Person.
       fPersonId},
39    {
40      $set: {
```

```
41          'data_Join': novoArray
42      }});
43  });
```

Listagem 4.4 – Consulta em JavaScript, Mapeamento 1-1, Query 2

4.1.2 Mapeamento com cardinalidade 1-N

Na Listagem 4.5 é apresentada uma das opções de mapeamento para entidades com cardinalidade 1-N. No tipo de documento *DocTypePerson* foram embutidos dentro de um *array* os atributos da entidade *Car* e do relacionamento *Drives*. O nome do campo de tipo *array* que contêm estes atributos, obrigatoriamente devem ter o nome "data_" seguida do nome da entidade que será embutida, ver linha 6 da Listagem 4.5. Para todas as opções de mapeamento para entidades com cardinalidade 1-N ou N-1, os atributos do relacionamento são mapeados para atributos da entidade no lado N, seguindo a modelagem de dados usando o modelo ER descrito por Elmasri et al. (2010). Neste mapeamento, o valor da entidade 1 é *Person*, entidade 2 *Car* e do relacionamento é *Drives*.

```
1  DocTypePerson [Person(main=true), Car(main=false), Drives(main=
       true)]
2  {
3     _id       : int [ Person.id ]
4     fName     : string [ Person.name ]
5     fAddress  : string [ Person.address ]
6     data_Car:
7    [
8      {
9         fCardId      : int [ Car.id ]
10        fObservation: string [Drives.observation]
11     }
12   ]
13 }
14 DocTypeCar [ Car(main=true) ]
15 {
16    _id     : int [ Car.id ]
17    fPlate  : string [ Car.plate ]
18    fColor  : string [ Car.color ]
19 }
```

Listagem 4.5 – Esquema MongoDB, Mapeamento 1-N

A seguir mostra-se a consulta de entrada para este exemplo: *"from Person p rjoin Drives d (Car c) select p.name, p.address, d.observation, c.plate, c.color"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.5 e da consulta de entrada, é executado o algoritmo, tendo como resultado a consulta em JavaScript apresentada na Listagem 4.6. O acesso é feito ao tipo de documento *DocTypePerson*, onde é recuperado os atributos da entidade 1 (linhas 1-7), em seguida com o método *forEach* é percorrido o campo *data_Car* e atualizado os atributos do relacionamento e da entidade 2 (linhas 10-24). Posterior, como ainda faltam dados da entidade 2, é completada e atualizada realizando a junção entre *EC* e *DocTypeCar* (linhas 25-45).

```
1  db.DocTypePerson.find().forEach(function(data){
2    db.EC.insert({
3      data_Person:{
4          _id: data._id,
5          fName: data.fName,
6          fAddress: data.fAddress
7      },
8      data_Join: []
9    });
10   data.data_Car.forEach(function(data1){
11     db.EC.update({'data_Person._id':data._id},
12     {
13       $addToSet:{
14            'data_Join':{
15              data_Drives:{
16                 fObservation: data1.fObservation
17              },
18              data_Car: {
19                 fCarId: data1.fCarId
20              }
21             }}
22         });
23     });
24 });
25 db.EC.find().forEach( function(data){
26   var varData = [];
27   data.data_Join.forEach(function(data1){
28    db.DocTypeCar.find({'_id': data1.data_Car.fCarId }).forEach(
     function(data2) {
29      varData.push({
30         data_Car: {
31            fCarId: data2._id,
32            fPlate: data2.fPlate,
33            fColor: data2.fColor
```

```
        },
          data_Drives: {
            fObservation: data1.data_Drives.fObservation,
         }
     })
   }) ;
  });
  db.EC.update({'data_Person._id': data.data_Person._id},
  { $set: {
        'data_Join': varData
  }});
});
```

Listagem 4.6 – Consulta em JavaScript, Mapeamento 1-N

4.1.3 Mapeamento com cardinalidade N-1

Na Listagem 4.7 é apresentada uma das opções de mapeamento para entidades com cardinalidade N-1. Neste mapeamento o campo *fPersonId* atua de referência dentro do tipo de documento *DocTypeCar*, para caso precisar, realizar a junção com o tipo de documento *DocTypePerson*. Neste mapeamento, o valor da entidade 1 é *Car*, entidade 2 *Person* e do relacionamento é *Drives*.

```
DocTypeCar[Car(main=true), Person(main=false), Drives(main=true)]
{
   _id          : int [ Car.id ]
   fPlate       : string [ Car.plate ]
   fColor       : string [ Car.color ]
   fObservation : string [Drives.observation]
   fPersonId    : int [ Person.id ]
   fName        : string [ Person.name ]
   fAddress     : string [ Person.address ]
}
DocTypePerson [ Person (main=true)]
{
   _id      : int [ Person.id ]
   fName    : string [ Person.name ]
   fAddress : string [ Person.address ]
}
```

Listagem 4.7 – Esquema MongoDB, Mapeamento N-1

A seguir mostra-se a consulta de entrada para este exemplo: *"from Car c rjoin Drives d (Person p) select c.plate, c.color, p.name, p.address, d.observation"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.7 e da consulta de entrada, é executado o algoritmo, tendo como resultado a consulta em JavaScript apresentada na Listagem 4.8. O acesso é feito ao tipo de documento *DocTypeCar*, onde é recuperado e inserido na *EC*, os atributos da entidade 1 (linhas 1-7), e os atributos do relacionamento e da entidade 2 (linhas 8-19).

```
1  db.DocTypeCar.find().forEach( function(data) {
2   db.EC.insert( {
3    data_Car: {
4       _id: data._id,
5       fPlate: data.fPlate,
6       fColor: data.fColor
7    },
8    data_Join: [{
9          data_Drives: {
10             fObservation: data.fObservation
11         },
12         data_Person: {
13             fPersonId: data.fPersonId,
14             fName: data.fName,
15             fAddress: data.fAddress
16         }
17     }]
18  });
19 });
```

Listagem 4.8 – Consulta em JavaScript, Mapeamento N-1

4.1.4 Mapeamento com cardinalidade N-N

Na Listagem 4.9 é apresentada uma das opções de mapeamento para entidades com cardinalidade N-N. Um relacionamento N:N no modelo ER é um exemplo de uma relação entre dois tipos de entidades, em que ambos podem ter muitos relacionamentos entre entidades. Um exemplo pode ser um Livro que foi escrito por muitos Autores. Ao mesmo tempo, um Autor pode ter escrito muitos Livros. As relações N:N são modeladas no banco de dados relacional usando uma tabela de junção que contêm as chaves primárias dos originais, cada uma representando uma chave externa e duas relações 1:N (STANESCU; BREZOVAN; BURDESCU, 2016).

No MongoDB podemos representar esta situação de várias maneiras. O primeiro caminho é chamado de incorporação de duas vias, "*Two Way Embedding*" (TRIVEDI, 2014). Na Listagem 4.9 apresenta-se a incorporação de duas vias. Foi incluído no tipo de documento *DocTypeGarage* campos do tipo de documento *DocTypeCar*, além do atributo

do relacionamento *Repairs*. Dentro de *DocTypeCar* foi incluído campos do *DocTypeGarage*, além do atributo do relacionamento *Repairs*.

Outra maneira de modelar os relacionamentos N:N é chamado de incorporação de uma via, "*One Way Embedding*" (TRIVEDI, 2014). A estratégia de incorporação de uma via escolhe otimizar o desempenho de leitura de uma relação N:N embutindo as referências de um lado do relacionamento (STANESCU; BREZOVAN; BURDESCU, 2016). Exemplo deste tipo de mapeamento é apresentado na Seção 4.2.4.

```
DocTypeCar [ Car(main=true), Garage(main=false), Repairs(main=
   true)]
{
    _id    : int [ Car.id ]
    fPlate : string [ Car.plate ]
    fColor : string [ Car.color ]
    data_Garage:
    [
        {
            fGarageId : int [ Garage.id ]
            fName     : string [Garage.name]
            fAddress  : string [Garage.address]
            fDate     : date [ Repairs.date ]
            fFix      : string [ Repairs.fix ]
        }
    ]
}
DocTypeGarage [ Garage(main=true), Car(main=false), Repairs(main=
   true)]
{
    id      : int [ Garage.id ]
    fName   : string [Garage.name]
    fAddress: string [Garage.address]
    data_Car:
    [
        {
            fCarId : int [ Car.id ]
            fPlate : string [ Car.plate ]
            fColor : string [ Car.color ]
            fDate  : date [ Repairs.date ]
            fFix   : string [ Repairs.fix ]
        }
    ]
}
```

```
33 DocTypeRepairs [Repairs(main=true)]
34 {
35     fDate: date
36     fFix: string
37 }
```

Listagem 4.9 – Esquema MongoDB, Mapeamento N-N

Neste mapeamento, o valor da entidade 1 é *Car*, entidade 2 *Garage* e do relacionamento é *Repairs*. A seguir mostra-se a consulta de entrada para este exemplo: *"from Car c rjoin Repairs r (Garage g) select c.plate, c.color, r.date, r.fix, g.name, g.address"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.9 e da consulta de entrada, é executado o algoritmo, tendo como resultados as consultas em JavaScript apresentadas na Listagem 4.10 e 4.11. Na Listagem 4.10 o acesso é feito ao tipo de documento *DocTypeCar*, onde é recuperado e inserido na *EC*, os atributos da entidade 1 (linhas 1-10) e atualizado os atributos do relacionamento e da entidade 2 (linhas 12-27). Na linha 11 é criado um índice, de maneira a evitar a inserção de dois Carros com mesmo *id*.

```
1  db.DocTypeCar.find().forEach( function(data) {
2    data.data_Garage.forEach(function(data1){
3      db.EC.insert({
4        data_Car: {
5          _id: data._id,
6          fPlate: data.fPlate,
7          fColor: data.fColor
8        },
9        data_Join: []
10     });
11     db.EC.createIndex({'data_Car._id': 1 }, {unique: true } );
12     db.EC.update({'data_Car._id':data._id},
13     { $addToSet: {
14         'data_Join': {
15           data_Repairs: {
16             fDate: data1.fDate,
17             fFix: data1.fFix
18           },
19           data_Garage: {
20             fGarageId: data1.fGarageId,
21             fName: data1.fName,
22             fAddress: data1.fAddress
23           }
24         }
```

```
25      }});
26     });
27 });
```

Listagem 4.10 – Consulta em JavaScript, Mapeamento N-N, Query 1

Na Listagem 4.11 o acesso é feito ao tipo de documento *DocTypeGarage*, onde a partir do método *forEach* que percorre o campo "data_Car", é recuperado e inserido na *EC*, os atributos da entidade 1 (linhas 1-10) e atualizado os atributos do relacionamento e da entidade 2 (linhas 12-27). Na linha 11 é criado um índice, de maneira e evitar a inserção de dois Carros com mesmo *fCarId*.

```
1  db.DocTypeGarage.find().forEach(function(data){
2    data.data_Car.forEach(function(data1){
3      db.EC.insert({
4        data_Car: {
5          fCarId: data1.fCarId,
6          fPlate: data1.fPlate,
7          fColor: data1.fColor
8        },
9        data_Join: []
10     });
11     db.EC.createIndex({'data_Car.fCarId': 1 }, {unique: true });
12     db.EC.update({'data_Car.fCarId':data1.fCarId},
13     {$addToSet: {
14         'data_Join': {
15            data_Repairs: {
16              fDate: data1.fDate,
17              fFix: data1.fFix
18            },
19            data_Garage: {
20              _id: data._id,
21              fName: data.fName,
22              fAddress: data.fAddress
23            }
24         }
25      }});
26     });
27 });
```

Listagem 4.11 – Consulta em JavaScript, Mapeamento N-N, Query 2

4.2 Estudo de caso: Modelo ER PrograduWeb

Apresenta-se o segundo Modelo ER, referente a um sistema de controle acadêmico que foi utilizado por muito tempo na UFSCar, denominado PrograduWeb. O Modelo ER original é muito maior do que o apresentado aqui. Mas para testar a abordagem proposta, apenas algumas Entidades/Relacionamentos, com seus respetivos atributos, são suficientes, até porque a semântica completa das consultas não foi implementada.

Na listagem 4.12 observa-se as seguintes entidades: *Alunograd*, *Matriculagrad*, *Disciplinagrad*, *Enfasegrad* e os seguintes relacionamentos: *Mora*, *Registra* e *Gradegrad*.

```
1  Alunograd{
2     codalu_alug   : int,
3     nomealu_alug  : string,
4     datanasc_alug: string,
5     cpf_alug      : string
6  }
7  Matriculagrad{
8     codalu_matr  : int,
9     codenf_matr  : int,
10    anoini_matr  : string,
11    semini_matr  : string
12 }
13 Disciplinagrad{
14    coddiscip_discip : string,
15    nome_discip      : string,
16    setor_discip     : string
17 }
18 Enfasegrad{
19    codenf_enf    : int,
20    siglaenf_enf  : string,
21    nomeenf_enf   : string
22 }
23 Endereco{
24     cod_endereco  : int,
25     nome_cidade   : string,
26     nome_bairro   : sgring,
27     cep           : string,
28     nome_rua      : string,
29     complemento   : string
30 }
31 Mora (Alunograd, Endereco) 1:1{
32 }
```

```
Registra (Alunograd, Matriculagrad) 1:N{
  opcao_reg  : string
}
Gradegrad (Disciplinagrad, Enfasegrad) N:N{
  perfil_grd : string,
  userid_grd : string,
  datatu_grd : string
}
```

Listagem 4.12 – Modelo Entidade - Relacionamento, estudo de caso Progradweb

4.2.1 Mapeamento com cardinalidade 1-1

Na Listagem 4.13 é apresentada uma das opções de mapeamento para entidades com cardinalidade 1-1. No tipo de documento *DocTypeAlunograd* foram embutidos os atributos da entidade *Endereco*.

```
DocTypeAlunograd [ Alunograd(main=true), Mora(main=false),
   Endereco(main=false) ]
{
   _id            : int [Alunograd.codalu_alug]
   fNomealu_alug    : string [Alunograd.nomealu_alug]
   fDatanasc_alug : string [Alunograd.datanasc_alug]
   fCpf_alug    : string [Alunograd.cpf_alug]
   data_Endereco:
   {
    fCod_endereco : int [Endereco.cod_endereco]
    fNome_cidade  : string [Endereco.nome_cidade]
    fNome_bairro  : sgring [Endereco.nome_bairro]
    fCep          : string [Endereco.cep]
    fNome_rua     : string [Endereco.nome_rua]
    fComplemento  : string [Endereco.complemento]
   }
}
DocTypeEndereco [ Endereco (main=true) ]
{
  _id       : int [Endereco.cod_endereco]
  fNome_cidade  : string [Endereco.nome_cidade]
  fNome_bairro  : sgring [Endereco.nome_bairro]
  fCep          : string [Endereco.cep]
  fNome_rua     : string [Endereco.nome_rua]
  fComplemento  : string [Endereco.complemento]
```

```
25 }
```

Listagem 4.13 – Esquema MongoDB, Mapeamento 1-1

Neste mapeamento, o valor da entidade 1 é *Alunograd*, da entidade 2 é *Endereco* e do relacionamento é *Mora*. A seguir mostra-se a consulta de entrada para este exemplo: *"from Alunograd a rjoin Mora m (Endereco e) select a.nomealu_alug, a.cpf_alug, e.nome_cidade, e.nome_bairro, e.cep"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.13 e da consulta de entrada, é executado o algoritmo, tendo como resultado a consulta em JavaScript apresentada na Listagem 4.14. O algoritmo a ser executado neste mapeamento, corresponde ao citado na Listagem 3.14, onde o relacionamento não possui atributos.

Para gerar a consulta, primeiramente, o acesso é feito ao tipo de documento *DocTypeAlunograd*, onde é recuperado os atributos da entidade 1 (linhas 1-8), e atributos da entidade 2 (linhas 9-20).

```
1  db.DocTypeAlunograd.find().forEach( function(data) {
2     db.EC.insert( {
3        data_Alunograd: {
4          _id: data._id,
5          fNomealu_alug: data.fNomealu_alug,
6          fDatanasc_alug: data.fDatanasc_alug,
7          fCpf_alug: data.fCpf_alug
8        },
9        data_Join: [{
10            data_Endereco: {
11                 fCod_endereco : data.data_Endereco.fCod_endereco,
12                 fNome_cidade  : data.data_Endereco.fNome_cidade,
13                 fNome_bairro  : data.data_Endereco.fNome_bairro,
14                 fCep          : data.data_Endereco.fCep,
15                 fNome_rua     : data.data_Endereco.fNome_rua,
16                 fComplemento  : data.data_Endereco.fComplemento
17       }
18     }]
19   });
20 });
```

Listagem 4.14 – Consulta em JavaScript, Mapeamento 1-1

4.2.2 Mapeamento com cardinalidade 1-N

Na Listagem 4.15 é apresentada uma das opções de mapeamento para entidades com cardinalidade 1-N. No tipo de documento *DocTypeAlunograd* foram embutidos den-

tro de um *array* os atributos da entidade *Matriculagrad* e do relacionamento *Registra*. Neste mapeamento, o valor da entidade 1 é *Alunograd*, da entidade é 2 *Matriculagrad* e o relacionamento é *Registra*.

```
1  DocTypeAlunograd [ Alunograd (main=true), Matriculagrad (main=
       true), Registra (main=true)]
2  {
3     _id             : int [ Alunograd.codalu_alug ]
4     fNomealu_alug   : string [ Alunograd.nomealu_alug ]
5     fDatanasc_alug  : string [ Alunograd.datanasc_alug ]
6     fCpf_alug       : string [ Alunograd.cpf_alug]
7     data_Matriculagrad:
8      [
9        {
10          fCodalu_matr : int [Matriculagrad.codalu_matr]
11          fCodenf_matr : int [Matriculagrad.codenf_matr]
12          fAnoini_matr : string [Matriculagrad.anoini_matr]
13          fSemini_matr : string [Matriculagrad.semini_matr]
14          fOpcao_reg   : string [Registra.opcao_reg]
15       }
16     ]
17 }
```

Listagem 4.15 – Esquema MongoDB, Mapeamento 1-N

A seguir mostra-se a consulta de entrada para este exemplo: *"from Alunograd a rjoin Registra r (Matriculagrad m) select a.nomealu_alug, a.cpf_alug, m.codenf_matr, m.anoini_matr, m.semini_matr, r.opcao_reg"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.15 e da consulta de entrada, é executado o algoritmo, tendo como resultado a consulta em JavaScript apresentada na Listagem 4.16. O acesso é feito ao tipo de documento *DocTypeAlunograd*, onde é recuperado os atributos da entidade 1 (linhas 1-10), em seguida com o método *forEach* é percorrido o campo *data_Matriculagrad* e atualizado os atributos do relacionamento e da entidade 2 (linhas 11-27).

```
1  db.DocTypeAlunograd.find().forEach(function(data) {
2    db.EC.insert( {
3      data_Alunograd: {
4          _id: data._id,
5          fNomealu_alug: data.fNomealu_alug,
6          fDatanasc_alug: data.fDatanasc_alug,
7          fCpf_alug: data.fCpf_alug
8      },
```

```
    data_Join: []
  });
  data.data_Matriculagrad.forEach(function(data1){
  db.EC.update({'data_Alunograd._id':data._id},
  { $addToSet: {
    'data_Join': {
        data_Registra: {
            fOpcao_reg: data1.fOpcao_reg
        },
        data_Matriculagrad: {
            fCodalu_matr: data1.fCodalu_matr,
            fCodenf_matr: data1.fCodenf_matr,
            fAnoini_matr: data1.fAnoini_matr,
            fSemini_matr: data1.fSemini_matr
        }
     }}
  });
 });
});
```

Listagem 4.16 – Consulta em JavaScript, Mapeamento 1-N

4.2.3 Mapeamento com cardinalidade N-1

Na Listagem 4.17 é apresentada uma das opções de mapeamento para entidades com cardinalidade N-1. Neste mapeamento, o valor da entidade 1 é *Matriculagrad*, entidade 2 *Alunograd* e do relacionamento é *Registra*. O *DocTypeMatriculagrad* contêm os campos embutidos da entidade 2, e o atributo do relacionamento definido como um campo mais do documento.

```
DocTypeMatriculagrad [ Matriculagrad(main=true), Alunograd(main=
   false), Registra(main=true)]
{
   _id          : int [Matriculagrad.codalu_matr]
   fCodenf_matr : int [Matriculagrad.codenf_matr]
   fAnoini_matr : string [Matriculagrad.anoini_matr]
   fSemini_matr : string [Matriculagrad.semini_matr]
   fOpcao_reg   : string [Registra.opcao_reg]
   data_Alunograd:
   {
     fcodalu_alug: int [ Alunograd.codalu_alug ]
   }
}
```

```
13 DocTypeAlunograd [ Alunograd (main=true)]
14 {
15    _id            : int [ Alunograd.codalu_alug ]
16    fNomealu_alug  : string [ Alunograd.nomealu_alug ]
17    fDatanasc_alug: string [ Alunograd.datanasc_alug ]
18    fCpf_alug      : string [ Alunograd.cpf_alug]
19 }
```

Listagem 4.17 – Esquema MongoDB, Mapeamento N-1

A seguir mostra-se a consulta de entrada para este exemplo: *"from Matriculagrad m rjoin Registra r (Alunograd a) select m.codenf_matr, m.anoini_matr, m.semini_matr, r.opcao_reg, a.nomealu_alug, a.cpf_alug"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.17 e da consulta de entrada, é executado o algoritmo, tendo como resultado a consulta em JavaScript apresentada na Listagem 4.18. O acesso é feito ao tipo de documento *DocTypeMatriculagrad*, onde é recuperado e inserido na *EC*, os atributos da entidade 1 (linhas 1-8), e os atributos do relacionamento e da entidade 2 (linhas 9-18). Em seguida, como ainda faltam atributos da entidade 2, é completada e atualizada realizando a junção entre *EC* e *DocTypeAlunograd* (linhas 19-29).

```
1  db.DocTypeMatriculagrad.find().forEach(function(data) {
2    db.EC.insert({
3      data_Matriculagrad: {
4          _id: data._id,
5          fCodenf_matr: data.fCodenf_matr,
6          fAnoini_matr: data.fAnoini_matr,
7          fSemini_matr: data.fSemini_matr,
8      },
9      data_Join: [{
10         data_Registra: {
11             fOpcao_reg: data.fOpcao_reg
12         },
13         data_Alunograd: {
14             fCodalu_alug: data.data_alunograd.fCodalu_alug
15         },
16     }]
17   });
18 });
19 db.EC.find().forEach(function(data){
20   var novoArray = [];
21   data.data_Join.forEach(function(data2) {
```

```
22     data2.data_Alunograd = db.DocTypeAlunograd.findOne({'_id':
       data2.data_Alunograd.fCodalu_alug});
23        novoArray.push(data2);
24  });
25  db.EC.update({'data_Matriculagrad._id': data.data_Matriculagrad.
       _id},
26  { $set: {
27     'data_Join': novoArray
28  }});
29 });
```

Listagem 4.18 – Consulta em JavaScript, Mapeamento N-1

4.2.4 Mapeamento com cardinalidade N-N

Na Listagem 4.19 é apresentada uma das opções de mapeamento para entidades com cardinalidade N-N. Modelamos o relacionamento N:N com incorporação de uma via, "*One Way Embedding*" (TRIVEDI, 2014). No tipo de documento *DocTypeGradegrad*, foram adicionados os campos identificadores dos tipos de documentos *DocTypeDisciplinagrad* e *DocTypeEnfasegrad*. Assim, é possível recuperar os dados das entidades Disciplinagrad e Enfasegrad através dos atributos identificadores.

```
1  DocTypeDisciplinagrad [ Disciplinagrad (main=true) ]
2  {
3     _id           : string [ Disciplinagrad.coddiscip_discip ]
4     fNome_discip  : string [ Disciplinagrad.nome_discip ]
5     fSetor_discip: string [ Disciplinagrad.setor_discip ]
6  }
7  DocTypeEnfasegrad [ Enfasegrad (main=true) ]
8  {
9      id             : int [ Enfasegrad.codenf_enf ]
10     fSiglaenf_enf  : string [Enfasegrad.siglaenf_enf]
11     fNomeenf_enf   : string [Enfasegrad.nomeenf_enf]
12 }
13 DocTypeGradegrad [Gradegrad (main=true), Disciplinagrad (main=
       false),  Enfasegrad (main=false)]
14 {
15     perfil_grd         : date [ Gradegrad.perfil_grd ]
16     userid_grd         : string [ Gradegrad.userid_grd ]
17     datatu_grd         : string [ Gradegrad.datatu_grd ]
18     fCoddiscip_discip  : int [ Disciplinagrad.coddiscip_discip ]
19     fCodenf_enf        : int [ Enfasegrad.codenf_enf ]
```

```
}
```

Listagem 4.19 – Esquema MongoDB, Mapeamento N-N

A seguir mostra-se a consulta de entrada para este exemplo: *"from Disciplinagrad d rjoin Gradegrad g (Enfasegrad e) select d.coddiscip_discip, d.nome_discip, e.siglaenf_enf, e.nomeenf_enf, g.perfil_grd, g.userid_grd, g.datatu_grd"*.

A partir deste mapeamento no esquema MongoDB mostrada na Listagem 4.19 e da consulta de entrada, é executado o algoritmo, tendo como resultados as consultas em JavaScript apresentadas na Listagem 4.20 e 4.21. Na Listagem 4.20 o acesso é feito ao tipo de documento *DocTypeDisciplinagrad*, onde é recuperado e inserido na *EC*, os atributos da entidade 1 "Disciplinagrad" (linhas 1-10). Depois, é realizada a junção entre *EC* e *DocTypeGradegrad*, onde são recuperados e atualizados na *EC* os atributos do relacionamento "Gradegrad" e da entidade 2 "Enfasegrad' (linhas 11-34). Como faltam atributos da Entidade 2, é realizada outra junção entre *EC* e *DocTypeEnfasegrad*, onde finalmente são recuperados e atualizados os todos os atributos da entidade 2.

```
1  db.DocTypeDisciplinagrad.find().forEach(function(data){
2    db.EC.insert({
3      data_Disciplinagrad: {
4       _id: data._id,
5       fNome_discip: data.fNome_discip,
6       fSetor_discip: data.fSetor_discip
7      },
8      data_Join: []
9    });
10 });
11 db.EC.find().forEach( function(data){
12   var varData = [];
13   data.data_Join.forEach(
14    function(dataCopy) {
15       varData.push(dataCopy);
16    });
17    db.DocTypeGradegrad.find({'fCoddiscip_grd': data.
      data_Disciplinagrad._id }).forEach(
18      function(data2){
19       varData.push({
20         data_Enfasegrad: {
21           fCodenf_grd: data2.fCodenf_grd
22         },
23         data_Gradegrad: {
24           perfil_grd: data2.perfil_grd,
```

```
25            userid_grd: data2.userid_grd,
26            datatu_grd: data2.datatu_grd
27          }
28        })
29       });
30    db.EC.update({'data_Disciplinagrad._id': data.
    data_Disciplinagrad._id},
31    { $set: {
32       'data_Join': varData
33     }});
34 });
35 db.EC.find().forEach( function(data){
36   var novoArray = [];
37   data.data_Join.forEach(function(data2){
38     data2.data_Enfasegrad = db.DocTypeEnfasegrad.findOne({ '_id':
     data2.data_Enfasegrad.fCodenf_grd });
39         novoArray.push(data2);
40     });
41     db.EC.update( {'data_Disciplinagrad._id': data.
    data_Disciplinagrad._id},
42     { $set: {
43       'data_Join': novoArray
44     }});
45 });
```

Listagem 4.20 – Consulta em JavaScript, Mapeamento N-N, Query 1

Para a mesma consulta de entrada descrita anteriormente, o algoritmo retorna a consulta apresentada na Listagem 4.21. O acesso é feito ao tipo de documento *DocType-Gradegrad*, onde é recuperado e inserido na *EC*, os atributos da entidade 1 "Disciplinagrad" (linhas 1-7) e atualizado os atributos do relacionamento "Gradegrad" e da entidade 2 "Enfasegrad" (linhas 9-22). Na linha 8 é criado um índice, de maneira e evitar a inserção de dois Disciplinas com mesmo *fCoddiscip_grd*. Com a operação de junção entre *EC* e *DocTypeDisciplinagrad* são atualizados os dados da entidade 1 dentro da *EC* (linhas 23-38). Depois, é realizada outra junção entre entre *EC* e *DocTypeEnfasegrad*, onde finalmente são recuperados e atualizados os todos os atributos da entidade 2 dentro da *EC* (linha 39-49).

```
1 db.DocTypeGradegrad.find().forEach(function(data){
2  db.EC.insert({
3   data_Disciplinagrad: {
4      fCoddiscip_grd: data.fCoddiscip_grd
5   },
```

```
  data_Join: []
  });
  db.EC.createIndex(‘data_Disciplinagrad.fCoddiscip_grd’: 1 }, {
   unique: true });
  db.EC.update({‘data_Disciplinagrad.fCoddiscip_grd’:data.
   fCoddiscip_grd},
  {$addToSet: {
    ’data_Join’: {
        data_Gradegrad: {
            perfil_grd: data.perfil_grd,
            userid_grd: data.userid_grd,
            datatu_grd: data.datatu_grd
        },
        data_Enfasegrad: {
            fCodenf_grd: data.fCodenf_grd
        }
    }
  }});
});
db.EC.find().forEach( function(data){
 var varData = [];
 db.DocTypeDisciplinagrad.find({‘_id’: data.data_Disciplinagrad.
   fCoddiscip_grd }).forEach(
   function(data2) {
     varData.push({
       data_Disciplinagrad: {
         fCoddiscip_grdID: data2._id,
         fNome_discip: data2.fNome_discip,
         fSetor_discip: data2.fSetor_discip
     }})
   }) ;
   db.EC.updateMany({‘data_Disciplinagrad.fCoddiscip_grd’: data.
   data_Disciplinagrad.fCoddiscip_grd},
   { $set: {
     ‘data_Disciplinagrad’: varData[0].data_Disciplinagrad
    }})
});
db.EC.find().forEach( function(data){
 var novoArray = [];
 data.data_Join.forEach(function(data2) {
   data2.data_Enfasegrad = db.DocTypeEnfasegrad.findOne({ ‘_id’:
   data2.data_Enfasegrad.fCodenf_grd });
```

```
43    novoArray.push(data2);
44    });
45    db.EC.update({'data_Disciplinagrad.fCoddiscip_grd': data.
      data_Disciplinagrad.fCoddiscip_grd},
46    { $set: {
47      'data_Join': novoArray
48    }});
49 });
```

Listagem 4.21 – Consulta em JavaScript, Mapeamento N-N, Query 2

4.3 Considerações finais

Os modelos e mapeamentos mostrados neste capítulo são apenas algumas das muitas possibilidades experimentadas. Após a definição dos mapeamentos nos metamodelos, a implementação do algoritmo e a execução das consultas, alguns aspectos podem ser considerados:

- É possível mapear um campo identificador por *DocumentType*.
- O algoritmo gera consultas envolvendo projeção de dados e junção de duas entidades.
- O algoritmo recupera todos os atributos expressados na lista *queryAttributes*, no entanto, ele não desconsidera os demais atributos pertencentes a entidade/relacionamento. Por esse motivo, afirma-se que o suporte à projeção é apenas parcial.
- Em caso de que o *DocumentType* ao qual foi mapeado a Entidade 1 tiver *main=false*, e realizar uma projeção dos dados sobre ele, o algoritmo emite uma advertência de que a consulta pode não retornar todas as instancias da entidade.
- Em caso de buscar um *DocumentType* para realizar junção com a *Entidade Computada*, se o *DocumentType* encontrado não possui identificadores comuns, o algoritmo emite uma mensagem de que é impossivel realizar junção, pois não há atributos *id* comuns entre os *DocumentTypes*. Se o *DocumentType* tiver mapeado *main=false*, o algoritmo emite a mensagem de que o mapeamento não é principal. Ou seja, a junção é realizada se tiver *id* comuns entre os *DocumentTypes*, e se o *DocumentType* tiver mapeado com *main=true*.
- Todas as consultas geradas em JavaScript dentro de um mesmo tipo de cardinalidade (1-1, 1-N, N-1, N-N), independente do tipo de mapeamento, retornam a mesma estrutura do resultado da consulta. Excepcionalmente, o resultado da consulta, em alguns casos pode variar só a ordem dos campos da Entidade 2 e do Relaciomento dentro do campo 'data_Join' da *Entidade Computada*.

Ainda é possível identificar algumas limitações nos resultados:

- É possível mapear as entidades/relacionamentos como referências e documentos embutidos somente no primeiro nível;
- Devido às limitações na implementação, não foi possível testar consultas reais dos sistemas estudados, em especial o sistema PrograduWeb, para o qual existem muitas consultas.

Dadas as observações e limitações acima, considera-se que os resultados foram positivos, pois eles demonstram, por meio de um conjunto de casos de testes em que se buscou cobrir exaustivamente todas as possibilidades de mapeamento, que a abordagem cumpre o seu objetivo.

5 Conclusão

Como apresentado no Capítulo 2, Seção 2.3, a maior dificuldade apontada pelos trabalhos relacionados envolvendo modelos de dados é a heterogeneidade dos bancos NoSQL. Existem diversos bancos de dados não relacionais, onde cada um tem linguagem de consulta própria, tipos de armazenamentos, estratégias, ferramentas, modelo de dados e outras características que tornam o processo de manipulação de dados uma atividade complexa (DHARMASIRI; GOONETILLAKE, 2013). É necessário levar em consideração que mesmo quando um único tipo de banco NoSQL é utilizado, existe mais de uma estratégia diferente para se organizar os dados.

Devido à falta de um padrão para as linguagens de manipulação de dados (DML) que unifica e simplifica as consultas em armazenamentos de dados NoSQL, o intuito geral dessa pesquisa foi possibilitar ao desenvolvedor especificar consultas de forma conceitual, independente do modelo de dados, utilizando para isso uma linguagem genérica. Posterior, automatizar o processo de tradução dessas consultas para consultas específicas na linguagem JavaScript, por meio de técnicas do desenvolvimento de software dirigido por modelos.

5.1 Contribuições alcançadas

Considera-se que o objetivo foi alcançado com sucesso. Ficou demonstrado ser possível que o Engenheiro de Software consiga especificar consultas uma única vez, utilizando apenas elementos do modelo ER, e gerar consultas automaticamente que obtém os mesmos resultados independente da forma com que o mapeamento é definido.

Outra contribuição importante foi a extensão de um trabalho desenvolvido por outros pesquisadores décadas atrás. Trabalhando com base na pesquisa de Parent e Spaccapietra (1984), foi desenvolvida uma semântica operacional para uma álgebra teórica, definida em 1984, que ganhou nova vida à luz de tecnologias recentes. Apesar de não ser originalmente concebida para isso, a álgebra de Parent e Spaccapietra (1984) se mostrou uma solução quase que perfeitamente ajustada para o objetivo desta pesquisa e completamente compatível com as tecnologias NoSQL.

Esta dissertação de mestrado apresentou também a geração de 3 metamodelos, a especificação de uma Gramática Livre de Contexto (GLC) e a implementação de um algoritmo que realiza a conversão de consultas especificadas numa linguagem comum para consultas escritas em JavaScript.

Por fim, uma contribuição importante desse projeto de mestrado é expor as pes-

quisas relacionadas à área. Muitas dessas recuperam dados do armazenamento NoSQL a partir de chaves primárias e a abordagem de consultas complexas é pouco estudada. O projeto desenvolvido abre espaços para novas linhas de pesquisa que podem ser exploradas, diferentes das vistas tradicionalmente na área de recuperação de dados, considerando todas as possíveis formas de mapeamento tratadas com o desenvolvimento de software dirigido por modelos e bancos de dados não relacionais orientado a documentos. Portanto, a última contribuição dessa pesquisa é a iniciativa em uma nova linha de pesquisa pouco explorada até então.

5.2 Limitações e Trabalhos futuros

Assim como previsto pela literatura, o tema desta pesquisa não é trivial. Conforme discutido no Capítulo 2, outros pesquisadores abordaram temas parecidos com os desta pesquisa, mas deixaram de lado a questão de consultas não triviais. De fato, uma das primeiras constatações feitas durante a realização desta pesquisa é que a literatura tinha razão. O tema não é trivial. E a ausência de trabalhos investigando o tema é completamente justificado. Fica como lição aprendida, que o assunto é delicado e exige ainda uma longa investigação, envolvendo aspectos técnicos e também não técnicos.

Do ponto de vista técnico, várias decisões foram feitas nesta pesquisa, visando limitar um pouco o trabalho para viabilizar a obtenção de resultados dentro do prazo de um curso de mestrado:

- A semântica completa da álgebra de Parent e Spaccapietra (1984) não foi implementada. O aspecto mais complicado (junção entre duas entidades) foi implementado, no entanto ainda restam, a saber, consultas simples, projeção, junções compostas, funções de agregação, e lógica de comparação, entre outras funções;
- Não se teve preocupação com o desempenho das consultas geradas, e sim com a corretude e consistência com a álgebra de Parent e Spaccapietra (1984). Este é um aspecto importante a ser considerado em trabalhos futuros, já que é premissa da abordagem NoSQL a busca pelo melhor desempenho;
- A gramática livre de contexto que formaliza a sintaxe concreta possui ao menos uma limitação identificada, além de outras que possam surgir no futuro;
- Não foi implementada a tradução da gramática para código-fonte que executa o algoritmo de geração;
- Os metamodelos não foram implementados utilizando EMF, e sim manualmente em Java;

- Não foi implementada nenhuma forma mais amigável de criação de modelos. Apesar de ter sido criada uma representação textual, a mesma foi utilizada apenas para facilitar a leitura deste documento. No entanto, essa representação se mostrou bastante útil, e pode ser utilizada como inspiração para uma ferramenta de modelagem.

Todas essas limitações são sugestões para trabalhos futuros. Como mencionado na Seção 5.1, uma nova linha de pesquisa pouco explorada pode atrair a atenção de pesquisadores da área de computação e principalmente na área de mapeamento dos esquemas conceituais para os modelos físicos de bancos de dados não relacionais.

O primeiro e mais importante trabalho futuro proposto é a implementação de toda a semântica da linguagem. Isso irá permitir, além de um refinamento no algoritmo, que os testes com sistemas e consultas reais sejam realizados, gerando evidências muito mais sólidas sobre a viabilidade da abordagem.

Outro possível trabalho futuro é a análise inteligente das informações existentes na abordagem. Em tese, é possível analisar os modelos fornecidos e as consultas, e automaticamente determinar quais consultas geradas são as que tem melhor desempenho. É possível também executar análises e simulações visando analisar a qualidade dos mapeamentos e até mesmo sugerir qual o melhor mapeamento deve ser adotado em cada caso, ajudando o Engenheiro de Software no refinamento do banco.

Técnicas de otimização automática também podem ser exploradas para melhorar o desempenho das consultas geradas, reduzindo operações desnecessárias, a exemplo do que já é feito em outras áreas. A indexação de documentos também pode ser uma importante fonte de otimização.

Também se deve considerar a construção das consultas em JavaScript com outros métodos, além dos utilizados nesta abordagem. As consultas foram feitas utilizando primariamente cursores, mas existem outras técnicas que podem ser empregadas, como a agregação do MongoDB, que pode se mostrar melhor em alguns casos.

Outro caminho que pode ser explorado é a execução distribuída. Muitos SGBDs não-relacionais possuem suporte à replicação de dados e distribuição. A geração de consultas poderia ser feita levando-se em conta a função exercida por frameworks de computação distribuída, visando desempenho, confiabilidade, entre outros benefícios.

Por fim, o projeto desenvolvido neste mestrado focou a conversão de consultas especificadas no modelo conceitual ER para linguagem JavaScript, linguagem utilizada no banco de dados orientado a documentos, MongoDB. Novas pesquisas podem abordar outros modelos conceituais, com seu respectivos construtores, outros modelos de dados NoSQL orientados a documentos, chave-valor, colunar e orientado a grafos. Tornar o algoritmo mais genérico para diferentes modelos NoSQL pode ser muito interessante para enriquecer o conhecimento na área.

Referências

ATZENI, P.; BUGIOTTI, F.; ROSSI, L. Uniform Access to Non-relational Database Systems: The SOS Platform. In: *Proceedings of the 24th International Conference on Advanced Information Systems Engineering*. Beralin, Heidelberg: Springer-Verlag, 2012. (CAiSE'12), p. 160–174. ISBN 978-3-642-31094-2. Disponível em: <http://dx.doi.org-/10.1007/978-3-642-31095-9_11>. Citado 9 vezes nas páginas 11, 21, 24, 32, 41, 49, 50, 51 e 63.

BARON, C. A. et al. NoSQL key-value DBs riak and redis. *Database Systems Journal*, Academy of Economic Studies-Bucharest, Romania, v. 4, p. 3–10, 2016. Citado 2 vezes nas páginas 40 e 41.

BRUNELIERE, H.; CABOT, J.; JOUAULT, F. Combining model-driven engineering and cloud computing. In: *Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*. [S.l.: s.n.], 2010. Citado na página 33.

BUGIOTTI, F. et al. *A Logical Approach to NoSQL Databases*. 2013. Citado na página 32.

CHANDRA, D. G. BASE analysis of NoSQL database. *Future Generation Computer Systems*, Elsevier, v. 52, p. 13–21, 2015. Citado na página 23.

CHANG, F. et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 26, n. 2, p. 4, 2008. Citado na página 37.

CHEN, P. P.-S. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 1, n. 1, p. 9–36, mar. 1976. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/320434.320440>. Citado na página 75.

CURÉ, O. et al. Data Integration over NoSQL Stores Using Access Path Based Mappings. In: *Proceedings of the 22Nd International Conference on Database and Expert Systems Applications - Volume Part I*. Berlin, Heidelberg: Springer-Verlag, 2011. (DEXA'11), p. 481–495. ISBN 978-3-642-23087-5. Disponível em: <http://dl.acm.org-/citation.cfm?id=2035368.2035414>. Citado 9 vezes nas páginas 11, 13, 21, 24, 49, 59, 60, 61 e 63.

DANIEL, G.; SUNYÉ, G.; CABOT, J. Mogwaï: A framework to handle complex queries on large models. In: *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*. Grenoble, France: IEEE, 2016. p. 1–12. Citado 2 vezes nas páginas 64 e 65.

DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, ACM, New York, NY, USA, v. 41, n. 6, p. 205–220, 2007. Citado na página 38.

DEHDOUH, K. et al. Columnar NoSQL CUBE: Agregation operator for columnar NoSQL data warehouse. In: *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. San Diego, CA, USA: IEEE, 2014. p. 3828–3833. ISSN 1062-922X. Citado na página 43.

DEURSEN, A. V. et al. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, New York, NY, USA, v. 5, p. 12, 2000. Citado na página 46.

DHARMASIRI, H. M. L.; GOONETILLAKE, M. D. J. S. A federated approach on heterogeneous NoSQL data stores. In: *2013 International Conference on Advances in ICT for Emerging Regions (ICTer)*. Colombo, Sri Lanka: IEEE, 2013. p. 234–239. Citado 3 vezes nas páginas 24, 65 e 125.

ELMASRI, R. et al. *Sistemas de banco de dados*. 6. ed. [S.l.]: Pearson Addison Wesley, 2010. 808 p. Citado 3 vezes nas páginas 24, 71 e 105.

FADOUA, H.; AMEL, G. T. Heterogeneous nosql databases abstraction approach based on full text search indexes. In: _____. *Intelligent Decision Technologies 2016: Proceedings of the 8th KES International Conference on Intelligent Decision Technologies (KES-IDT 2016) – Part II*. Cham: Springer International Publishing, 2016. p. 417–427. ISBN 978-3-319-39627-9. Disponível em: <http://dx.doi.org/10.1007/978-3-319-39627-9_37>. Citado na página 38.

FIORAVANTI, S. et al. Experimental performance evaluation of different data models for a reflection software architecture over nosql persistence layers. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. New York, NY, USA: ACM, 2016. (ICPE '16), p. 297–308. ISBN 978-1-4503-4080-9. Disponível em: <http://doi.acm.org/10.1145/2851553.2851561>. Citado na página 32.

FRANCE, R.; RUMPE, B. Model-driven Development of Complex Software: A Research Roadmap. In: *2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 37–54. ISBN 0-7695-2829-5. Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.14>. Citado na página 33.

GIACHETTI, G.; MARIN, B.; PASTOR, O. Using UML profiles to interchange DSML and UML models. In: *2009 Third International Conference on Research Challenges in Information Science*. Fez, Morocco: IEEE, 2009. p. 385–394. ISSN 2151-1349. Citado na página 45.

GILBERT, S.; LYNCH, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, ACM, New York, NY, USA, v. 33, n. 2, p. 51–59, jun. 2002. ISSN 0163-5700. Disponível em: <http://doi.acm.org/10-.1145/564585.564601>. Citado na página 38.

GRONBACK, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321534077, 9780321534071. Citado 2 vezes nas páginas 47 e 49.

HAN, J. et al. Survey on NoSQL database. In: *2011 6th International Conference on Pervasive Computing and Applications*. Port Elizabeth, South Africa: IEEE, 2011. p. 363–366. Citado na página 38.

KARNITIS, G.; ARNICANS, G. Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation. In: *2015 7th International Conference on Computational Intelligence, Communication Systems and Networks*. Riga, Latvia: IEEE, 2015. p. 113–118. Citado na página 23.

KAUR, K.; RANI, R. Modeling and querying data in NoSQL databases. In: *2013 IEEE International Conference on Big Data*. Silicon Valley, CA, USA: IEEE, 2013. p. 1–7. Citado 2 vezes nas páginas 24 e 44.

KAUR, K.; RANI, R. A smart polyglot solution for big data in healthcare. *IT Professional*, IEEE, v. 17, n. 6, p. 48–55, 2015. Citado na página 43.

KITCHENHAM, B.; CHARTERS, S. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. 2007. Citado na página 33.

KLEPPE, A. G.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 032119442X. Citado na página 45.

KOLEV, B. et al. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, Springer, v. 34, n. 4, p. 463–503, 2016. Citado 3 vezes nas páginas 39, 63 e 64.

LI, X.; MA, Z.; CHEN, H. QODM: A query-oriented data modeling approach for NoSQL databases. In: *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*. Ottawa, ON, Canada: IEEE, 2014. p. 338–345. Citado 9 vezes nas páginas 11, 21, 23, 32, 49, 57, 58, 59 e 63.

LIMA, C. de; MELLO, R. dos S. A workload-driven logical design approach for nosql document databases. In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*. New York, NY, USA: ACM, 2015. (iiWAS '15), p. 73:1–73:10. ISBN 978-1-4503-3491-4. Disponível em: <http://doi.acm.org/10.1145/2837185.2837218>. Citado na página 32.

LUCRÉDIO, D. Uma abordagem orientada a modelos para reutilização de software. *INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO UNIVERSIDADE DE SÃO PAULO*, p. 37, 2009. Citado 4 vezes nas páginas 11, 46, 47 e 48.

MATHEW, A. B.; KUMAR, S. D. M. Analysis of data management and query handling in social networks using NoSQL databases. In: *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Kochi, India: IEEE, 2015. p. 800–806. Citado na página 41.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM, New York, NY, USA, v. 37, n. 4, p. 316–344, 2005. Citado na página 46.

MONGODB. *Manual MongoDB*. 2018. Acessado: 15/mar/2018. Disponível em: <https://docs.mongodb.com/manual/>. Citado 2 vezes nas páginas 34 e 41.

MOORE, B. et al. Eclipse development using the graphical editing framework and the eclipse modeling framework). *IBM Redbook*, NJ, USA, 2004. Citado na página 49.

NAYAK, A.; PORIYA, A.; POOJARY, D. Type of NOSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, New York, USA, v. 5, n. 4, p. 16–19, 2013. Citado 4 vezes nas páginas 40, 41, 43 e 44.

PARENT, C.; SPACCAPIETRA, S. An entity-relationship algebra. In: *1984 IEEE First International Conference on Data Engineering*. Los Angeles, CA, USA, USA: IEEE, 1984. p. 500–507. Citado 41 vezes nas páginas 11, 21, 22, 32, 33, 34, 49, 61, 62, 63, 69, 70, 81, 86, 99, 100, 101, 103, 105, 107, 109, 111, 125, 126, 135, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164 e 166.

PEREIRA, D.; OLIVEIRA, P.; RODRIGUES, F. Data warehouses in MongoDB vs SQL Server: A comparative analysis of the querie performance. In: *2015 10th Iberian Conference on Information Systems and Technologies (CISTI)*. Aveiro, Portugal: IEEE, 2015. p. 1–7. ISSN 2166-0727. Citado 3 vezes nas páginas 40, 43 e 44.

POKORNY, J. NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, Emerald Group Publishing Limited, v. 9, n. 1, p. 69–82, 2013. Citado 3 vezes nas páginas 38, 40 e 41.

PRITCHETT, D. Base: An acid alternative. *Queue*, ACM, New York, NY, USA, v. 6, n. 3, p. 48–55, 2008. Citado na página 38.

RANKING. *DB-Engines Ranking*. 2018. Https://db-engines.com/en/ranking. Acessado: 08/mar/2018. Citado na página 24.

ROSA, A. S. et al. Uma abordagem orientada a modelos para modelagem conceitual de banco de dados. *Anais SULCOMP*, v. 6, 2013. Citado na página 75.

ROSS, D. T. Origins of the APT Language for Automatically Programmed Tools. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 13, n. 8, p. 61–99, ago. 1978. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/960118.808374>. Citado na página 46.

SADALAGE, P. J.; FOWLER, M. *NoSQL Essencial: Um guia conciso para o Mundo emergente da persistência poliglota*. [S.l.]: Novatec Editora, 2013. Citado 9 vezes nas páginas 13, 23, 37, 38, 39, 40, 41, 43 e 44.

SCAVUZZO, M.; NITTO, E. D.; CERI, S. Interoperable Data Migration between NoSQL Columnar Databases. In: *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*. Ulm, Germany: IEEE, 2014. p. 154–162. ISSN 2325-6583. Citado na página 23.

SCHMID, S.; GALICZ, E.; REINHARDT, W. WMS performance of selected SQL and NoSQL databases. In: *International Conference on Military Technologies (ICMT) 2015*. Brno, Czech Republic: IEEE, 2015. p. 1–6. Citado na página 23.

SELLAMI, R.; BHIRI, S.; DEFUDE, B. ODBAPI: A Unified REST API for Relational and NoSQL Data Stores. In: *2014 IEEE International Congress on Big Data*. Anchorage, AK, USA: IEEE, 2014. p. 653–660. ISSN 2379-7703. Citado 9 vezes nas páginas 11, 13, 21, 32, 34, 49, 52, 53 e 63.

SELLAMI, R.; BHIRI, S.; DEFUDE, B. Supporting Multi Data Stores Applications in Cloud Environments. *IEEE Transactions on Services Computing*, v. 9, n. 1, p. 59–71, Jan 2016. ISSN 1939-1374. Citado 11 vezes nas páginas 11, 13, 21, 24, 49, 54, 55, 56, 57, 63 e 64.

SILVA, E. A. N. d. et al. *Uma abordagem dirigida por modelos para portabilidade entre plataformas de computação em nuvem*. Dissertação (Mestrado) — Universidade Federal de São Carlos, 2013. Citado 4 vezes nas páginas 46, 48, 49 e 74.

STANESCU, L.; BREZOVAN, M.; BURDESCU, D. D. Automatic mapping of MySQL databases to NoSQL MongoDB. In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. Gdansk, Poland: IEEE, 2016. p. 837–840. Citado 3 vezes nas páginas 24, 108 e 109.

STONEBRAKER, M. SQL databases v. NoSQL databases. *Communications of the ACM*, ACM, New York, NY, USA, v. 53, n. 4, p. 10–11, 2010. Citado na página 24.

STONEBRAKER, M. Stonebraker on NoSQL and enterprises. *Commun. ACM*, New York, NY, USA, v. 54, n. 8, p. 10–11, 2011. Citado na página 24.

STROZZI, C. *NoSQL: A non-SQL RDBMS*. 2010. Www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/. Acessado: 18/mai/2017. Citado na página 37.

TRIVEDI, A. *Mapping Relational Databases and SQL to MongoDB*. 2014. Https://code.tutsplus.com/articles/mapping-relational-databases-and-sql-to-mongodb–net-35650. Acessado: 25/fev/2018. Citado 3 vezes nas páginas 108, 109 e 118.

VERA W. BOAVENTURA, M. H. V. G. F. H. H. Data Modeling for NoSQL Document-Oriented Databases. 2015. Citado na página 99.

WANG, G.; TANG, J. The NoSQL Principles and Basic Application of Cassandra Model. In: *2012 International Conference on Computer Science and Service System*. Nanjing, China: IEEE, 2012. p. 1332–1335. Citado 2 vezes nas páginas 11 e 38.

WENDE, C. et al. Feature-Based Customisation of Tool Environments for Model-Driven Software Development. In: *2011 15th International Software Product Line Conference*. Munich, Germany: IEEE, 2011. p. 45–54. Citado na página 45.

YALAMANCHI, A.; GAWLICK, D. Compensation-aware Data Types in RDBMS. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2009. (SIGMOD '09), p. 931–938. ISBN 978-1-60558-551-2. Disponível em: <http://doi.acm.org/10.1145/1559845.1559950>. Citado na página 38.

YANG, D.; LIU, M.; WANG, S. Object-oriented methodology meets MDA software paradigm. In: *2012 IEEE International Conference on Computer Science and Automation Engineering*. Beijing, China: IEEE, 2012. p. 208–211. ISSN 2327-0586. Citado na página 48.

APÊNDICE A – Testes Modelo ER de Parent e Spaccapietra (1984)

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade 1-1, sendo *Person* igual a Entidade 1, *DriversLicense* a Entidade 2 e *Registration* o relacionamento.

```
Entity person = (Entity) mm.getERModel().findERElement("Person");
Relationship registration = (Relationship) mm.getERModel().findERElement("Registration");
Entity driversLicense = (Entity) mm.getERModel().findERElement("DriversLicense");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(person.getAttribute("name"));
queryAttributes.add(person.getAttribute("address"));
queryAttributes.add(registration.getAttribute("observation"));
queryAttributes.add(driversLicense.getAttribute("number"));

List<Query> queries = ma.binaryJoin(person, registration, driversLicense, queryAttributes);
```

Figura 24 – Lista *queryAttributes*, Modelo ER Parent, mapeamento 1-1

A.1 Mapeamentos com cardinalidade 1-1

```
DocTypePerson [ Person(main=true) ]
{
    _id      : int        [ Person. id ]
    fName    : string     [ Person.name ]
    fAddress: string      [ Person.address ]
}
DocTypeDriversLicense [ DriversLicense(main=true) ]
{
    _id      : int        [ DriversLicense.id ]
    fNumber  : string     [ DriversLicense.number ]
    fDate    : date       [ DriversLicense.date ]
}
DocTypeRegistration [ Registration(main=true), DriversLicense(
   main=false), Person(main=false) ]
{
    _id                  : int        [ ]
    observation          : string     [ Registration.observation ]
    fPersonId            : int        [ Person.id ]
```

```
    fName                : string    [ Person.name ]
    fAddress             : string    [ Person.address ]
    fDriversLicenseId    : int       [ DriversLicense.id ]
}
```

Listagem A.1 – Mapeamento 1-1, opção 1a

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
    db.DocTypeRegistration.find({ 'fPersonId': data.data_Person.
   _id }).forEach(
    function(data2) {
     varData.push( {
        data_DriversLicense: {
         fDriversLicenseId: data2.fDriversLicenseId
        },
        data_Registration: {
         observation: data2.observation
        }
     })
    });
 db.EC.update( {'data_Person._id': data.data_Person._id},
 {$set: {
    'data_Join': varData
 }});
});
db.EC.find().forEach( function(data){
    var novoArray = [];
    data.data_Join.forEach(function(data2) {
```

```
36      data2.data_DriversLicense = db.DocTypeDriversLicense.findOne
    ({ '_id': data2.data_DriversLicense.fDriversLicenseId });
37       novoArray.push(data2);
38      });
39    db.EC.update( {'data_Person._id': data.data_Person._id},
40    { $set: {
41    'data_Join': novoArray
42    }});
43 });
```

Listagem A.2 – Consulta em Javascript, Mapeamento 1-1, opção 1a, Query 1

```
1  db.DocTypeRegistration.find().forEach( function(data) {
2     db.EC.insert( {
3      data_Person: {
4          fPersonId: data.fPersonId
5      },
6      data_Join: [{
7          data_Registration: {
8              observation: data.observation
9          },
10         data_DriversLicense: {
11             fDriversLicenseId: data.fDriversLicenseId
12         }
13     }]
14    });
15 });
16 db.EC.find().forEach( function(data){
17     var varData = [];
18     db.DocTypePerson.find({ '_id': data.data_Person.fPersonId }).
    forEach(
19     function(data2) {
20      varData.push( {
21         data_Person: {
22          fPersonId: data2._id,
23          fName: data2.fName,
24          fAddress: data2.fAddress
25      }})
26     }) ;
27     db.EC.update( {'data_Person.fPersonId': data.data_Person.
    fPersonId},
28     { $set: {
29      'data_Person': varData[0].data_Person
```

```
    }})
});
db.EC.find().forEach( function(data){
    var novoArray = [];
    data.data_Join.forEach(function(data2) {
    data2.data_DriversLicense = db.DocTypeDriversLicense.findOne
   ({ '_id': data2.data_DriversLicense.fDriversLicenseId });
     novoArray.push(data2);
    });
    db.EC.update( {'data_Person.fPersonId': data.data_Person.
   fPersonId},
    { $set: {
        'data_Join': novoArray
    }});
});
```

Listagem A.3 – Consulta em Javascript, Mapeamento 1-1, opção 1a, Query 2

```
DocTypePerson [ Person(main=true) ]
{
    _id     : int       [ Person. id ]
    fName   : string    [ Person.name ]
    fAddress: string    [ Person.address ]
}
DocTypeDriversLicense [ DriversLicense(main=true) ]
{
    _id     : int       [ DriversLicense.id ]
    fNumber : string    [ DriversLicense.number ]
    fDate   : date      [ DriversLicense.date ]
}
DocTypeRegistration [ Registration(main=true), DriversLicense(
   main=false), Person(main=false)]
{
    _id                 : int       [ ]
    observation         : string    [ Registration.observation ]
    data_Person         :
    {
        fPersonId       : int       [ Person.id ]
    }
    fDriversLicenseId   : int       [ DriversLicense.id ]
}
```

Listagem A.4 – Mapeamento 1-1, opção 1b

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: []
    });
});
db.EC.find().forEach( function(data){
    var varData = [];
    data.data_Join.forEach(
    function(dataCopy) {
      varData.push(dataCopy);
    });
    db.DocTypeRegistration.find({ 'data_Person.fPersonId': data.
   data_Person._id }).forEach(
    function(data2) {
     varData.push( {
        data_DriversLicense: {
         fDriversLicenseId: data2.fDriversLicenseId
        },
        data_Registration: {
         observation: data2.observation
     }})
    });
    db.EC.update( {'data_Person._id': data.data_Person._id},
    { $set: {
        'data_Join': varData
    }});
});
db.EC.find().forEach( function(data){
    var novoArray = [];
    data.data_Join.forEach(function(data2) {
    data2.data_DriversLicense = db.DocTypeDriversLicense.findOne
   ({ '_id': data2.data_DriversLicense.fDriversLicenseId });
        novoArray.push(data2);
    });
    db.EC.update( {'data_Person._id': data.data_Person._id},
    { $set: {
        'data_Join': novoArray
```

```
    }});
});
```

Listagem A.5 – Consulta em Javascript, Mapeamento 1-1, opção 1b, Query 1

```
db.DocTypeRegistration.find().forEach( function(data) {
  db.EC.insert( {
    data_Person: {
        fPersonId: data.data_Person.fPersonId
    },
    data_Join: [{
        data_Registration: {
            observation: data.observation
        },
        data_DriversLicense: {
            fDriversLicenseId: data.fDriversLicenseId
        }
    }]
  });
});
db.EC.find().forEach( function(data){
    var varData = [];
    db.DocTypePerson.find({ '_id': data.data_Person.fPersonId }).
   forEach(
    function(data2) {
        varData.push( {
          data_Person: {
            fPersonId: data2._id,
            fName: data2.fName,
            fAddress: data2.fAddress
        }})
    });
  db.EC.update( {'data_Person.fPersonId': data.data_Person.
   fPersonId},
  { $set: {
    'data_Person': varData[0].data_Person
  }})
});
db.EC.find().forEach( function(data){
    var novoArray = [];
    data.data_Join.forEach(function(data2) {
    data2.data_DriversLicense = db.DocTypeDriversLicense.findOne
   ({ '_id': data2.data_DriversLicense.fDriversLicenseId });
```

```
36          novoArray.push(data2);
37      });
38      db.EC.update( {'data_Person.fPersonId': data.data_Person.
    fPersonId},
39      { $set: {
40          'data_Join': novoArray
41      }});
42  });
```

Listagem A.6 – Consulta em Javascript, Mapeamento 1-1, opção 1b, Query 2

```
1  DocTypePerson [ Person(main=true) ]
2  {
3      _id      : int        [ Person. id ]
4      fName    : string     [ Person.name ]
5      fAddress: string     [ Person.address ]
6  }
7  DocTypeDriversLicense [ DriversLicense(main=true) ]
8  {
9      _id      : int        [ DriversLicense.id ]
10     fNumber  : string     [ DriversLicense.number ]
11     fDate    : date       [ DriversLicense.date ]
12 }
13 DocTypeRegistration [ Registration(main=true), Person(main=false)
    , DriversLicense(main=false) ]
14 {
15     _id                      : int        [ ]
16     observation              : string     [ Registration.
    observation ]
17     fPersonId                : int        [ Person.id ]
18     fName                    : string     [ Person.name ]
19     fAddress                 : string     [ Person.address ]
20     data_DriversLicense      :
21     {
22         fDriversLicenseId    : int        [ DriversLicense.id ]
23         fNumber              : int        [ DriversLicense.number ]
24         fDate                : date       [ DriversLicense.date ]
25     }
26 }
```

Listagem A.7 – Mapeamento 1-1, opção 1c

```
1  db.DocTypePerson.find().forEach( function(data) {
2     db.EC.insert( {
```

```
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
    db.DocTypeRegistration.find({ 'fPersonId': data.data_Person._id }).forEach(
    function(data2) {
     varData.push( {
        data_DriversLicense: {
         fDriversLicenseId: data2.data_DriversLicense.fDriversLicenseId,
         fNumber: data2.data_DriversLicense.fNumber,
         fDate: data2.data_DriversLicense.fDate
        },
        data_Registration: {
         observation: data2.observation
        }
   })
  });
  db.EC.update( { 'data_Person._id': data.data_Person._id},
  { $set: {
     'data_Join': varData
  }});
});
```

Listagem A.8 – Consulta em Javascript, Mapeamento 1-1, opção 1c, Query 1

```
db.DocTypeRegistration.find().forEach( function(data) {
    db.EC.insert( {
      data_Person: {
         fPersonId: data.fPersonId
      },
      data_Join: [{
```

```
        data_Registration: {
           observation: data.observation
        },
        data_DriversLicense: {
            fDriversLicenseId: data.data_DriversLicense.
   fDriversLicenseId,
            fNumber: data.data_DriversLicense.fNumber,
            fDate: data.data_DriversLicense.fDate
        }
     }]
    });
});
db.EC.find().forEach( function(data){
    var varData = [];
    db.DocTypePerson.find({ '_id': data.data_Person.fPersonId }).
   forEach(
    function(data2) {
     varData.push( {
        data_Person: {
         fPersonId: data2._id,
         fName: data2.fName,
         fAddress: data2.fAddress,
      }})
    });
    db.EC.update( {'data_Person.fPersonId': data.data_Person.
   fPersonId},
    { $set: {
       'data_Person': varData[0].data_Person
    }})
});
```

Listagem A.9 – Consulta em Javascript, Mapeamento 1-1, opção 1c, Query 2

```
DocTypePerson [ Person(main=true), Registration(main=false),
   DriversLicense(main=false)  ]
{
    _id                 : int       [ Person.id ]
    fName               : string    [ Person.name ]
    fAddress            : string    [ Person.address ]
    observation         : string    [ Registration.observation ]
    data_DriversLicense :
  {
        fDate           : date      [ DriversLicense.date ]
```

```
  }
}
DocTypeDriversLicense [ DriversLicense(main=true) ]
{
    _id     : int       [ DriversLicense.id ]
    fNumber : string    [ DriversLicense.number ]
    fDate   : date      [ DriversLicense.date ]
}
DocTypeRegistration [ Registration(main=true) ]
{
    _id          : int       []
    observation : string    [ Registration.observation ]
}
```

Listagem A.10 – Mapeamento 1-1, opção 2a

A execução do algoritmo para o mapeamento da Listagem A.10, mostrou como resultado a consulta da Listagem A.11 seguida da mensagem: "impossível join entre DriversLicense via Person-Registration-DriversLicense e DocTypeDriversLicense, pois não há atributos id comuns entre os document types".

```
db.DocTypePerson.find().forEach( function(data) {
  db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: [{
        data_Registration: {
            observation: data.observation
        },
        data_DriversLicense: {
            fDate: data.data_DriversLicense.fDate
        }
      }]
  });
});
```

Listagem A.11 – Consulta em Javascript, Mapeamento 1-1, opção 2a

```
DocTypePerson [ Person(main=true), Registration(main=false),
   DriversLicense(main=false) ]
{
```

```
    _id                 : int       [ Person.id ]
    fName               : string    [ Person.name ]
    fAddress            : string    [ Person.address ]
    observation         : string    [ Registration.observation ]
    fDriversLicenseId   : int       [ DriversLicense.id ]
    fNumber             : int       [ DriversLicense.number ]
    fDate               : date      [ DriversLicense.date ]
}
DocTypeDriversLicense [ DriversLicense(main=true) ]
{
    _id     : int       [ DriversLicense.id ]
    fNumber : string    [ DriversLicense.number ]
    fDate   : date      [ DriversLicense.date ]
}
DocTypeRegistration [ Registration(main=true) ]
{
    _id         : int       []
    observation : string    [ Registration.observation ]
}
```

Listagem A.12 – Mapeamento 1-1, opção 2b

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: [{
          data_Registration: {
               observation: data.observation
          },
          data_DriversLicense: {
               fDriversLicenseId: data.fDriversLicenseId,
               fNumber: data.fNumber,
               fDate: data.fDate
          }
      }]
   });
});
```

Listagem A.13 – Consulta em Javascript, Mapeamento 1-1, opção 2b

```
DocTypePerson [ Person(main=true), Registration(main=false),
   DriversLicense(main=false) ]
{
    _id                 : int       [ Person.id ]
    fName               : string    [ Person.name ]
    fAddress            : string    [ Person.address ]
    observation         : string    [ Registration.observation ]
    fDriversLicenseId   : int       [ DriversLicense.id ]
    fNumber             : int       [ DriversLicense.number ]
    fDate               : date      [ DriversLicense.date ]
}
DocTypeDriversLicense [ DriversLicense(main=true) ]
{
    _id     : int       [ DriversLicense.id ]
    fNumber : string    [ DriversLicense.number ]
    fDate   : date      [ DriversLicense.date ]
}
DocTypeRegistration [ Registration(main=true) ]
{
    _id         : int       []
    observation : string    [ Registration.observation ]
}
```

Listagem A.14 – Mapeamento 1-1, opção 2c

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         fPersonId: data.data_Person.fPersonId,
         fName: data.data_Person.fName,
         fAddress: data.data_Person.fAddress
      },
      data_Join: [{
            data_Registration: {
                fObservation: data.fObservation
            },
            data_DriversLicense: {
                _id: data._id,
                fNumber: data.fNumber,
                fDate: data.fDate
            }
      }]
    });
```

```
});
```

Listagem A.15 – Consulta em Javascript, Mapeamento 1-1, opção 2c

```
DocTypePerson [ Person(main=true)]
{
    _id                 : int       [ Person.id ]
    fName               : string    [ Person.name ]
    fAddress            : string    [ Person.address ]
}
DocTypeDriversLicense [ DriversLicense(main=true), Registration(
   main=false), Person(main=false)  ]
{
    _id         : int       [ DriversLicense.id ]
    fNumber     : string    [ DriversLicense.number ]
    fDate       : date      [ DriversLicense.date ]
    observation: string     [ Registration.observation ]
    data_Person:
    {
        fPersonId: int          [ Person.id ]
        fName    : string       [ Person.number ]
        fAddress : string       [ Person.date ]
    }
}
DocTypeRegistration [ Registration(main=true) ]
{
    _id         : int       []
    observation : string    [ Registration.observation ]
}
```

Listagem A.16 – Mapeamento 1-1, opção 2d

A execução do algoritmo para o mapeamento da Listagem A.16, retorna duas consultas, Listagem A.17 e A.18 . A primeira A.17 mostra as seguintes mensagens “impossível join entre Person e Registration via Person-Registration-DriversLicense e DocTypeRegistration, pois não há atributos id comuns entre os document types” e “impossível pois não há entidade computada válida”.

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress,
      },
      data_Join: []
    });
});
```

Listagem A.17 – Consulta em Javascript, Mapeamento 1-1, opção 2d, Query 1

```
db.DocTypeDriversLicense.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         fPersonId: data.fPersonId,
         fName: data.fName,
         fAddress: data.fAddress
      },
      data_Join: [{
        data_Registration: {
            observation: data.observation
        },
        data_DriversLicense: {
            _id: data._id,
            fNumber: data.fNumber,
            fDate: data.fDate
        }
      }]
   });
});
```

Listagem A.18 – Consulta em Javascript, Mapeamento 1-1, opção 2d, Query 2

```
DocTypePerson [ Person(main=true), DriversLicense(main=false) ]
{
    _id                     : int       [ Person.id ]
    fName                   : string    [ Person.name ]
    fAddress                : string    [ Person.address ]
    data_DriversLicense     :
  {
        fDriversLicenseId   : int       [ DriversLicense.id ]
        fNumber             : int       [ DriversLicense.number ]
```

```
        fDate                : date      [ DriversLicense.date ]
  }
}
DocTypeDriversLicense [ DriversLicense(main=true), Registration(
   main=false) ]
{
    _id          : int       [ DriversLicense.id ]
    fNumber      : string    [ DriversLicense.number ]
    fDate        : date      [ DriversLicense.date ]
    observation  : string    [ Registration.observation ]
}

DocTypeRegistration [ Registration(main=true) ]
{
    _id          : int       []
    observation  : string    [ Registration.observation ]
}
```

Listagem A.19 – Mapeamento 1-1, opção 2e

A execução do algoritmo para o mapeamento da Listagem A.19, retorna a consulta da Listagem A.20. A consulta mostra as seguintes mensagens "impossível join entre Person e Registration via Person-Registration-DriversLicense e DocTypeRegistration, pois não há atributos id comuns entre os document types" e "impossível pois não há entidade computada válida".

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress
      },
       data_Join: [{
            data_DriversLicense: {
                fDriversLicenseId: data.data_DriversLicense.
   fDriversLicenseId,
                fNumber: data.data_DriversLicense.fNumber,
                fDate: data.data_DriversLicense.fDate
            }
        }]
    });
```

```
16 });
```

Listagem A.20 – Consulta em Javascript, Mapeamento 1-1, opção 2e

A.2 Mapeamentos com cardinalidade 1-N

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade 1-N, sendo *Person* igual a Entidade 1, *Car* a Entidade 2 e *Drives* o relacionamento.

```
Entity person = (Entity) mm.getERModel().findERElement("Person");
Relationship drives = (Relationship) mm.getERModel().findERElement("Drives");
Entity car = (Entity) mm.getERModel().findERElement("Car");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(person.getAttribute("name"));
queryAttributes.add(car.getAttribute("plate"));
queryAttributes.add(car.getAttribute("color"));

List<Query> queries = ma.binaryJoin(person, drives, car, queryAttributes);
```

Figura 25 – Lista queryAttributes, Modelo ER Parent, mapeamento 1-N

```
1  DocTypePerson [ Person (main=true)]
2  {
3     _id       : int [ Person.id ]
4     fName     : string [ Person.name ]
5     fAddress  : string [ Person.address ]
6  }
7  DocTypeCar [ Car(main=true), Person(main=false), Drives(main=true
      )]
8  {
9     _id           : int [ Car.id ]
10    fPlate        : string [ Car.plate ]
11    fColor        : string [ Car.color ]
12    fPersonId     : int [ Person.id ]
13    fName         : string [ Person.name ]
14    fAddress      : string [ Person.address ]
15    fObservation  : string [Drives.observation]
16 }
```

Listagem A.21 – Mapeamento 1-N, opção OneToMany1b

```
1  db.DocTypePerson.find().forEach( function(data) {
2     db.EC.insert( {
3        data_Person: {
4           _id: data._id,
```

```
          fName: data.fName,
          fAddress: data.fAddress
       },
       data_Join: []
    });
});
db.EC.find().forEach( function(data){
    var varData = [];
    var varData2 = [];
    data.data_Join.forEach(
        function(dataCopy) {
              varData.push(dataCopy);
        });
    db.DocTypeCar.find({ 'fPersonId': data.data_Person._id }).
   forEach(
     function(data2) {
     varData.push( {
        data_Drives: {
            fObservation: data2.fObservation
        },
        data_Car: {
            _id: data2._id,
            fPlate: data2.fPlate,
            fColor: data2.fColor
        }
     });
  });
  db.EC.update( {'data_Person._id': data.data_Person._id},
  { $set: {
    'data_Join': varData
  }});
});
```

Listagem A.22 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1b, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem A.21 pode não retornar todos os dados, a Entidade 1: Person não é principal no documento: DocTypeCar.

```
db.DocTypeCar.find().forEach( function(data) {
   db.EC1.insert( {
      data_Person: {
```

```
        fPersonId: data.fPersonId,
        fName: data.fName,
        fAddress: data.fAddress
      },
      data_Join: []
    });
    db.EC1.createIndex({ 'data_Person.fPersonId': 1 }, {unique:
   true } );
    db.EC1.update( {'data_Person.fPersonId':data.fPersonId},
    {$addToSet: {
        'data_Join': {
            data_Drives: {
                fObservation: data.fObservation
            },
            data_Car: {
                _id: data._id,
                fPlate: data.fPlate,
                fColor: data.fColor
            }
        }
     }});
});
```

Listagem A.23 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1b, Query 2

```
DocTypePerson [ Person (main=true)]
{
   _id      : int [ Person.id ]
   fName    : string [ Person.name ]
   fAddress : string [ Person.address ]
}
DocTypeCar [ Car(main=true), Person(main=false), Drives(main=true
   )]
{
   _id          : int [ Car.id ]
   fPlate       : string [ Car.plate ]
   fColor       : string [ Car.color ]
   fObservation : string [Drives.observation]
   data_Person:
  {
     fPersonId : int [ Person.id ]
  }
```

```
}

```

Listagem A.24 – Mapeamento 1-N, opção OneToMany1c

```
db.DocTypePerson.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id,
         fName: data.fName,
         fAddress: data.fAddress,
      },
      data_Join: []
    });
});
db.EC.find().forEach( function(data){
     var varData = [];
     var varData2 = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
     db.DocTypeCar.find({ 'data_Person.fPersonId': data.
   data_Person._id }).forEach(
     function(data2) {
     varData.push( {
        data_Drives: {
                fObservation: data2.fObservation
        },
        data_Car: {
                _id: data2._id,
                fPlate: data2.fPlate,
                fColor: data2.fColor
        }
     });
    });
    db.EC.update( {'data_Person._id': data.data_Person._id},
    { $set: {
        'data_Join': varData
    }});
});
```

Listagem A.25 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1c, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem A.24 pode não retornar todos os dados, a Entidade 1: Person não é principal no documento: DocTypeCar.

```
db.DocTypeCar.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         fPersonId: data.data_Person.fPersonId,
      },
      data_Join: []
    });
    db.EC.createIndex({ 'data_Person.fPersonId': 1 }, {unique:
   true } );
    db.EC.update( {'data_Person.fPersonId':data.data_Person.
   fPersonId},
   {$addToSet: {
        'data_Join': {
            data_Drives: {
                fObservation: data.fObservation
            },
            data_Car: {
                _id: data._id,
                fPlate: data.fPlate,
                fColor: data.fColor
            }
        }
    }});
});
db.EC.find().forEach( function(data){
     var varData = [];
     db.DocTypePerson.find({ '_id': data.data_Person.fPersonId })
   .forEach(
     function(data2) {
     varData.push( {
        data_Person: {
         fPersonId: data2._id,
         fName: data2.fName,
         fAddress: data2.fAddress,
      }})}) ;
    db.EC.updateMany( {'data_Person.fPersonId': data.data_Person.
   fPersonId},
    { $set: {
```

```
35          'data_Person': varData[0].data_Person
36      }})
37 });
```

Listagem A.26 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1c, Query 2

```
1  DocTypePerson [ Person (main=true)]
2  {
3     _id      : int [ Person.id ]
4     fName    : string [ Person.name ]
5     fAddress : string [ Person.address ]
6  }
7  DocTypeCar [ Car (main=true) ]
8  {
9     _id    : int [ Car.id ]
10    fPlate : string [ Car.plate ]
11    fColor : string [ Car.color ]
12 }
13 DocTypeDrives [ Drives (main=true), Person (main=false), Car (
      main=false)] ]
14 {
15    fPersonId : int [ Person.id ]
16    fName : string [ Person.name ]
17    fAddress : string [ Person.address ]
18    fObservation: string [Drives.observation]
19    data_Car:
20   [
21     {
22       fCarId : int [ Car.id ]
23       fPlate : string [ Car.plate ]
24       fColor : string [ Car.color ]
25     }
26   ]
27 }
```

Listagem A.27 – Mapeamento 1-N, opção OneToMany1d

```
1  db.DocTypePerson.find().forEach( function(data) {
2     db.EC.insert( {
3        data_Person: {
4           _id: data._id,
5           fName: data.fName,
6           fAddress: data.fAddress
```

```
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
     db.DocTypeDrives.find({ '_id': data.data_Person._id }).
   forEach(
     function(data2) {
      data2.data_Car.forEach(function(data1){
      db.EC.update( {'data_Person._id': data.data_Person._id},
        { $addToSet: {
            'data_Join': {
                data_Drives: {
                    fObservation: data1.fObservation
                },
                data_Car: {
                    fCarId: data1.fCarId,
                    fPlate: data1.fPlate,
                    fColor: data1.fColor
                }
            }
         }});
       });
   });
});
```

Listagem A.28 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1d, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem A.27 pode não retornar todos os dados, a Entidade 1: Person não é principal no documento: DocTypeDrives.

```
db.DocTypeDrives.find().forEach( function(data) {
   db.EC.insert( {
      data_Person: {
         _id: data._id
         fName: data.fName,
```

```
6           fAddress: data.fAddress
7       },
8       data_Join: []});
9       data.data_Car.forEach(function(data1){
10      db.EC.update( {'data_Person._id':data._id},
11      { $addToSet: {
12            'data_Join': {
13                data_Drives: {
14                    fObservation: data1.fObservation
15                },
16                data_Car: {
17                    fCarId: data1.fCarId
18                    fPlate: data1.fPlate,
19                    fColor: data1.fColor
20                }
21          }}
22      });
23    });
24 });
```

Listagem A.29 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1d, Query 2

A.3 Mapeamentos com cardinalidade N-1

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade N-1, sendo *Car* igual a Entidade 1, *Person* a Entidade 2 e *Drives* o relacionamento.

```
Entity car = (Entity) mm.getERModel().findERElement("Car");
Relationship drives = (Relationship) mm.getERModel().findERElement("Drives");
Entity person = (Entity) mm.getERModel().findERElement("Person");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(person.getAttribute("name"));
queryAttributes.add(car.getAttribute("plate"));
queryAttributes.add(car.getAttribute("color"));

List<Query> queries = ma.binaryJoin(car, drives, person, queryAttributes);
```

Figura 26 – Lista queryAttributes, Modelo ER Parent, mapeamento N-1

```
1 DocTypePerson [ Person (main=true), Car (main=true), Drives (main
    =true)] ]
2 {
3    _id      : int [ Person.id ]
```

```
    fName    : string [ Person.name ]
    fAddress : string [ Person.address ]
    data_Car:
     [
         {
             _id          : int [ Car.id ]
             fObservation: string [Drives.observation]
         }
     ]
}
```

Listagem A.30 – Mapeamento N-1, opção ManyToOne1a

A execução do algoritmo para o mapeamento da Listagem A.30, retorna a consulta da Listagem A.31. A consulta mostra as seguintes mensagens: “A consulta pode não retornar todos os dados, a Entidade 1: Car não é principal no documento: DocTypePerson ”, “Não pode recuperar Car.plate, pois não existe um DocumentType (main=true) mapeado a Car” e “Não pode recuperar Car.color, pois não existe um DocumentType (main=true) mapeado a Car”.

```
db.DocTypePerson.find().forEach(function(data) {
data.data_Car.forEach(function(data1){
   db.EC.insert( {
      data_Car: {
         _id: data1._id
      }});
   db.EC.update( {'data_Car._id':data1._id},
   {$set: {
        'data_Join': [{
            data_Drives: {
                fObservation: data1.fObservation
            },
            data_Person: {
                _id: data._id,
                fName: data.fName,
                fAddress: data.fAddress
            }
        }]
     }});
  });
});
```

Listagem A.31 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1a

```
DocTypeCar [ Car(main=true), Person(main=false), Drives(main=true
   )]
{
   _id          : int [ Car.id ]
   fPlate       : string [ Car.plate ]
   fColor       : string [ Car.color ]
   fPersonId    : int [ Person.id ]
   fName        : string [ Person.name ]
   fAddress     : string [ Person.address ]
   fObservation : string [Drives.observation]
}
DocTypePerson [ Person (main=true)]
{
   _id      : int [ Person.id ]
   fName    : string [ Person.name ]
   fAddress : string [ Person.address ]
}
```

Listagem A.32 – Mapeamento N-1, opção ManyToOne1b

```
db.DocTypeCar.find().forEach(function(data) {
   db.EC.insert({
      data_Car: {
         _id: data._id,
         fPlate: data.fPlate,
         fColor: data.fColor
      },
      data_Join: [{
        data_Drives: {
            fObservation: data.fObservation
        },
        data_Person: {
            fPersonId: data.fPersonId,
      fName: data.fName,
      fAddress: data.fAddress
        }
      }]
   });
});
```

Listagem A.33 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1b

```
DocTypeCar [ Car(main=true), Person(main=false), Drives(main=true
   )]
```

```
{
   _id          : int [ Car.id ]
   fPlate       : string [ Car.plate ]
   fColor       : string [ Car.color ]
   fObservation : string [Drives.observation]
   data_Person:
  {
     fPersonId : int [ Person.id ]
  }
}
DocTypePerson [ Person (main=true)]
{
   _id      : int [ Person.id ]
   fName    : string [ Person.name ]
   fAddress : string [ Person.address ]
}


```

Listagem A.34 – Mapeamento N-1, opção ManyToOne1c

```
db.DocTypeCar.find().forEach(function(data) {
   db.EC.insert({
      data_Car: {
         _id: data._id,
         fPlate: data.fPlate,
         fColor: data.fColor
      },
      data_Join: [{
        data_Drives: {
            fObservation: data.fObservation
        },
        data_Person: {
            fPersonId: data.data_Person.fPersonId
        }
      }]
   });
});
db.EC.find().forEach( function(data){
 var novoArray = [];
    data.data_Join.forEach(function(data2) {
    data2.data_Person = db.DocTypePerson.findOne({'_id': data2.
   data_Person.fPersonId });
```

```
22          novoArray.push(data2);
23      });
24      db.EC.update( {'data_Car._id': data.data_Car._id},
25      { $set: {
26          'data_Join': novoArray
27      }});
28  });
```

Listagem A.35 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1c

```
1  DocTypeCar [ Car (main=true) ]
2  {
3     _id    : int [ Car.id ]
4     fPlate : string [ Car.plate ]
5     fColor : string [ Car.color ]
6  }
7  DocTypeDrives [ Drives (main=true), Person (main=false), Car (
       main=false)] ]
8  {
9     fPersonId : int [ Person.id ]
10    fName : string [ Person.name ]
11    fAddress : string [ Person.address ]
12    fObservation: string [Drives.observation]
13    data_Car:
14   [
15     {
16       fCarId : int [ Car.id ]
17       fPlate : string [ Car.plate ]
18       fColor : string [ Car.color ]
19     }
20   ]
21 }
22 DocTypePerson [ Person (main=true)]
23 {
24    _id      : int [ Person.id ]
25    fName    : string [ Person.name ]
26    fAddress : string [ Person.address ]
27 }
```

Listagem A.36 – Mapeamento N-1, opção ManyToOne1d

```
1  db.DocTypeCar.find().forEach(function(data){
2     db.EC.insert({
3        data_Car: {
```

```
         _id: data._id,
         fPlate: data.fPlate,
         fColor: data.fColor
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
 var varData = [];
 data.data_Join.forEach(
     function(dataCopy) {
          varData.push(dataCopy);
 });
 db.DocTypeDrives.find().forEach(
     function(data1) {
          data1.data_Car.forEach(function(data2){
              db.EC.update({'data_Car._id': data2.fCarId},
              { $addToSet: {
                'data_Join': {
                  data_Drives: {
                       fObservation: data2.fObservation
                  },
                  data_Person: {
                       _id: data1._id,
                       fName: data1.fName,
                       fAddress: data1.fAddress
                  }
                }
             }});
         });
     });
});
```

Listagem A.37 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1d, Query 1

```
db.DocTypeDrives.find().forEach( function(data) {
 data.data_Car.forEach(function(data1){
   db.EC.insert( {
      data_Car: {
         fCarId: data1.fCarId,
         fPlate: data1.fPlate,
         fColor: data1.fColor
```

```
      }});
   db.EC.update( {'data_Car.fCarId':data1.fCarId},
   {$set: {
        'data_Join': [{
            data_Drives: {
                fObservation: data1.fObservation
            },
            data_Person: {
                _id: data._id,
                fName: data.fName,
                fAddress: data.fAddress
            }
         }]
    }});
 });
});
```

Listagem A.38 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1d, Query 2

A.4 Mapeamentos com cardinalidade N-N

```
DocTypeCar [ Car(main=true) ]
{
   _id    : int [ Car.id ]
   fPlate : string [ Car.plate ]
   fColor : string [ Car.color ]
}
DocTypeGarage [ Garage(main=true) ]
{
    id      : int [ Garage.id ]
    fName   : string [Garage.name]
    fAddress: string [Garage.address]
}
DocTypeRepairs [Repairs(main=true), Car(main=false), Garage(main=
   false)]
{
    fDate    : date [ Repairs.date ]
    fFix     : string [ Repairs.fix ]
    fCarId   : int [ Car.id ]
    fGarageId: int [ Garage.id ]
```

```
}
```

Listagem A.39 – Mapeamento N-N, opção ManyToMany1a

```
db.DocTypeCar.find().forEach(function(data){
   db.EC.insert({
      data_Car: {
         _id: data._id,
         fPlate: data.fPlate,
         fColor: data.fColor
      },
      data_Join: []
    });
});
db.EC.find().forEach( function(data){
    var varData = [];
    data.data_Join.forEach(
     function(dataCopy) {
       varData.push(dataCopy);
    });
    db.DocTypeRepairs.find({'fCarId': data.data_Car._id }).
   forEach(
    function(data2){
     varData.push({
        data_Garage: {
          fGarageId: data2.fGarageId
        },
        data_Repairs: {
          fDate: data2.fDate,
          fFix: data2.fFix
        }
   })
    });
    db.EC.update({'data_Car._id': data.data_Car._id},
    { $set:{
      'data_Join': varData
    }});
});
db.EC.find().forEach( function(data){
    var novoArray = [];
    data.data_Join.forEach(function(data2) {
    data2.data_Garage = db.DocTypeGarage.findOne({'_id': data2.
   data_Garage.fGarageId });
```

```
38      novoArray.push(data2);
39    });
40    db.EC.update({'data_Car._id': data.data_Car._id},
41    { $set: {
42      'data_Join': novoArray
43    }});
44 });
```

Listagem A.40 – Consulta em Javascript, Mapeamento N-N, opção ManyToMany1a, Query 1

```
1  db.DocTypeRepairs.find().forEach(function(data){
2     db.EC.insert({
3        data_Car: {
4           fCarId: data.fCarId
5        },
6        data_Join: []
7     });
8     db.EC.createIndex({'data_Car.fCarId': 1 }, {unique: true } );
9     db.EC.update( {'data_Car.fCarId':data.fCarId},
10    {$addToSet: {
11        'data_Join': {
12            data_Repairs: {
13                fDate: data.fDate,
14                fFix: data.fFix
15            },
16            data_Garage: {
17                fGarageId: data.fGarageId
18            }
19        }
20    }});
21 });
22 db.EC.find().forEach( function(data){
23    var varData = [];
24    db.DocTypeCar.find({'_id': data.data_Car.fCarId }).forEach(
      function(data2) {
25       varData.push( {
26         data_Car: {
27          fCarId: data2._id,
28          fPlate: data2.fPlate,
29          fColor: data2.fColor
30       }})
31    });
```

```
32      db.EC.updateMany({‘data_Car.fCarId’: data.data_Car.fCarId},
33      { $set: {
34        ‘data_Car’: varData[0].data_Car
35      }})
36  });
37  db.EC.find().forEach( function(data){
38      var novoArray = [];
39      data.data_Join.forEach(function(data2) {
40      data2.data_Garage = db.DocTypeGarage.findOne({ ’_id’: data2.
    data_Garage.fGarageId });
41        novoArray.push(data2);
42      });
43      db.EC.update({‘data_Car.fCarId’: data.data_Car.fCarId},
44      { $set: {
45        ‘data_Join’: novoArray
46      }});
47  });
```

Listagem A.41 – Consulta em Javascript, Mapeamento N-N, opção ManyToMany1a, Query 2

APÊNDICE B – Testes Modelo ER - ProgradWeb

B.1 Mapeamentos com cardinalidade 1-1

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade 1-1, sendo *Alunograd* igual a Entidade 1, *Endereço* a Entidade 2 e *Mora* o relacionamento.

```
Entity alunograd = (Entity) mm.getERModel().findERElement("Alunograd");
Relationship mora = (Relationship) mm.getERModel().findERElement("Mora");
Entity endereco = (Entity) mm.getERModel().findERElement("Endereco");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(alunograd.getAttribute("nomealu_alug"));
queryAttributes.add(endereco.getAttribute("cod_endereco"));
queryAttributes.add(endereco.getAttribute("nome_cidade"));

List<Query> queries = ma.binaryJoin(alunograd, mora, endereco, queryAttributes);
```

Figura 27 – Lista queryAttributes, Modelo ER ProgradWeb, mapeamento 1-1

```
1  DocTypeAlunograd [ Alunograd(main=true) ]
2  {
3     _id          : int [Alunograd.codalu_alug]
4     fNomealu_alug : string [Alunograd.nomealu_alug]
5     fDatanasc_alug: string [Alunograd.datanasc_alug]
6     fCpf_alug    : string [Alunograd.cpf_alug]
7  }
8  DocTypeEndereco [ Endereco(main=true), Alunograd(main=false) ]
9  {
10     _id        : int [Endereco.cod_endereco]
11     fNome_cidade  : string [Endereco.nome_cidade]
12     fNome_bairro  : sgring [Endereco.nome_bairro]
13     fCep          : string [Endereco.cep]
14     fNome_rua     : string [Endereco.nome_rua]
15     fComplemento  : string [Endereco.complemento]
16     data_Alunograd:
17     {
18         fAlunogradId  : int [Alunograd.codalu_alug]
19         fNomealu_alug : string [Alunograd.nomealu_alug]
20         fDatanasc_alug: string [Alunograd.datanasc_alug]
```

```
        fCpf_alug  : string [Alunograd.cpf_alug]
    }
}
```

Listagem B.1 – Mapeamento 1-1, opção 1a - PrograduWeb

```
db.DocTypeAlunograd.find().forEach(function(data){
   db.EC.insert( {
      data_Alunograd: {
         _id: data._id,
         fNomealu_alug: data.fNomealu_alug,
         fDatanasc_alug: data.fDatanasc_alug,
         fCpf_alug: data.fCpf_alug
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
     db.DocTypeEndereco.find({'data_Alunograd._id': data.
   data_Alunograd._id }).forEach(
     function(data2) {
     varData.push( {
        data_Endereco: {
         _id: data2._id,
         fNome_cidade: data2.fNome_cidade,
         fNome_bairro: data2.fNome_bairro,
         fCep: data2.fCep,
         fNome_rua: data2.fNome_rua,
         fComplemento: data2.fComplemento
      }})
    });
    db.EC.update({'data_Alunograd._id': data.data_Alunograd._id},
    { $set: {
        'data_Join': varData
    }});
});
```

Listagem B.2 – Consulta em Javascript, Mapeamento 1-1, opção 1a - PrograduWeb, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem B.1 pode não retornar todos os dados, a Entidade 1: Alunograd não é principal no documento: DocTypeEndereco.

```
db.DocTypeEndereco.find().forEach( function(data) {
   db.EC.insert( {
      data_Alunograd: {
         _id: data.data_Alunograd._id,
         fNomealu_alug: data.data_Alunograd.fNomealu_alug,
         fDatanasc_alug: data.data_Alunograd.fDatanasc_alug,
         fCpf_alug: data.data_Alunograd.fCpf_alug
      }
      ,data_Join: [{
            data_Endereco: {
                _id: data._id,
                fNome_cidade: data.fNome_cidade,
                fNome_bairro: data.fNome_bairro,
                fCep: data.fCep,
                fNome_rua: data.fNome_rua,
                fComplemento: data.fComplemento
            }
        }]
   });
});
```

Listagem B.3 – Consulta em Javascript, Mapeamento 1-1, opção 1a - ProgradWeb, Query 2

```
DocTypeAlunograd [ Alunograd(main=true)]
{
   _id         : int [Alunograd.codalu_alug]
   fNomealu_alug  : string [Alunograd.nomealu_alug]
   fDatanasc_alug : string [Alunograd.datanasc_alug]
   fCpf_alug      : string [Alunograd.cpf_alug]
}
DocTypeEndereco [ Endereco(main=true), Alunograd(main=false) ]
{
  _id       : int [Endereco.cod_endereco]
  fNome_cidade  : string [Endereco.nome_cidade]
  fNome_bairro  : sgring [Endereco.nome_bairro]
  fCep          : string [Endereco.cep]
  fNome_rua     : string [Endereco.nome_rua]
  fComplemento  : string [Endereco.complemento]
```

```
    fAlunogradId  : int [Alunograd.codalu_alug]

}
```

Listagem B.4 – Mapeamento 1-1, opção 1b - PrograduWeb

```
db.DocTypeAlunograd.find().forEach( function(data) {
   db.EC.insert( {
      data_Alunograd: {
         _id: data._id,
         fNomealu_alug: data.fNomealu_alug,
         fDatanasc_alug: data.fDatanasc_alug,
         fCpf_alug: data.fCpf_alug
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
    db.DocTypeEndereco.find({'fAlunogradId': data.data_Alunograd.
   _id }).forEach(
    function(data2){
     varData.push({
        data_Endereco: {
         _id: data2._id,
         fNome_cidade: data2.fNome_cidade,
         fNome_bairro: data2.fNome_bairro,
         fCep: data2.fCep,
         fNome_rua: data2.fNome_rua,
         fComplemento: data2.fComplemento
     }})
    });
    db.EC.update({'data_Alunograd._id': data.data_Alunograd._id},
    { $set: {
        'data_Join': varData
    }});
});
```

Listagem B.5 – Consulta em Javascript, Mapeamento 1-1, opção 1b - PrograduWeb, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem B.4 pode não retornar todos os dados, a Entidade 1: Alunograd não é principal no documento: DocTypeEndereco.

```
db.DocTypeEndereco.find().forEach(function(data) {
   db.EC.insert( {
      data_Alunograd: {
         fAlunogradId: data.fAlunogradId
      },
      data_Join: [{
        data_Endereco: {
            _id: data._id,
            fNome_cidade: data.fNome_cidade,
            fNome_bairro: data.fNome_bairro,
            fCep: data.fCep,
            fNome_rua: data.fNome_rua,
            fComplemento: data.fComplemento
        }
      }]
   });
});
db.EC.find().forEach(function(data){
    var varData = [];
    db.DocTypeAlunograd.find({'_id': data.data_Alunograd.
   fAlunogradId }).forEach(
    function(data2){
     varData.push({
        data_Alunograd: {
         fAlunogradId: data2._id,
         fNomealu_alug: data2.fNomealu_alug,
         fDatanasc_alug: data2.fDatanasc_alug,
         fCpf_alug: data2.fCpf_alug
      }})
    }) ;
    db.EC.update({'data_Alunograd.fAlunogradId': data.
   data_Alunograd.fAlunogradId},
    { $set: {
        'data_Alunograd': varData[0].data_Alunograd
    }})
});
```

Listagem B.6 – Consulta em Javascript, Mapeamento 1-1, opção 1b - PrograduWeb, Query 2

```
DocTypeAlunograd [ Alunograd(main=true), Endereco(main=false)]
{
    _id       : int [Alunograd.codalu_alug]
    fNomealu_alug : string [Alunograd.nomealu_alug]
    fDatanasc_alug: string [Alunograd.datanasc_alug]
    fCpf_alug   : string [Alunograd.cpf_alug]
    fEnderecoId   : int [Endereco.cod_endereco]
    fNome_cidade  : string [Endereco.nome_cidade]
    fNome_bairro  : sgring [Endereco.nome_bairro]
    fCep          : string [Endereco.cep]
    fNome_rua     : string [Endereco.nome_rua]
    fComplemento  : string [Endereco.complemento]
}
DocTypeEndereco [ Endereco (main=true) ]
{
  _id       : int [Endereco.cod_endereco]
  fNome_cidade  : string [Endereco.nome_cidade]
  fNome_bairro  : sgring [Endereco.nome_bairro]
  fCep          : string [Endereco.cep]
  fNome_rua     : string [Endereco.nome_rua]
  fComplemento  : string [Endereco.complemento]
}
```

Listagem B.7 – Mapeamento 1-1, opção 1c - PrograduWeb

```
db.DocTypeAlunograd.find().forEach( function(data) {
   db.EC.insert( {
      data_Alunograd: {
         _id: data._id,
         fNomealu_alug: data.fNomealu_alug,
         fDatanasc_alug: data.fDatanasc_alug,
         fCpf_alug: data.fCpf_alug
      },
      data_Join: [{
            data_Endereco: {
                fEnderecoId: data.fEnderecoId,
                fNome_cidade: data.fNome_cidade,
                fNome_bairro: data.fNome_bairro,
                fCep: data.fCep,
                fNome_rua: data.fNome_rua,
                fComplemento: data.fComplemento
            }
      }]
```

```
19      });
20 });
```

Listagem B.8 – Consulta em Javascript, Mapeamento 1-1, opção 1c - PrograduWeb

```
1  DocTypeAlunograd [ Alunograd(main=true)]
2  {
3    _id       : int [Alunograd.codalu_alug]
4    fNomealu_alug : string [Alunograd.nomealu_alug]
5    fDatanasc_alug: string [Alunograd.datanasc_alug]
6    fCpf_alug   : string [Alunograd.cpf_alug]
7  }
8  DocTypeEndereco [ Endereco(main=true), Alunograd(main=false) ]
9  {
10   _id       : int [Endereco.cod_endereco]
11   fNome_cidade  : string [Endereco.nome_cidade]
12   fNome_bairro  : sgring [Endereco.nome_bairro]
13   fCep          : string [Endereco.cep]
14   fNome_rua     : string [Endereco.nome_rua]
15   fComplemento  : string [Endereco.complemento]
16   data_Alunograd:
17   {
18     fAlunogradId  : int [Alunograd.codalu_alug]
19     fNomealu_alug : string [Alunograd.nomealu_alug]
20     fDatanasc_alug: string [Alunograd.datanasc_alug]
21     fCpf_alug   : string [Alunograd.cpf_alug]
22   }
23 }
```

Listagem B.9 – Mapeamento 1-1, opção 1d - PrograduWeb

```
1  db.DocTypeAlunograd.find().forEach( function(data) {
2     db.EC.insert( {
3        data_Alunograd: {
4           _id: data._id,
5           fNomealu_alug: data.fNomealu_alug,
6           fDatanasc_alug: data.fDatanasc_alug,
7           fCpf_alug: data.fCpf_alug,
8        },
9        data_Join: []
10    });
11 });
12 db.EC.find().forEach( function(data){
13     var varData = [];
```

```
14     data.data_Join.forEach(
15      function(dataCopy) {
16           varData.push(dataCopy);
17      });
18     db.DocTypeEndereco.find({ 'data_Alunograd.fAlunogradId': data
    .data_Alunograd._id }).forEach(
19     function(data2) {
20         varData.push( {
21             data_Endereco: {
22                  _id: data2._id,
23                  fNome_cidade: data2.fNome_cidade,
24                  fNome_bairro: data2.fNome_bairro,
25                  fCep: data2.fCep,
26                  fNome_rua: data2.fNome_rua,
27                  fComplemento: data2.fComplemento
28             }
29         })
30     });
31     db.EC.update({'data_Alunograd._id': data.data_Alunograd._id},
32         { $set: {
33             'data_Join': varData
34     } } );
35 });
```

Listagem B.10 – Consulta em Javascript, Mapeamento 1-1, opção 1d - ProgradWeb, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem B.9 pode não retornar todos os dados, a Entidade 1: Alunograd não é principal no documento: DocTypeEndereco.

```
1  db.DocTypeEndereco.find().forEach( function(data) {
2     db.EC.insert( {
3        data_Alunograd: {
4           fAlunogradId: data.data_Alunograd.fAlunogradId,
5           fNomealu_alug: data.data_Alunograd.fNomealu_alug,
6           fDatanasc_alug: data.data_Alunograd.fDatanasc_alug,
7           fCpf_alug: data.data_Alunograd.fCpf_alug
8        },
9        data_Join: [{
10            data_Endereco: {
11                _id: data._id,
12                fNome_cidade: data.fNome_cidade,
```

```
                fNome_bairro: data.fNome_bairro,
                fCep: data.fCep,
                fNome_rua: data.fNome_rua,
                fComplemento: data.fComplemento
            }
        }]
    });
});
```

Listagem B.11 – Consulta em Javascript, Mapeamento 1-1, opção 1d - ProgradWeb, Query 2

```
DocTypeAlunograd [ Alunograd(main=true), Endereco(main=true)]
{
  _id       : int [Alunograd.codalu_alug]
  fNomealu_alug : string [Alunograd.nomealu_alug]
  fDatanasc_alug: string [Alunograd.datanasc_alug]
  fCpf_alug   : string [Alunograd.cpf_alug]
  fEnderecoId   : int [Endereco.cod_endereco]
  fNome_cidade  : string [Endereco.nome_cidade]
  fNome_bairro  : sgring [Endereco.nome_bairro]
  fCep          : string [Endereco.cep]
  fNome_rua     : string [Endereco.nome_rua]
  fComplemento  : string [Endereco.complemento]
}
```

Listagem B.12 – Mapeamento 1-1, opção 1e - ProgradWeb

```
db.DocTypeAlunograd.find().forEach( function(data) {
   db.EC.insert( {
      data_alunograd: {
         _id: data._id,
         fNomealu_alug: data.fNomealu_alug,
         fDatanasc_alug: data.fDatanasc_alug,
         fCpf_alug: data.fCpf_alug,
      },
      data_Join: [{
            data_endereco: {
                fEnderecoId: data.fEnderecoId,
                fNome_cidade: data.fNome_cidade,
                fNome_bairro: data.fNome_bairro,
                fCep: data.fCep,
                fNome_rua: data.fNome_rua,
                fComplemento: data.fComplemento
```

```
            }
        }]
    });
});
```

Listagem B.13 – Consulta em Javascript, Mapeamento 1-1, opção 1e - PrograduWeb

B.2 Mapeamentos com cardinalidade 1-N

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade 1-N, sendo *Alunograd* igual a Entidade 1, *Matriculagrad* a Entidade 2 e *Registra* o relacionamento.

```
Entity alunograd = (Entity) mm.getERModel().findERElement("Alunograd");
Relationship registra = (Relationship) mm.getERModel().findERElement("Registra");
Entity matriculagrad = (Entity) mm.getERModel().findERElement("Matriculagrad");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(alunograd.getAttribute("nomealu_alug"));
queryAttributes.add(matriculagrad.getAttribute("anoini_matr"));
queryAttributes.add(matriculagrad.getAttribute("semini_matr"));


List<Query> queries = ma.binaryJoin(alunograd, registra, matriculagrad, queryAttributes);
```

Figura 28 – Lista queryAttributes, Modelo ER PrograduWeb, mapeamento 1-N

```
DocTypeAlunograd [ Alunograd (main=true)]
{
   _id           : int [ Alunograd.codalu_alug ]
   fNomealu_alug : string [ Alunograd.nomealu_alug ]
   fDatanasc_alug: string [ Alunograd.datanasc_alug ]
   fCpf_alug     : string [ Alunograd.cpf_alug]
}
DocTypeMatriculagrad [ Matriculagrad(main=true), Alunograd(main=
   false), Registra(main=true)]
{
    _id           : int [Matriculagrad.codalu_matr]
    fCodenf_matr  : int [Matriculagrad.codenf_matr]
    fAnoini_matr  : string [Matriculagrad.anoini_matr]
    fSemini_matr  : string [Matriculagrad.semini_matr]
    fOpcao_reg    : string [Registra.opcao_reg]
    data_alunograd:
    {
     fcodalu_alug : int [ Alunograd.codalu_alug ]
  }
```

```
}
```

Listagem B.14 – Mapeamento 1-N, opção OneToMany1a - PrograduWeb

```
db.DocTypeAlunograd.find().forEach( function(data) {
   db.EC.insert( {
      data_Alunograd: {
         _id: data._id,
         fNomealu_alug: data.fNomealu_alug,
         fDatanasc_alug: data.fDatanasc_alug,
         fCpf_alug: data.fCpf_alug
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     var varData2 = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
     db.DocTypeMatriculagrad.find({ 'data_Alunograd.
   fCodalu_alugID': data.data_Alunograd._id }).forEach(
     function(data2) {
     varData.push( {
        data_Registra: {
             fOpcao_reg: data2.fOpcao_reg
        },
        data_Matriculagrad: {
             _id: data2._id,
             fCodenf_matr: data2.fCodenf_matr,
             fAnoini_matr: data2.fAnoini_matr,
             fSemini_matr: data2.fSemini_matr
        ,
     });
    });
    db.EC.update( {'data_Alunograd._id': data.data_Alunograd._id
   },
    { $set: {
        'data_Join': varData
    } } );
```

```
});
```

Listagem B.15 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1a - PrograduWeb, Query 1

```
db.DocTypeMatriculagrad.find().forEach( function(data) {
   db.EC.insert( {
      data_Alunograd: {
         fCodalu_alug: data.data_alunograd.fCodalu_alug
      },
      data_Join: []
    });
   db.EC.createIndex({'data_Alunograd.fCodalu_alug': 1 }, {unique
   : true } );
   db.EC.update( {'data_Alunograd.fCodalu_alug':data.
   data_Alunograd.fCodalu_alug},
   {$addToSet: {
        'data_Join': {
            data_Registra: {
                fOpcao_reg: data.fOpcao_reg,
            },
            data_Matriculagrad: {
                _id: data._id,
                fCodenf_matr: data.fCodenf_matr,
                fAnoini_matr: data.fAnoini_matr,
                fSemini_matr: data.fSemini_matr
            }
        }
     }
    });
});
db.EC.find().forEach( function(data){
     var varData = [];
     db.DocTypeAlunograd.find({ '_id': data.data_Alunograd.
   fCodalu_alug }).forEach(
     function(data2) {
     varData.push( {
        data_Alunograd: {
         fCodalu_alug: data2._id,
         fNomealu_alug: data2.fNomealu_alug,
         fDatanasc_alug: data2.fDatanasc_alug,
         fCpf_alug: data2.fCpf_alug,
      }})}) ;
```

```
36     db.EC.updateMany( {'data_Alunograd.fCodalu_alug': data.
   data_Alunograd.fCodalu_alug},
37     { $set: {
38       'data_Alunograd': varData[0].data_Alunograd
39     }})
40 });
```

Listagem B.16 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1a - ProgradWeb, Query 2

```
1  DocTypeAlunograd [ alunograd (main=true)]
2  {
3     _id             : int [ alunograd.codalu_alug ]
4     fNomealu_alug   : string [ alunograd.nomealu_alug ]
5     fDatanasc_alug  : string [ alunograd.datanasc_alug ]
6     fCpf_alug       : string [ alunograd.cpf_alug]
7  }
8  DocTypeMatriculagrad [ matriculagrad (main=true)]
9  {
10    _id           : int [matriculagrad.codalu_matr]
11    fCodenf_matr  : int [matriculagrad.codenf_matr]
12    fAnoini_matr  : string [matriculagrad.anoini_matr]
13    fSemini_matr  : string [matriculagrad.semini_matr]
14    fOpcao_reg    : string [registra.opcao_reg]
15 }
16 DocTypeRegistra[ Registra (main=true), Alunograd (main=false),
   Matriculagrad (main=false)]]
17 {
18    fcodalu_alug      : int [ alunograd.codalu_alug ]
19    data_Matriculagrad :
20   [
21     {
22        fMatriculaId : int [matriculagrad.codalu_matr]
23        fCodenf_matr : int [matriculagrad.codenf_matr]
24        fAnoini_matr : string [matriculagrad.anoini_matr]
25        fSemini_matr : string [matriculagrad.semini_matr]
26        fOpcao_reg   : string [registra.opcao_reg]
27     }
28   ]
29 }
```

Listagem B.17 – Mapeamento 1-N, opção OneToMany1b - ProgradWeb

```
1 db.DocTypeAlunograd.find().forEach( function(data) {
```

```
2    db.EC.insert( {
3       data_Alunograd: {
4          _id: data._id,
5          fNomealu_alug: data.fNomealu_alug,
6          fDatanasc_alug: data.fDatanasc_alug,
7          fCpf_alug: data.fCpf_alug
8       },
9       data_Join: []
10   });
11 });
12 db.EC.find().forEach( function(data){
13      var varData = [];
14      data.data_Join.forEach(
15          function(dataCopy) {
16               varData.push(dataCopy);
17          });
18      db.DocTypeRegistra.find({'fcodalu_alug': data.data_Alunograd
    ._id }).forEach(
19      function(data1) {
20       data1.data_Matriculagrad.forEach(function(data2){
21       db.EC.update( {'data_Alunograd._id': data.data_Alunograd.
    _id},
22            { $addToSet: {
23                'data_Join': {
24                        data_Registra: {
25                            fOpcao_reg: data2.fOpcao_reg
26                        },
27                        data_Matriculagrad: {
28                            fMatricula: data2.fMatricula,
29                            fCodenf_matr: data2.fCodenf_matr,
30                            fAnoini_matr: data2.fAnoini_matr,
31                            fSemini_matr: data2.fSemini_matr
32                        }
33                    }
34               }});
35        });
36   });
37 });
```

Listagem B.18 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1b - PrograduWeb, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da

Listagem B.17 pode não retornar todos os dados, a Entidade 1: Alunograd não é principal no documento: DocTypeRegistra.

```
db.DocTypeRegistra.find().forEach( function(data) {
   db.EC.insert( {
      data_Alunograd: {
         fcodalu_alug: data.fcodalu_alug
      },
      data_Join: []
  });
    data.data_Matriculagrad.forEach(function(data1){
    db.EC.update( {'data_Alunograd.fcodalu_alug':data.
   fcodalu_alug},
    { $addToSet: {
      'data_Join': {
        data_Registra: {
          fOpcao_reg: data1.fOpcao_reg
        },
          data_Matriculagrad: {
          fMatricula: data1.fMatricula,
          fCodenf_matr: data1.fCodenf_matr,
          fAnoini_matr: data1.fAnoini_matr,
          fSemini_matr: data1.fSemini_matr
        }
      }}
    });
    });
});
db.EC.find().forEach( function(data){
    var varData = [];
    db.DocTypeAlunograd.find({ '_id': data.data_Alunograd.
   fcodalu_alug }).forEach(
    function(data2) {
     varData.push( {
        data_Alunograd: {
         fcodalu_alug: data2._id,
         fNomealu_alug: data2.fNomealu_alug,
         fDatanasc_alug: data2.fDatanasc_alug,
         fCpf_alug: data2.fCpf_alug
      }})
    }) ;
    db.EC.updateMany( {'data_Alunograd.fcodalu_alug': data.
```

```
   data_Alunograd.fcodalu_alug},
38    { $set: {
39        'data_Alunograd': varData[0].data_Alunograd
40    }})
41 });
```

Listagem B.19 – Consulta em Javascript, Mapeamento 1-N, opção OneToMany1b - PrograduWeb, Query 2

B.3 Mapeamentos com cardinalidade N-1

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade N-1, sendo *Matriculagrad* igual a Entidade 1, *Alunograd* a Entidade 2 e *Registra* o relacionamento.

```
Entity matriculagrad = (Entity) mm.getERModel().findERElement("matriculagrad");
Relationship registra = (Relationship) mm.getERModel().findERElement("registra");
Entity alunograd = (Entity) mm.getERModel().findERElement("alunograd");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(alunograd.getAttribute("nomealu_alug"));
queryAttributes.add(matriculagrad.getAttribute("anoini_matr"));
queryAttributes.add(matriculagrad.getAttribute("semini_matr"));


List<Query> queries = ma.binaryJoin(matriculagrad, registra, alunograd, queryAttributes);
```

Figura 29 – Lista queryAttributes, Modelo ER PrograduWeb, mapeamento N-1

```
1  DocTypeMatriculagrad [ Matriculagrad(main=true), Alunograd (main=
       false), Registra(main=true)]
2  {
3     _id           : int [Matriculagrad.codalu_matr]
4     fCodenf_matr  : int [Matriculagrad.codenf_matr]
5     fAnoini_matr  : string [Matriculagrad.anoini_matr]
6     fSemini_matr  : string [Matriculagrad.semini_matr]
7     fOpcao_reg    : string [Registra.opcao_reg]
8     data_Alunograd:
9    {
10       fcodalu_alug : int [ Alunograd.codalu_alug ]
11    }
12 }
13 DocTypeAlunograd [ Alunograd(main=true)]
14 {
15    _id               : int [ Alunograd.codalu_alug ]
16    fNomealu_alug     : string [ Alunograd.nomealu_alug ]
```

```
17    fDatanasc_alug    : string [ Alunograd.datanasc_alug ]
18    fCpf_alug         : string [ Alunograd.cpf_alug]
19 }
```

Listagem B.20 – Mapeamento N-1, opção ManyToOne1a - PrograduWeb

```
1  db.DocTypeMatriculagrad.find().forEach( function(data) {
2   db.EC.insert( {
3    data_Matriculagrad: {
4       _id: data._id,
5       fCodenf_matr: data.fCodenf_matr,
6       fAnoini_matr: data.fAnoini_matr,
7       fSemini_matr: data.fSemini_matr
8    },
9    data_Join: [{
10          data_Registra: {
11              fOpcao_reg: data.fOpcao_reg
12          },
13          data_Alunograd: {
14              fCodalu_alug: data.data_Alunograd.fCodalu_alug
15          }
16      }]
17  });
18 });
19 db.EC.find().forEach( function(data){
20     var novoArray = [];
21     data.data_Join.forEach(function(data2) {
22     data2.data_Alunograd = db.DocTypeAlunograd.findOne({ '_id':
    data2.data_Alunograd.fCodalu_alug });
23       novoArray.push(data2);
24     });
25     db.EC.update( {'data_Matriculagrad._id': data.
    data_Matriculagrad._id},
26     { $set: {
27         'data_Join': novoArray
28     }});
29 });
```

Listagem B.21 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1a - PrograduWeb

```
1 DocTypeMatriculagrad [ Matriculagrad(main=true)]
2 {
3    _id           : int [Matriculagrad.codalu_matr]
```

```
   fCodenf_matr : int [Matriculagrad.codenf_matr]
   fAnoini_matr : string [Matriculagrad.anoini_matr]
   fSemini_matr : string [Matriculagrad.semini_matr]
}
DocTypeRegistra [ Registra(main=true), Alunograd(main=false),
   Matriculagrad(main=false)]]
{
   fAlunogradId         : int [ Alunograd.codalu_alug ]
   data_Matriculagrad   :
  [
    {
       fCodalu_matr : int [Matriculagrad.codalu_matr]
       fOpcao_reg   : string [Registra.opcao_reg]
    }
  ]
}
DocTypeAlunograd [ Alunograd (main=true)]
{
   _id              : int [ Alunograd.codalu_alug ]
   fNomealu_alug    : string [ Alunograd.nomealu_alug ]
   fDatanasc_alug   : string [ Alunograd.datanasc_alug ]
   fCpf_alug        : string [ Alunograd.cpf_alug]
}
```

Listagem B.22 – Mapeamento N-1, opção ManyToOne1b - PrograduWeb

```
db.DocTypeMatriculagrad.find().forEach( function(data) {
   db.EC.insert( {
      data_Matriculagrad: {
         _id: data._id,
         fCodenf_matr: data.fCodenf_matr,
         fAnoini_matr: data.fAnoini_matr,
         fSemini_matr: data.fSemini_matr
      },
      data_Join: []
   });
});
db.EC.find().forEach( function(data){
     var varData = [];
     data.data_Join.forEach(
         function(dataCopy) {
              varData.push(dataCopy);
         });
```

```
18      db.DocTypeRegistra.find().forEach(
19       function(data1) {
20       data1.data_Matriculagrad.forEach(function(data2){
21           db.EC.update( {'data_Matriculagrad._id': data2.
    fCodalu_matr},
22           { $addToSet: {
23             'data_Join': {
24               data_Registra: {
25                    fOpcao_reg: data2.fOpcao_reg
26               },
27               data_Alunograd: {
28                    fAlunogradId: data1.fAlunogradId
29               }
30             }
31          }});
32        });
33    });
34 });
35 db.EC.find().forEach( function(data){
36     var novoArray = [];
37     data.data_Join.forEach(function(data2) {
38     data2.data_Alunograd = db.DocTypeAlunograd.findOne({ '_id':
    data2.data_Alunograd.fAlunogradId });
39         novoArray.push(data2);
40     });
41     db.EC.update( {'data_Matriculagrad._id': data.
    data_Matriculagrad._id},
42     { $set: {
43         'data_Join': novoArray
44     } } );
45 });
```

Listagem B.23 – Consulta em Javascript, Mapeamento N-1, opção ManyToOne1b - ProgradWeb, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem B.22 pode não retornar todos os dados, a Entidade 1: Matriculagrad não é principal no documento: DocTypeRegistra.

```
1 db.DocTypeRegistra.find().forEach( function(data) {
2 data.data_Matriculagrad.forEach(function(data1){
3    db.EC.insert( {
4       data_matriculagrad: {
```

```
5           fCodalu_matr: data1.fCodalu_matr
6     }});
7     db.EC.update( {'data_Matriculagrad.fCodalu_matr':data1.
      fCodalu_matr},
8     {$set: {
9           'data_Join': [{
10              data_Registra: {
11                  fOpcao_reg: data1.fOpcao_reg
12              },
13              data_Alunograd: {
14                  fAlunogradId: data.fAlunogradId
15              }
16          }]
17       }});
18      });
19  });
20  db.EC.find().forEach( function(data){
21      var varData = [];
22      db.DocTypeMatriculagrad.find({ '_id': data.data_Matriculagrad
      .fCodalu_matr }).forEach(
23      function(data2) {
24       varData.push( {
25          data_Matriculagrad: {
26           fCodalu_matr: data2._id,
27           fCodenf_matr: data2.fCodenf_matr,
28           fAnoini_matr: data2.fAnoini_matr,
29           fSemini_matr: data2.fSemini_matr
30        }})
31      }) ;
32      db.EC.update( {'data_Matriculagrad.fCodalu_matr': data.
      data_Matriculagrad.fCodalu_matr},
33      { $set: {
34          'data_Matriculagrad': varData[0].data_Matriculagrad
35      }})
36  });
37  db.EC.find().forEach( function(data){
38      var novoArray = [];
39      data.data_Join.forEach(function(data2) {
40      data2.data_Alunograd = db.DocTypeAlunograd.findOne({ '_id':
      data2.data_Alunograd.fAlunogradId });
41          novoArray.push(data2);
42      });
```

```
43      db.EC.update( {'data_Matriculagrad.fCodalu_matr': data.
    data_Matriculagrad.fCodalu_matr},
44      { $set: {
45          'data_Join': novoArray
46      } } );
47  });
```

Listagem B.24 – Consulta em Javascript, Mapeamento 1-N, opção ManyToOne1b - PrograduateWeb, Query 2

B.4 Mapeamentos com cardinalidade N-N

A seguir é apresentada a consulta expressada na linguagem Java, com os quais foram feitos os testes para mapeamentos com cardinalidade N-N, sendo *Disciplinagrad* igual a Entidade 1, *Enfasegrad* a Entidade 2 e *Gradegrad* o relacionamento.

```
Entity disciplinagrad = (Entity) mm.getERModel().findERElement("Disciplinagrad");
Relationship gradegrad = (Relationship) mm.getERModel().findERElement("Gradegrad");
Entity enfasegrad = (Entity) mm.getERModel().findERElement("Enfasegrad");

List<Attribute> queryAttributes = new ArrayList<>();
queryAttributes.add(disciplinagrad.getAttribute("nome_discip"));
queryAttributes.add(gradegrad.getAttribute("perfil_grd"));
queryAttributes.add(enfasegrad.getAttribute("nomeenf_enf"));


List<Query> queries = ma.binaryJoin(disciplinagrad, gradegrad, enfasegrad, queryAttributes);
```

Figura 30 – Lista queryAttributes, Modelo ER PrograduateWeb, mapeamento N-N

```
1  DocTypeDisciplinagrad [ Disciplinagrad(main=true), Enfasegrad (
       main=false), Gradegrad (main=true)]
2  {
3      _id      : string [ Disciplinagrad.coddiscip_discip ]
4      fNome_discip : string [ Disciplinagrad.nome_discip ]
5      fSetor_discip: string [ Disciplinagrad.setor_discip ]
6      data_Enfasegrad:
7      [
8       {
9           fCodenf_enf  : int [ Enfasegrad.codenf_enf ]
10          fSiglaenf_enf: string [Enfasegrad.siglaenf_enf]
11          fNomeenf_enf : string [Enfasegrad.nomeenf_enf]
12          perfil_grd   : date [ Gradegrad.perfil_grd ]
13          userid_grd   : string [ Gradegrad.userid_grd ]
14          datatu_grd   : string [ Gradegrad.datatu_grd ]
15      }
```

```
16    ]
17 }
18 DocTypeEnfasegrad [ Enfasegrad(main=true), Disciplinagrad(main=
     false), Gradegrad(main=true)]
19 {
20     id         : int [ Enfasegrad.codenf_enf ]
21     fSiglaenf_enf : string [Enfasegrad.siglaenf_enf]
22     fNomeenf_enf  : string [Enfasegrad.nomeenf_enf]
23   data_Disciplinagrad:
24   [
25     {
26         fCoddiscip_discip: string [Disciplinagrad.
     coddiscip_discip]
27         fNome_discip     : string [ Disciplinagrad.nome_discip ]
28         fSetor_discip    : string [ Disciplinagrad.setor_discip ]
29         perfil_grd     : date [ Gradegrad.perfil_grd ]
30         userid_grd     : string [ Gradegrad.userid_grd ]
31         datatu_grd     : string [ Gradegrad.datatu_grd ]
32       }
33   ]
34 }
35 DocTypeGradegrad [Gradegrad(main=true))]
36 {
37     perfil_grd: date [ Gradegrad.perfil_grd ]
38     userid_grd: string [ Gradegrad.userid_grd ]
39     datatu_grd: string [ Gradegrad.datatu_grd ]
40 }
```

Listagem B.25 – Mapeamento N-N, opção ManyToMany1a - PrograduWeb

```
1  db.DocTypeDisciplinagrad.find().forEach(function(data){
2    data.data_Enfasegrad.forEach(function(data1){
3     db.EC.insert( {
4         data_Disciplinagrad: {
5             _id: data._id,
6             fNome_discip: data.fNome_discip,
7             fSetor_discip: data.fSetor_discip
8         },
9         data_Join: []
10    });
11    db.EC.createIndex({'data_Disciplinagrad._id': 1 }, {unique:
     true } );
12    db.EC.update( {'data_Disciplinagrad._id':data._id},
```

```
13    {$addToSet: {
14         'data_Join': {
15             data_Gradegrad: {
16                 perfil_grd: data1.perfil_grd,
17                 userid_grd: data1.userid_grd,
18                 datatu_grd: data1.datatu_grd
19             },
20             data_Enfasegrad: {
21                 fCodenf_enf: data1.fCodenf_enf,
22                 fSiglaenf_enf: data1.fSiglaenf_enf,
23                 fNomeenf_enf: data1.fNomeenf_enf
24             }
25         }
26      }});
27     });
28 });
```

Listagem B.26 – Consulta em Javascript, Mapeamento N-N, opção ManyToMany1a - PrograduWeb, Query 1

A consulta 2, correspondente a execução do algoritmo sobre o mapeamento da Listagem B.25 pode não retornar todos os dados, a Entidade 1: Disciplinagrad não é principal no documento: DocTypeEnfasegrad.

```
1  db.DocTypeEnfasegrad.find().forEach( function(data) {
2    data.data_Disciplinagrad.forEach(function(data1){
3     db.EC.insert( {
4         data_Disciplinagrad: {
5             fCoddiscip_discip: data1.fCoddiscip_discip,
6             fNome_discip: data1.fNome_discip,
7             fSetor_discip: data1.fSetor_discip
8         },
9       data_Join: []
10    });
11   db.EC.createIndex({ 'data_Disciplinagrad.fCoddiscip_discip': 1
     }, {unique: true } );
12   db.EC.update( {'data_Disciplinagrad.fCoddiscip_discip':data1.
     fCoddiscip_discip},
13   {$addToSet: {
14        'data_Join': {
15            data_Gradegrad: {
16                perfil_grd: data1.perfil_grd,
17                userid_grd: data1.userid_grd,
```

```
18                  datatu_grd: data1.datatu_grd
19              },
20              data_Enfasegrad: {
21                  _id: data._id,
22                  fSiglaenf_enf: data.fSiglaenf_enf,
23                  fNomeenf_enf: data.fNomeenf_enf
24              }
25          }
26       }});
27     });
28 });
```

Listagem B.27 – Consulta em Javascript, Mapeamento N-N, opção ManyToMany1a - ProgradWeb, Query 2