João Vitor Lopes Cabral

Uma álgebra ER para consultas em bancos de dados NoSQL: implementação de operadores adicionais e análise de desempenho

Brasil

2020, Março

João Vitor Lopes Cabral

Uma álgebra ER para consultas em bancos de dados NoSQL: implementação de operadores adicionais e análise de desempenho

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: Ricardo Rodrigues Ciferri

Brasil

2020, Março

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 do candidato João Vitor Lopes Cabral, realizada em 09/04/2020:

Prof. Dr. Ricardo Rodrigues Ciferri
UFSCar

Prof. Dr. Valter Vieira de Camargo
UFSCar

Profa. Dra. Cristina Dutra de Aguiar Ciferri
USP

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Valter Vieira de Camargo, Cristina Dutra de Aguiar Ciferri 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. Ricardo Rodrigues Ciferri

Agradecimentos

A Deus por me manter firme nos momentos difíceis.

A minha família por me apoiar e me incentivar a todo momento.

A meu irmão Paulo Henrique pelas conversas que me ajudaram a resolver diversos problemas durante a pesquisa.

Ao professor Dr. Daniel Lúcredio por toda ajuda e orientação.

Ao professor Dr. Ricardo Rodrigues Ciferri pela orientação do trabalho.

Resumo

Bancos de dados e sistemas de bancos de dados são essenciais para a vida moderna. Em um ambiente com crescente volume de dados surgiu a necessidade de se encontrar novas formas de armazenamento e processamento de dados que também sejam capazes de se adequar à rápida evolução da sociedade. Bancos de dados NoSQL são uma alternativa nesse cenário mas a falta de padrão entre as implementações torna algumas tarefas mais trabalhosas do que em bancos de dados SQL, entre elas a escrita de consultas. Para resolver esse problema, este trabalho expande a abordagem proposta por Noguera e Lucrédio (2019), utilizando a linguagem de consulta proposta, incrementa a implementação da operação de junção e implementa as operações *Produto Cartesiano*, *Seleção* e *Projeção* definidas na álgebra proposta por Parent e Spaccapietra (1984). Para tornar mais amigável o processo de criação dos metamodelos foi criada uma representação textual para estes e um "parser" para geração de código compatível com o algoritmo. Foram analisados dois sistemas a fim de validar o código gerado pelo algoritmo e a conformidade do resultado com a álgebra. Também foi analisado o desempenho das consultas geradas pelo algoritmo e comparadas com consultas escritas à mão por um Engenheiro de Software. Neste trabalho mostra-se que a álgebra ER é compatível com bases de dados NoSQL orientada a documentos e que a geração automatizada de consultas não afeta significativamente o desempenho das consultas.

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

Abstract

Databases and database systems are essential to modern life. In a scenario of increasing data handling arose the need to find newer ways to store and process data that are able to cope with the rapid evolution of society. NoSQL databases are an alternative to this scenario but the lack of standard between each database implementation increases the difficulty to perform some tasks when compared to a SQL database. To solve this problem, this paper increments the work of Noguera e Lucrédio (2019), using the query language it improves the join operation and implements the operations *Cartesian Product*, *Selection* and *Projection* that were proposed by Parent e Spaccapietra (1984). In order to make the metamodel creation process more friendly, a textual representation was created for the metamodels and a parser for generating code compatible with the algorithm. Two software systems were analyzed to validate the MongoDB code generated by the algorithm and to check if the query result conforms to the structure defined by the algebra. The query performance was also analyzed, comparing it to the performance of queries that were handcrafted by a Software Engineer. This paper shows that the ER algebra is compatible with document-oriented NoSQL databases and that automated query generation does not significantly affects performance.

Key-words: NoSQL, Databases, Queries, Entity Relationship, Mapping, MongoDB

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*
BD	*Banco de dados*
BASE	*Basically Available, Soft state, Eventual Consistency / Basicamente Disponível, Estado Leve, Consistência eventual*
BQL	*Bridge Query Language*
CAP	*Consistency, Availability, Tolerance of network partition / Consistencia, Disponibilidade, Tolerância a partição da rede*
CMS	*Content Management System*
SBDD	*Sistema de Banco de Dados Distribuído / Distributed Database System*
DDL	*Data Definition Language / Linguagem de Definição de Dados*
DML	*Data Manipulation Language / Linguagem de Manipulação de Dados*
ER	*Entidade Relacionamento*
HTTP	*Hypertext Transfer Protocol / Protocolo de Transferência de Hipertexto*
JSON	*JavaScript Object Notation / Notação de Objetos JavaScript*
MKCMS	*Marketing Content Management System*
MMS	*Model Management System / Sistema Gerenciador de Modelos*
NoSQL	*Not Only Structured Query Language / Não apenas Linguagem de Consulta Estruturada*
PACELC	*Partition Availability and Consistency Else Latency and Consistency / Partição Disponibilidade e Consistência Ou Latência e Consistência*
PDF	*Portable Document Format / Formato Portátil de Documento*

REST	*Representational State Transfer / Transferência de Estado Representacional*
SGBD	*Sistema de Gerenciamento de Banco de Dados*
SOS	*Save Our Systems*
SQL	*Structured Query Language / Linguagem de Consulta Estruturada*
UML	*Unified Modeling Language / Linguagem de Modelagem Unificada*
URI	*Uniform Resource Identifier / Identificador Uniforme de Recurso*
XML	*eXtensible Markup Language / Linguagem de marcação extensible*

Sumário

1 Introdução

Bancos de dados e sistemas de banco de dados são componentes essenciais para a vida em uma sociedade moderna: a maioria de nós realiza diversas atividades todos os dias que interagem com um banco de dados, como ir ao banco para sacar ou depositar fundos, reservar um hotel ou uma passagem aérea (ELMASRI et al., 2010).

Um banco de dados é uma coleção de dados relacionados. Os dados são fatos conhecidos que podem ser registrados e que possuem significado. Por exemplo, nomes, telefones e endereços de pessoas conhecidas (ELMASRI et al., 2010).

Com o crescente volume de dados, surgiu a necessidade de se encontrar novas formas de armazenamento e processamento das informações que possam acompanhar o ritmo em que os dados são adquiridos e com a constante mudança em sua estrutura.

Nesse cenário de constantes mudanças, os bancos de dados NoSQL aparecem como uma opção, permitindo uso de recursos conforme necessário e sem a obrigação de possuir uma estrutura de dados predefinida.

Os bancos de dados NoSQL são flexíveis quanto à estrutura de dados, possuem diferentes formas de armazenamentos de dados, as quais podem ser classificadas em quatro grupos principais: **Chave-valor, família de colunas, documentos e grafos**.

De forma resumida, pode-se dizer o seguinte sobre os tipos de bases de dados NoSQL:

- Chave-valor: são similares a mapas ou dicionários onde os dados são acessados através de uma chave única. Os valores são isolados e independentes entre si, e valores podem ser de qualquer tipo (FUNCK; JABLONSKI, 2011).

- Família de colunas: Os bancos de dados deste tipo são inspirados no Google Bigtable (FUNCK; JABLONSKI, 2011). O Bigtable é definido como um mapa multidimensional ordenado, persistente e distribuído. Os dados são organizados em três dimensões: linhas, colunas e marcadores de tempo. As linhas são agrupadas para forma uma unidade de divisão de carregamento, as colunas são agrupadas para forma a unidade de controle de acesso e histórico de recursos (CHANG et al., 2008).

- Documentos: esse tipo de armazenamento de dados encapsula pares chave-valor em JSON ou documentos similares a JSON. Para cada documento as chaves são únicas e cada documento contém uma chave especial que atua como seu identificador em uma coleção de documentos (FUNCK; JABLONSKI, 2011).

- Grafos: os dados são armazenados como nós em um grafo, e cada ligação com outros nós representa a relação entre eles (DAVOUDIAN; CHEN; LIU, 2018).

Existem mais de 150 implementações de bancos de dados NoSQL (SCAVUZZO; NITTO; CERI, 2014). Não existe um padrão de consultas para os bancos de dados NoSQL. Cada banco de dados possui uma linguagem diferente para consultas, mesmo que sejam do mesmo tipo. Por exemplo, MongoDB[1] e CouchDB[2] são do tipo documentos e possuem linguagens e padrões de consultas diferentes.

Scavuzzo, Nitto e Ceri (2014) dizem que durante o desenvolvimento de uma aplicação novos requisitos ou oportunidades podem surgir. Esses fatores podem levar a necessidade de trocar o tipo de banco de dados utilizado, mas, dadas as significativas diferenças entre cada implementação, isso pode ser problemático em termos de tempo e custos.

1.1 Motivação e Caracterização do Problema

Bancos de dados relacionais consistem de um conjunto de *relações* definidas em certos *atributos*. Uma relação pode ser visualizada como uma *tabela* na qual suas colunas são nomeadas com atributos e suas linhas representam uma tupla (BEERI; BERNSTEIN; GOODMAN, 1989). A descrição de um banco de dados é chamada de esquema, e contém a descrição de todas as relações do banco de dados (BEERI; BERNSTEIN; GOODMAN, 1989)(ELMASRI et al., 2010). O conteúdo de uma relação é chamado de *estado* ou *extensão* do esquema correspondente (BEERI; BERNSTEIN; GOODMAN, 1989).

Já os bancos de dados NoSQL normalmente não seguem um padrão para estruturação de dados, ou seja, nem sempre estão normalizados (SADALAGE; FOWLER, 2013). Esse comportamento permite, caso necessário, que novos registros sejam inseridos mesmo com uma estrutura diferente do resto dos dados, ou seja, permite a adaptação do banco de dados conforme necessidade sem que seja necessário alterar previamente o esquema do banco de dados (SADALAGE; FOWLER, 2013).

Normalização é um processo reversível, executado passo-a-passo que vai substituindo uma coleção de relações por sucessivas coleções onde as relações possuem estrutura progressivamente mais simples e regular. O processo de simplificação é baseado em critérios não-estatísticos. A reversibilidade garante que podemos recuperar a coleção original de relações, portanto, nenhum dado é perdido (CODD, 1971).

De acordo com Codd (1971), os objetivos da normalização são:

- Possibilitar a tabulação de qualquer relação em um banco de dados.

[1] https://www.mongodb.com/
[2] http://couchdb.apache.org/

- Obter uma poderosa capacidade de busca a partir de uma coleção mais simples de operações relacionais.
- Liberar a coleção de relações de dependências indesejadas vindas de inserções, atualizações e deleções.
- Reduzir a necessidade de restruturação da coleção de relações ao introduzir novos tipos de dados.
- Deixar o modelo relacional mais informativo para o usuário.

A Figura 1 mostra o modelo ER de um sistema gerenciador de conteúdo destinado a marketing digital que será referenciado como MKCMS. Com base nesse modelo, será mostrado como os dados podem ser representados em um banco de dados relacional (nesse caso o MySQL) e em um banco de dados NoSQL (MongoDB).

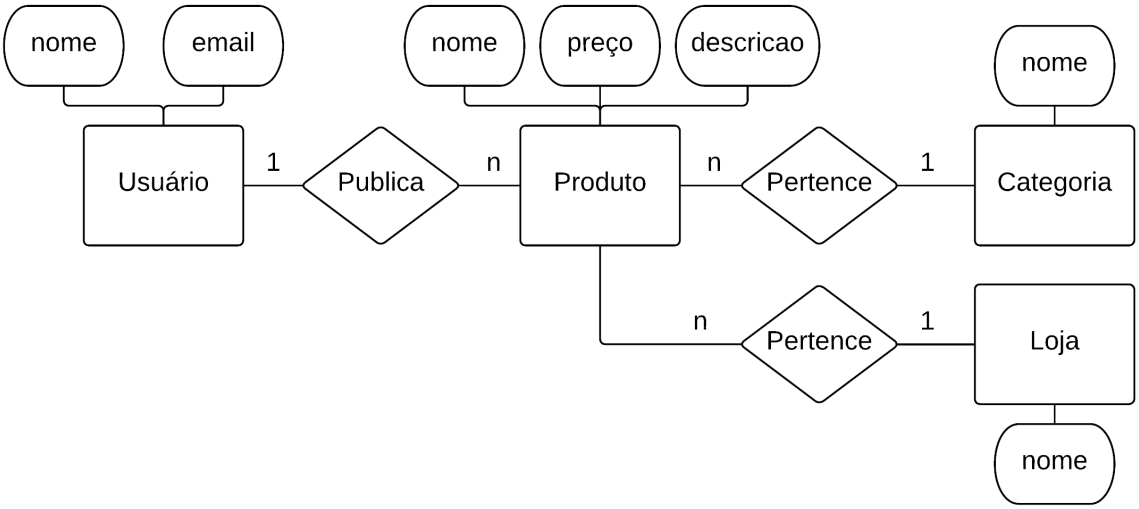

Figura 1 – Modelo ER para o sistema MKCMS

Na Figura 1 estão representadas quatro entidades, chamadas *Usuário*, *Produto*, *Categoria* e *Loja*. O usuário publica um produto que pertence a uma categoria e a uma loja. A Figura ainda mostra que as entidades possuem atributos, tais como nome e email de usuários e nome, preço e descrição de produtos. A entidade *Produto* está relacionada com as entidades *Usuário*, *Categoria* e *Loja*, o relacionamento entre elas é do tipo "um-para-muitos", ou seja, um produto é publicado por um usuário e pertence a uma loja e a uma categoria. Já uma entidade *Usuário* pode publicar muitos produtos, uma loja pode ter vários produtos assim como uma categoria também pode ter vários produtos.

A Figura 2 é um modelo lógico (relacional), correspondente a um possível mapeamento a partir do modelo ER apresentado na Figura 1.

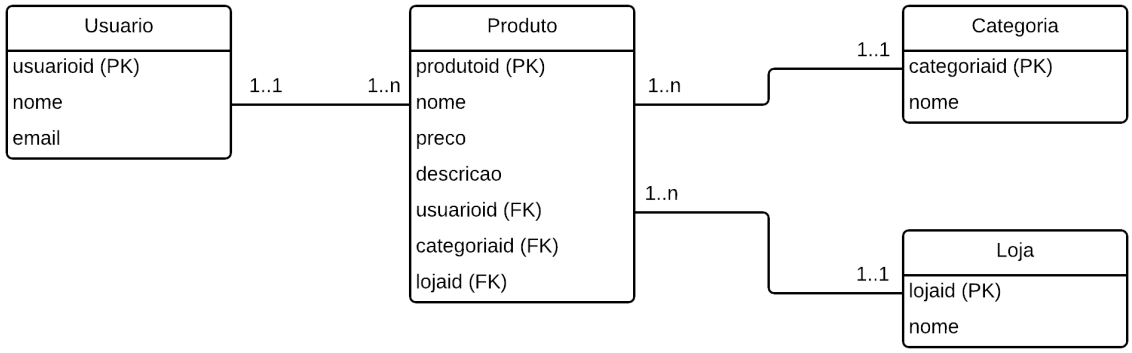

Figura 2 – Modelo lógico (relacional) para o sistema MKCMS

A Figura 2 permite uma visualização detalhada de como as entidades estão relacionadas quando é utilizado um SGBD relacional. Neste modelo lógico, as entidades e relacionamentos são representadas por meio de relações, que posteriormente serão implementadas por meio de tabelas. As instâncias das entidades são identificadas por meio de atributos identificadores (Chave Primária/*Primary Key* ou PK, na Figura). Os relacionamentos se estabelecem por meio de atributos de referência (Chave Estrangeira/*Foreign Key* ou FK, na Figura).

É importante destacar que, no modelo relacional, normalmente busca-se a normalização, visando alcançar as propriedades anteriormente descritas. Portanto, ainda que haja outras possibilidades de mapeamento, estas são restritas pelas regras de normalização, reduzindo a possibilidade de variação. Por exemplo, para relacionamentos do tipo "um-para-muitos", normalmente é acrescentada, na relação da entidade do lado "muitos", uma *Foreign Key* que referencia o atributo identificador da relação da entidade do lado "um". Também seria possível a criação de uma relação intermediária, dedicada exclusivamente ao relacionamento.

Já em bancos não-relacionais, a possibilidade de variação é maior, não apenas pela desnormalização que às vezes é implementada em prol do desempenho, mas pela maior flexibilidade em sua estrutura. Em um banco orientado a documentos, por exemplo, é possível utilizar atributos multivalorados. A seguir apresentam-se dois exemplos de como o modelo ER definido na Figura 1 poderia ser representado logicamente.

```
1    Coleção Usuario
2    {
3        "_id": int,
4        "nome": string,
5        "email": string
6    }
7
```

```
8     Coleção Produto
9     {
10        "_id": int,
11        "nome": string,
12        "preco": double,
13        "descricao": string,
14        "usuarioid": int,
15        "categoriaid": int,
16        "lojaid": int
17    }
18
19    Coleção Categoria
20    {
21        "_id": int,
22        "nome": string
23    }
24
25    Coleção Loja
26    {
27        "_id": int,
28        "nome": string
29    }
```

Listagem 1.1 – Modelo lógico (MongoDB) para um sistema gerenciador de conteúdo

A Listagem 1.1 representa o modelo ER da Figura 1 como coleções do banco de dados MongoDB (orientado a documentos). No MongoDB, a chave primária de todos os documentos é chamada de "_id", assim, as referências às coleções Usuario, Categoria e Loja presentes na coleção Produto (linhas 14, 15 e 16) representam o valor contido no atributo "_id" das respectivas coleções.

Supondo que as coleções *Categoria* e *Loja* são sempre acessadas junto com a coleção *Produto*, pode-se optar por incorporar os documentos destas coleções na coleção *Produto*, conforme mostra a Listagem 1.2.

```
1     Coleção Usuario
2     {
3         "_id": int,
4         "nome": string,
5         "email": string
6     }
7
8     Coleção Produto
9     {
```

```
10          "_id": int,
11          "nome": string,
12          "preco": double,
13          "descricao": string,
14          "usuarioid": int,
15          "categoria": {
16              "_id": int,
17              "nome": string
18          },
19          "loja": {
20              "_id": int,
21              "nome": string
22          }
23      }
24
25      Coleção Categoria
26      {
27          "_id": int,
28          "nome": string
29      }
30
31      Coleção Loja
32      {
33          "_id": int,
34          "nome": string
35      }
```

Listagem 1.2 – Modelo conceitual (MongoDB) com incorporação de documentos para um sistema gerenciador de conteúdo

Na Listagem 1.2 as coleções Categoria e Loja foram incorporadas à entidade Produto (linhas 15 a 18 e 19 a 22, respectivamente). Comparando com a Listagem 1.1 nota-se que há duplicação de dados, ou seja, para registros que pertençam a uma mesma categoria ou loja haverá repetição dos dados que descrevem essas coleções.

Enquanto a desnormalização pode beneficiar o desempenho, é necessário cuidado maior para manutenção da consistência. Além disso, e este é o principal foco deste trabalho, perde-se em uniformidade nas consultas. Para ilustrar o problema, supõe-se que exista um banco de dados com a estrutura representada na Figura 1 e que deseja-se realizar a seguinte consulta (expressa em linguagem natural):

```
1 SELECIONAR o Nome, Preço, Categoria e Loja dos Produtos que
```

```
PERTENÇAM a Categoria "Games"
```

Listagem 1.3 – Exemplo de consulta em linguagem natural

A consulta definida na Listagem 1.3 deverá realizar uma busca a partir da entidade Produto, realizando junções com as entidades Categoria e Loja. Será aplicada uma operação de seleção (“Produtos que PERTENÇAM a Categoria Games”) e então uma projeção, que exibirá apenas os atributos nome, preço, categoria e loja.

Supõe-se também que o resultado obtido ao executar a consulta definida na Listagem 1.3 seja o seguinte:

Nome	Preço	Categoria	Loja
Recore - Xbox One	19,99	Games	Submarino
Shadow of the Tomb Raider - Xbox One	204,99	Games	Submarino

Tabela 1 – Resultado obtido ao executar a consulta definida na listagem 1.3

Supondo agora que o banco de dados utilizado é o MySQL, e os dados representam fielmente o modelo da figura 2, o resultado exibido na Tabela 1 pode ser obtido utilizando a consulta definida na listagem 1.4.

```
1  SELECT
2      p.nome,
3      p.preco,
4      c.nome,
5      l.nome
6  FROM Produto p
7  JOIN Categoria c ON p.categoriaid = c.categoriaid
8  JOIN Loja l ON p.lojaid = l.lojaid
9  WHERE
10     c.nome = 'Games'
```

Listagem 1.4 – Consulta SQL para obter o resultado definido na Tabela 1

Caso o banco de dados utilizado seja o MongoDB, será criada uma consulta para cada modelo definido nas Listagens 1.1 e 1.2. Essas consultas estão definidas nas Listagens 1.5 e 1.6.

```
1 db.Produto.aggregate([
2   {$lookup: {
3     from: "Categoria",
4     localField: "categoriaid",
5     foreignField: "_id",
6     as: "categoria"
7   }},
```

```
8   {$lookup: {
9     from: "Loja",
10    localField: "lojaid",
11    foreignField: "_id",
12    as: "loja"
13  }},
14  {$unwind: "$categoria"},
15  {$unwind: "$loja"},
16  {$match: {
17    "categoria.nome": "Games"
18  }},
19  {$project: {
20    _id: 0,
21    nome: 1,
22    preco: 1,
23    categoria: "$categoria.nome",
24    loja: "$loja.nome"
25  }}
26 ]);
```

Listagem 1.5 – Consulta para MongoDB utilizando o modelo definido na Listagem 1.1

Na consulta definida pela Listagem 1.5 é utilizado o framework de agregação do MongoDB. Com a agregação, é possível utilizar o operador **$lookup** nas linhas 2 e 8 que funciona de maneira similar ao **JOIN** do SQL.

O **$lookup** resulta em uma array contendo os registros encontrados na coleção definida em 'from' com o valor da chave '_id'. Como esse valor é único para a coleção, utilizamos o operador **$unwind** (linhas 14 e 15) para gerar um documento para cada item do array **categoria** e **loja**. Em sequência utiliza-se o operador **$match** na linha 16, para filtrar os resultados, e o operador **$project** na linha 19, que definirá a estrutura do documento final.

Para o modelo definido na Listagem 1.2, poderá ser utilizada a consulta definida na Listagem 1.6 para obter os resultados descritos na Tabela 1.

```
1 db.Produto.aggregate([
2   {$match: {
3     "categoria.nome": "Games"
4   }},
5   {$project: {
6     _id: 0,
7     nome: 1,
8     preco: 1,
```

```
9       categoria: "$categoria.nome",
10      loja: "$loja.nome"
11    }}
12  ]);
```

Listagem 1.6 – Consulta para MongoDB utilizando o modelo definido na Listagem 1.2

Nota-se que a consulta definida na Listagem 1.6 é mais simples que a consulta definida na Listagem 1.5 por dispensar o uso de outros operadores além da seleção e projeção.

Nota-se também que apesar de bancos de dados NoSQL serem mais flexíveis quanto à estrutura, é necessário adequar as consultas (Listagens 1.5 e 1.6) ao modelo de dados (Listagens 1.1 e 1.2) definido a fim de se obter o mesmo resultado.

Com o objetivo de viabilizar a escrita de consultas de uma forma padronizada e independente da estrutura de armazenamento (esquema) dos dados, Noguera (2018) propôs a extensão de uma álgebra ER para consultas em bancos de dados NoSQL orientado a documentos (MongoDB), que:

- **(i) permita a especificação de consultas de forma independente do tipo de banco NoSQL usado:** O trabalho de Noguera (2018) é baseado na álgebra proposta por Parent e Spaccapietra (1984). Essa álgebra consiste de um conjunto de operadores de manipulação de dados que atuam sobre elementos do modelo Entidade-Relacionamento tradicional, sendo portanto independentes da forma com que os dados são armazenados, seja em bancos relacionais, seja NoSQL. Noguera (2018) criou uma sintaxe concreta para essa álgebra;
- **(ii) possibilite consultas não triviais:** A literatura apresenta diversos trabalhos que atendem ao item (i), mas quase sempre as abordagens são limitadas a consultas triviais, como a busca de uma entidade apenas, por exemplo (ATZENI; BUGIOTTI; ROSSI, 2012)(SELLAMI; BHIRI; DEFUDE, 2014)(LI; MA; CHEN, 2014). No trabalho de Noguera (2018), as consultas especificadas na álgebra ER são transformadas em consultas equivalentes no MongoDB, utilizando principalmente o operador de junção, possibilitando assim a criação de consultas não triviais;
- **(iii) as consultas sempre gerem o mesmo resultado:** As transformações analisam as diferentes possibilidades de mapeamento entre o modelo ER e as coleções de documentos no MongoDB, incluindo duplicações, documentos embutidos e referências entre documentos, visando obter sempre a mesma estrutura definida por Parent e Spaccapietra (1984).

Formalmente, diz-se que as transformações descritas nos itens (ii) e (iii) acima consistem em uma semântica operacional para a álgebra ER (PARENT; SPACCAPIETRA,

1984). Testes realizados por Noguera (2018) demonstram que a semântica implementada é consistente com a álgebra ER. No entanto, apenas o operador de **junção** (JOIN) entre duas entidades foi implementado. Apesar de ser o componente mais complexo, sem os demais operadores não é possível realizar testes com exemplos reais.

Além disso, o trabalho de Noguera (2018) não considera o desempenho das consultas, apenas a consistência com a álgebra. Apesar de esta consistência ser essencial, o verdadeiro motivador do trabalho, em última instância, é proporcionar um ganho de desempenho, caso contrário é mais simples e seguro manter-se com a tradicional abordagem relacional, já testada e validada ao longo dos anos. Apesar de, em tese, o uso de NoSQL por si só trazer potencial para melhorar o desempenho em larga escala, isso não foi testado devido às limitações da implementação.

1.2 Objetivo

Fatores como a falta de padrão de consultas nos diferentes bancos de dados NoSQL, os diversos mapeamentos possíveis de dados para representar a mesma informação, e a necessidade de se criar uma consulta diferente para cada tipo a fim de obter o mesmo resultado, servem como motivação para este trabalho.

O objetivo foi completar o trabalho de Noguera (2018), implementando os outros operadores propostos no trabalho de Parent e Spaccapietra (1984). Além da operação de junção entre duas entidades, que já foi implementada por Noguera (2018), foram implementados os operadores de junção composta, entre mais de duas entidades, projeção, seleção e produto cartesiano. Com esses operadores, foi possível ampliar os testes incluindo outras consultas e testes em sistemas reais.

Com a álgebra completa funcionando em testes reais, o próximo objetivo foi analisar o desempenho das consultas geradas. A análise visou comparar o tempo de execução das consultas geradas com o tempo de execução de consultas escritas à mão por um engenheiro de software. O objetivo dessa comparação é entender o possível impacto da álgebra no desempenho das consultas.

1.3 Metodologia de pesquisa

Para esse projeto, adotou-se uma metodologia teórica e prática.

Primeiro, foi realizado um estudo aprofundado sobre o banco de dados não relacional MongoDB. O MongoDB é um banco de dados NoSQL orientado a documentos. Assim como no trabalho de Noguera (2018), optou-se por utilizar o MongoDB pelo fato de ser um dos bancos não relacionais mais utilizados do mercado, além de atender às necessidades técnicas de implementação da pesquisa.

Foram estudados também outros trabalhos na literatura que abordam o mesmo problema.

Em seguida foi feita uma análise no estado anterior da implementação da álgebra ER proposta por Parent e Spaccapietra (1984) realizada por Noguera (2018), a fim de compreender seu funcionamento e principais decisões de projeto e implementação.

Em seguida, foram feitas as alterações necessárias na implementação existente para que os geradores de código passassem a contemplar os operadores adicionais: junção composta, seleção, projeção e produto cartesiano.

Um outro limitante do trabalho de Noguera (2018), este de ordem mais prática, é a verificação da consistência da implementação com a álgebra de Parent e Spaccapietra (1984), que tinha sido feita manualmente. Apesar de ser suficiente para os propósitos de sua dissertação, em exemplos maiores e mais complexos, como aqueles avaliados nesta pesquisa, essa abordagem se mostrou inviável. Assim, neste trabalho adotou-se a verificação e teste automatizado dos resultados das consultas. Desta forma foi possível ampliar o conjunto de testes, aumentando assim a confiança nos resultados. Os testes foram realizados com o apoio das ferramentas Newtonsoft.Json[3] que é uma biblioteca para manipulação de dados no formato JSON, e FluentAssertions[4], que é uma biblioteca dedicada à construção de testes de forma objetiva. As duas ferramentas permitem a comparação de documentos no formato JSON. O objetivo foi verificar se as consultas geradas para diferentes mapeamentos produzem os mesmos resultados, e se esses resultados estão consistentes com a álgebra ER de Parent e Spaccapietra (1984).

Por fim, foram realizadas análises de desempenho, com o auxílio da ferramenta "explain" do MongoDB. Essa ferramenta gera, para cada consulta, dados detalhados sobre como cada consulta foi realizada, assim como o tempo de execução. O tempo foi a principal variável, mas outros detalhes também foram analisados, como o tipo de operação escolhida pelo MongoDB, o uso de índices, entre outros detalhes que podem ser utilizados para explicar as diferenças. Foram comparados os resultados de consultas construídas manualmente com os resultados de consultas geradas automaticamente para os diferentes mapeamentos.

Ambas as análises (de consistência e desempenho) foram feitas em dois sistemas reais: o sistema PrograduWeb, que é um sistema real que foi utilizado durante muitos anos para controle acadêmico da UFSCar, e um sistema chamado MKCMS (Sistema gerenciador de conteúdo para marketing digital). Destes sistemas, foram utilizados os modelos conceitual e lógico, e exemplos de consultas reais neles implementadas.

Por fim, foram discutidos os resultados, onde foi possível identificar possíveis melhorias e trabalhos futuros.

[3] https://www.newtonsoft.com/json
[4] https://fluentassertions.com/

1.4 Resultados e Contribuições Obtidas

Os resultados evidenciam que a consistência foi constatada em 100% dos testes realizados de forma automática. E a análise de desempenho identificou diferenças significativas entre os resultados para os diferentes mapeamentos.

Assim, em complemento ao trabalho de Noguera (2018), considera-se que o principal resultado desta pesquisa foi atingido:

Dados

- um único modelo ER,
- uma única consulta Q, e
- n mapeamentos ER-MongoDB M1, M2, ..., Mn,

é possível gerar, automaticamente,

- n consultas nativas para o MongoDB Q1, Q2, ..., Qn,
- que geram n resultados R1, R2, ..., Rn,
- onde R1, R2, ..., Rn são todos consistentes entre si e com a álgebra ER,
- possuem desempenhos diferentes D1, D2, ..., Dn, e
- o desempenho observado é inferior, porém em quase todos os casos muito próximo ao desempenho de consultas escritas à mão.

Para traduzir este resultado em termos práticos, podemos retomar a motivação original: com consultas nativas escritas à mão, caso seja constatado, em produção, que o desempenho de um determinado modelo lógico no MongoDB não seja satisfatório, é possível melhorá-lo modificando-o, embutindo coleções e duplicando dados, por exemplo. Mas isso implica em reescrever todas as consultas relacionadas, e possivelmente o código que utiliza os resultados dessas consultas. Isso se dá pois não existe uma padronização e uniformização das consultas em um modelo não-relacional.

A abordagem aqui realizada resolve este problema. Para ilustrar essa resolução, considere-se o seguinte cenário apresentado na Figura 3: inicialmente tem-se um mapeamento não embutido (normalizado) onde cada entidade é mapeada para uma própria coleção. O Engenheiro de Software cria, primeiro uma representação do modelo ER (Figura 3A). Em seguida, ele cria a representação desse mapeamento (Figura 3B) para o algoritmo e escreve as consultas sobre o modelo ER (Figura 3C), sem considerar os detalhes de implementação. As consultas nativas são automaticamente geradas (Figura 3D).

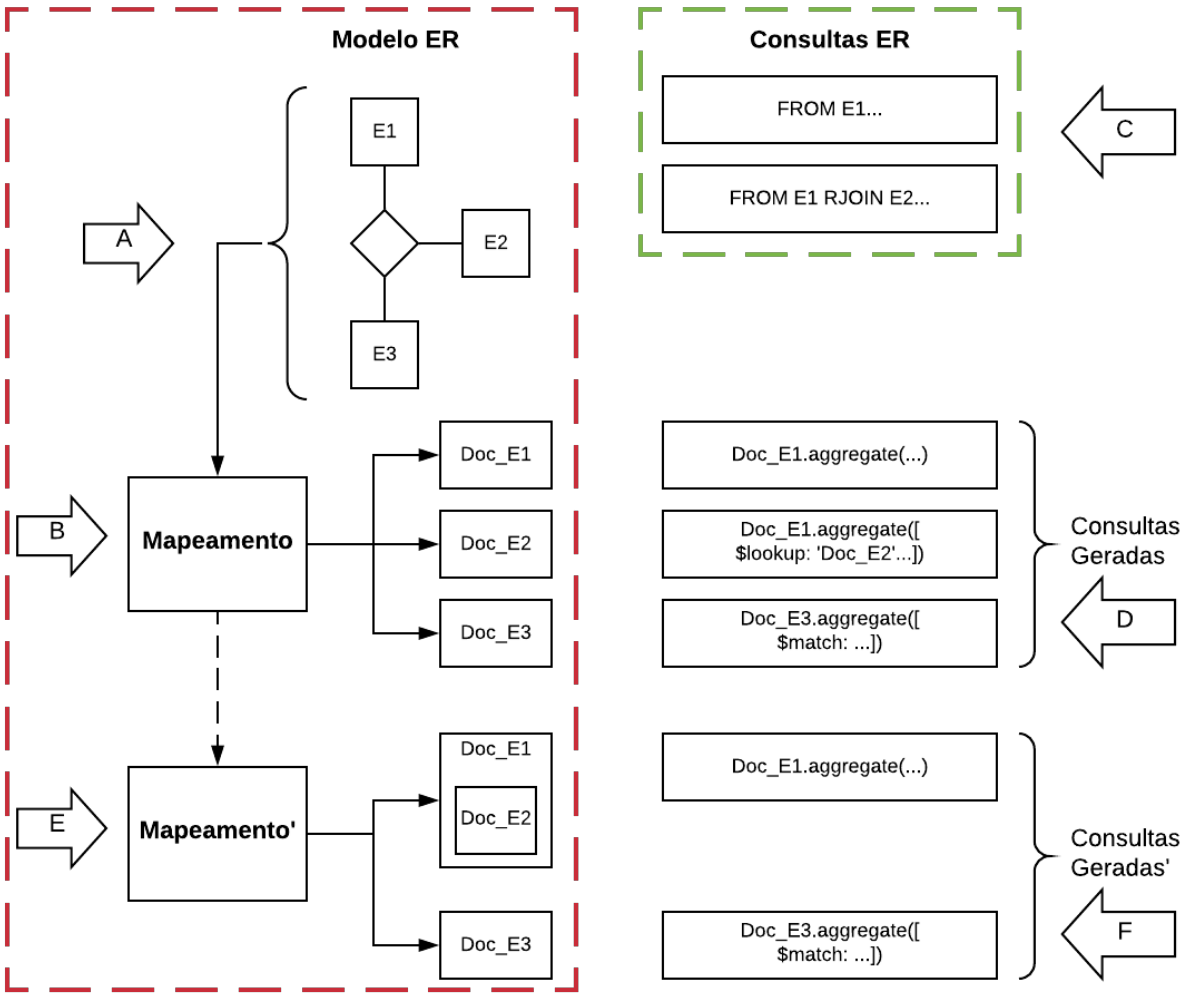

Figura 3 – Visão geral do cenário de uso

Em um segundo momento, o Engenheiro de Software decide embutir um documento em outro, para melhorar o desempenho. Para isso, ele deve modificar a representação do mapeamento original, produzindo assim uma nova versão do mapeamento (Figura 3E). Assim, para que esse novo mapeamento seja implementado, basta gerar as consultas novamente (Figura 3F). Nesse processo, nota-se que o Engenheiro de Software trabalha em um nível conceitual (áreas tracejadas na Figura 3). Com isso, tem-se uma maior rapidez no processo. Por exemplo, suponha que foram escritas 39 consultas. Ao invés de analisar as 39 consultas e modificar aquelas que são afetadas, basta modificar o mapeamento e gerar as consultas novamente. Tem-se também maior abstração, deixando o trabalho mais focado nos conceitos do domínio e menos preocupado com detalhes de implementação.

A dependência de um único modelo ER e uma única consulta Q significa que, para o Engenheiro de Software, é possível trabalhar em nível de modelagem ER, efetivamente abstraindo os detalhes de mapeamento para fins de escrita das consultas. Caso sejam desejáveis modificações no modelo lógico para melhorar o desempenho, basta modificar o mapeamento e gerar novamente as consultas. Como o resultado é sempre consistente com

a álgebra, independente do mapeamento, a quantidade de esforço necessário para esta tarefa é bastante reduzida.

Um benefício adicional é que, por ser um processo automático, torna-se possível experimentar rapidamente com diferentes mapeamentos para uma mesma consulta, possibilitando análises rápidas do impacto de cada decisão de projeto.

Outro benefício é a abstração. Por serem baseadas no modelo ER, e não no modelo lógico, as consultas podem ser especificadas sem serem considerados detalhes do mapeamento, o que torna o processo mais flexível permitindo que todo planejamento seja feita em nível mais alto.

O desempenho também é um resultado positivo. Apesar de ser inferior ao das consultas escritas à mão, em quase todos os testes essa diferença é pouco significativa, e naqueles casos em que a diferença se mostrou grande foram identificadas possibilidades de melhorias. Apesar disso, a consistência com a álgebra pode ser um fator decisivo, pois ela configura uma vantagem em relação às consultas escritas à mão. Essa consistência dá ao engenheiro de software a garantia de que os resultados sempre terão a mesma estrutura, facilitando a interpretação e uso dos dados.

1.5 Limitações

A proposta inicial pretendia finalizar a implementação dos operadores da álgebra proposta por Parent e Spaccapietra (1984) e incluir análise de desempenho das consultas geradas. O trabalho de Parent e Spaccapietra (1984) também define que o resultado dos operadores **Relationship Join** (do Inglês, junção de relacionamento) e **Cartesian Product** (do Inglês, produto cartesiano) herde todos os relacionamentos das entidades que compõe as operações.

O problema em questão não possui solução trivial como constado por outros autores como Sellami, Bhiri e Defude (2014) e Noguera (2018), considerando também que a álgebra proposta por Parent e Spaccapietra (1984) define uma estrutura para o resultado das operações, tem-se além do mapeamento inicial entre entidades e relacionamentos do modelo ER com o esquema NoSQL, o mapeamento entre esses mesmos elementos e documento resultando das operações. Optou-se então por não incluir a herança de relacionamento, destaca-se que isso não afeta a execução desses operadores nem o uso de consultas aninhadas (por exemplo, a junção de uma entidade que é resultado da junção de outras entidades).

Também foram identificadas algumas limitações na implementação realizada. A primeira delas foi que a ordem das operações nas consultas geradas não foi otimizada, o que gerou resultados com desempenhos inferiores ao que seria possível. Em particular,

observou-se nos testes que em alguns exemplos a operação de seleção foi colocada no final, após a junção, o que levou a um desempenho ruim. Enquanto aparentemente o problema poderia ser resolvido rapidamente, optou-se por deixar a investigação para trabalhos futuros, uma vez que a consistência poderia ser prejudicada caso o algoritmo de geração de código fosse alterado sem uma análise mais aprofundada.

Outros limitantes em termos de desempenho também foram identificados. De acordo com Gorla, Ng e Law (2012) a transferência de dados entre CPU e o dispositivo de armazenamento é claramente o maior gargalo no desempenho de um banco dados, e quanto menos acessos ao disco, maior é a eficiência do banco de dados ao realizar transações. Fragmentação ou particionamento é uma técnica que melhora o desempenho de um banco dados, baseado no principio que fragmentos de dados que são frequentemente acessados juntos requerem menos acessos ao disco se armazenados juntos (GORLA; NG; LAW, 2012). De acordo com a documentação do MongoDB (MONGODB, 2019) a combinação de dados relacionados é uma forma de melhorar o desempenho assim como o uso de índices. A documentação cita ainda outras soluções para a melhoria de desempenho, como subdividir coleções muito grandes e embutir documentos de coleções pequenas (poucos dados) em outras. Muitas dessas sugestões acabaram não sendo incorporadas nos geradores de código implementados.

Por fim, o operador de seleção implementado tem limitações quanto ao uso de expressões lógicas e aritméticas complexas e ao uso de funções de agregação. Devido às limitações de tempo, optou-se por implementar apenas comparações simples, que constituem o estritamente necessário para execução dos testes.

1.6 Organização desta Dissertação

Esta dissertação está organizada da seguinte maneira: este capítulo apresentou o contexto sobre o qual o trabalho foi desenvolvido, suas motivações, metodologia, resumo das contribuições obtidas e limitações do trabalho.

O Capítulo 2 contém a revisão bibliográfica situando o contexto teórico do trabalho.

No Capítulo 3 traz uma revisão dos trabalhos relacionados com o tema desta pesquisa a fim de discutir o estado da arte dos tópicos abordados.

No Capítulo 4 está descrito com detalhes o desenvolvimento deste trabalho, a extensão na implementação dos operadores da álgebra e descrição da validação das operações implementadas.

No Capítulo 5 apresenta-se as avaliações de conformidade e consistência realizadas para validar os resultados das consultas geradas pelo algoritmo para os mapeamentos

apresentados e também a análise de desempenho dessas consultas quando comparadas a um consulta escrita à mão.

E por fim, o Capítulo 6 descreve as contribuições alcançadas e as conclusões deste trabalho.

2 Revisão Bibliográfica

Neste capitulo são descritos os conceitos relacionados a este trabalho. São introduzidos conceitos sobre banco de dados relacional, modelo de dados e bancos de dados NoSQL.

2.1 Banco de dados

Bancos de dados e sistemas de banco de dados são componentes essenciais para a vida em uma sociedade moderna: a maioria de nós realiza atividades diariamente que envolvem a interação com um banco de dados (ELMASRI et al., 2010).

Banco de dados é uma coleção de dados relacionados. Considera-se **dados** como fatos conhecidos que podem ser registrados e possuem um significado implícito (ELMASRI et al., 2010).

Um **SGBD (sistema gerenciador de banco de dados)** é um conjunto de programas que permite usuários criar e manter um banco de dados. O SGBD é um software genérico que facilita o processo de definição, construção, manipulação e compartilhamento de banco de dados entre vários usuários e aplicações (ELMASRI et al., 2010).

2.2 Modelos de dados

De acordo com Elmasri et al. (2010) foram propostos diversos modelos de dados, que podemos categorizar de acordo com os conceitos que são utilizados para descrever a estrutura do banco de dados. Modelos conceituais ou de alto-nível provêm de conceitos que estão próximos da maneira que muitos usuários entendem os dados, enquanto modelos de baixo-nível ou modelos físicos descrevem os detalhes de como esses dados são armazenados no computador. Entre esses dois extremos existe uma classe de modelos representacionais (ou de implementação). Estes modelos provem conceitos que podem ser facilmente entendidos pelo usuário final mas que não estão distantes da forma com que dados são organizados durante o armazenamento (ELMASRI et al., 2010).

Um dos modelos conceituais mais utilizados é o modelo Entidade-Relacionamento. Esse tipo de modelo utiliza conceitos de entidades, atributos e relacionamentos.

Modelos físicos descrevem como os dados são armazenados em arquivos no computador, representando informações como formato de registro, ordenação e caminhos de acessos. Caminho de acesso é uma estrutura que torna a busca por registros de um determinado banco de dados eficiente. Um **índice** é um exemplo de caminho de acesso que

permite acesso direto aos dados usando um termo de índice ou palavra-chave (ELMASRI et al., 2010).

2.2.1 Modelo entidade-relacionamento

De acordo com Chen (1976) uma entidade é uma "coisa" que pode ser identificada de forma distinta. Uma determinada pessoa, empresa ou evento são exemplos de uma entidade. Um atributo representa uma propriedade que descreve parte de uma entidade, como por exemplo, o nome de um empregado ou o salário (ELMASRI et al., 2010). Um relacionamento pode ser definido como a associação entre entidades. Por exemplo, "pai-filho" é um relacionamento entre duas ocorrências da entidade "pessoa" (CHEN, 1976).

Entidades podem ser classificadas em diferentes grupos, como por exemplo, *Empregado*, *Projeto* e *Departamento*. Para cada grupo, existe um predicado que determina se uma entidade pertence ao conjunto (CHEN, 1976).

Chen (1976) define que um conjunto de relacionamentos é a relação matemática entre n entidades, sendo cada uma retirada de um grupo de entidades, conforme definido pela Expressão 2.1.

$$[e_1, e_2, ..., e_n] | e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n \tag{2.1}$$

As informações sobre uma entidade ou relacionamento são obtidas através de observação ou mensuração, e são expressadas por um conjunto de pares atributo-valor. *"3", "vermelho", "João"* são valores. Valores são classificados em diferentes grupos, como *"Cor", "Nome", "Sobrenome"*. Existe um predicado associado a cada grupo de valores para determinar se um valor pertence ao conjunto (CHEN, 1976).

Um atributo pode ser formalmente definido como uma função que mapeia de um conjunto de entidades ou relacionamentos para um grupo de valores ou o produto cartesiano de grupos de valores, conforme definido na Expressão 2.2 (CHEN, 1976).

$$f : E_i \; ou \; R_i \rightarrow V_i \; ou \; V_{i1} \times V_{i2} \times \cdots \times V_{in} \tag{2.2}$$

Relacionamentos também possuem atributos. O conceito de atributo de um relacionamento é importante no entendimento da semântica dos dados ao determinar as dependências funcionais dos dados (CHEN, 1976).

Segundo Elmasri et al. (2010), a cardinalidade para relacionamentos binários especifica o número máximo de instâncias do relacionamento em que uma entidade pode fazer parte. Por exemplo, na Figura 4 o relacionamento binário entre as entidades "DEPARTMENT" e "EMPLOYEE" definido por "WORKS_FOR" possui cardinalidade 1:N, significando que cada departamento pode ser relacionado com (isso é, emprega) qualquer

número de empregados, mas um empregado pode se relacionar apenas com (trabalha para) um único departamento. Isso significa que para esse relacionamento em particular (WORKS_FOR), uma particular entidade departamento pode ser relacionada com qualquer número de empregados (N indica que não existe número máximo). No entanto, um empregado pode se relacionar no máximo com um departamento. As possíveis cardinalidades para relacionamentos binários são: 1:1, 1:N, N:1 e M:N.

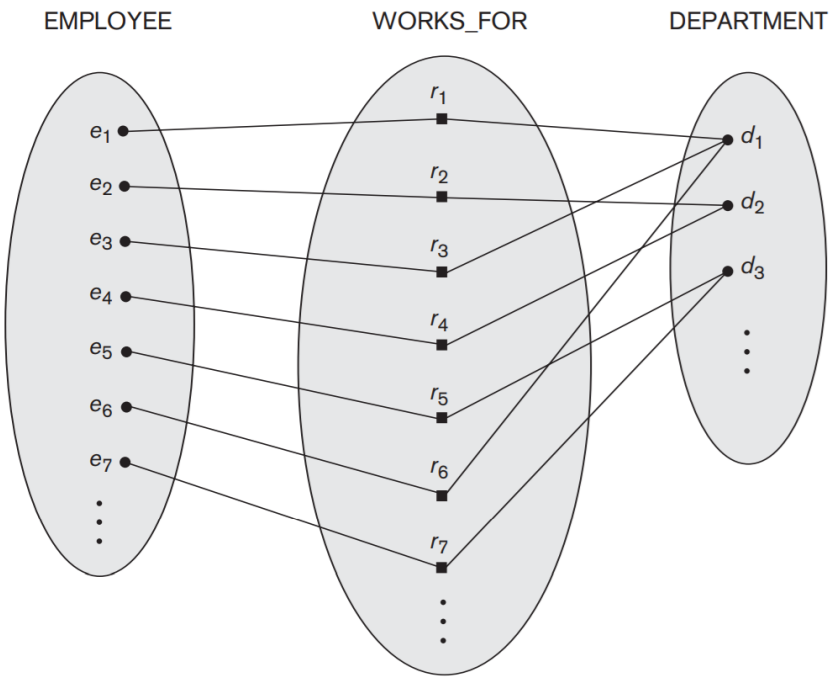

Figura 4 – Algumas instâncias do relacionamento WORKS_FOR (ELMASRI et al., 2010)

Um exemplo de um relacionamento binário de cardinalidade 1:1 é a relação MANAGES (Figura 5), que relaciona a entidade departamento com o empregado que gerencia o departamento. Isso representa uma restrição local que, a qualquer momento, um empregado pode gerenciar apenas um departamento e um departamento pode ter apenas um gerente (ELMASRI et al., 2010).

A cardinalidade para relacionamentos binários é representada em diagramas ER através da notação 1, M e N nas formas (losango) representado o relacionamento (ver Figura 1) (ELMASRI et al., 2010).

2.3 Banco de dados relacional

Um banco de dados relacional é composto de tabelas ou relações. A terminologia tabela é mais comum nos produtos comerciais e na prática. Já a terminologia relação foi utilizada na literatura original sobre a abordagem relacional (HEUSER, 2009).

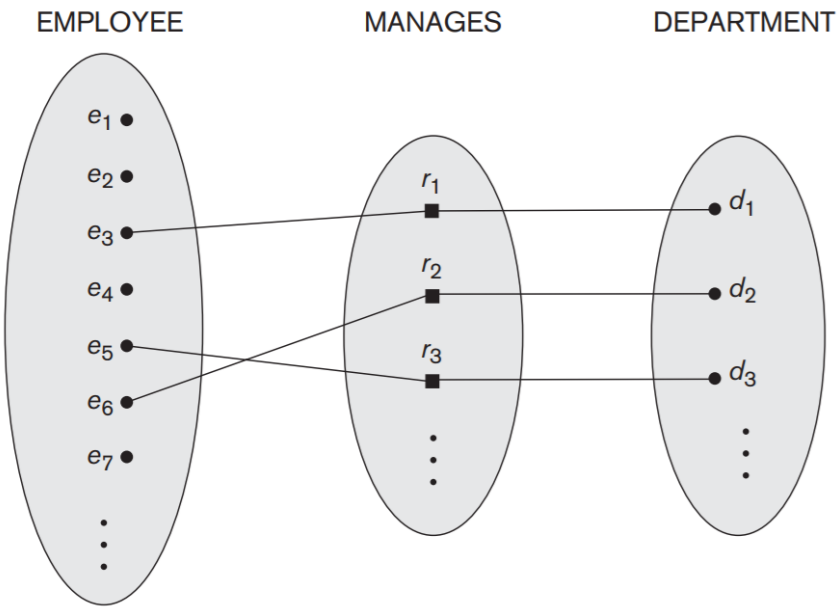

Figura 5 – Relacionamento 1:1 MANAGES (ELMASRI et al., 2010)

2.3.1 Tabelas

Uma tabela é um conjunto não ordenado de linhas. Cada linha é composta por uma série de campos (HEUSER, 2009). A Figura 6 demonstra uma tabela em um banco de dados, a tabela é chamada "Emp" e possui os atributos (colunas) *CódigoEmp, Nome, CodigoDepto, CategFuncional*.

De acordo com Heuser (2009) ao criar um modelo relacional a partir de um modelo entidade-relacionamento (conceitual) as entidades são representadas por tabelas e os atributos são representados pelas colunas das tabelas.

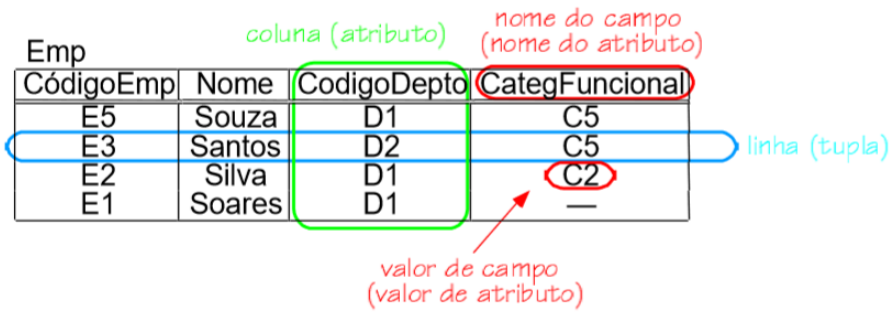

Figura 6 – Tabela (HEUSER, 2009)

Segundo Heuser (2009) pode-se comparar uma tabela de um banco de dados relacional a um arquivo convencional do sistema de arquivos de um computador, obtendo as seguintes diferenças:

- As linhas de uma tabela não estão ordenadas. A ordem de recuperação pelo SGBD

é arbitrária.

- Os valores do campo de uma tabela são atômicos e monovalorados.
- As linguagens de consulta a bases de dados relacionais permitem o acesso por quaisquer critérios envolvendo os campos de uma ou mais linhas.

2.3.2 Chaves

Chaves são campos especiais que possuem papeis específicos em uma tabela. O tipo de chave determina o seu propósito na tabela (VIESCAS; HERNANDEZ, 2007).

De acordo com Viescas e Hernandez (2007) uma chave primária é um campo ou grupo de campos que identificam de maneira única cada registro em uma tabela. E quando composta por dois ou mais campos, chama-se *chave primária composta*. A chave primária é a mais importante por duas razões: (a) seu valor identifica um registro específico em todo o banco de dados, (b) a coluna (campo) identifica a tabela em todo o banco de dados. As chaves primárias também impõem integridade dos dados em nível de tabela e ajudam a estabelecer relacionamentos com outras tabelas.

Uma chave estrangeira é uma coluna ou uma combinação de colunas, cujos valores aparecem necessariamente na chave primária de uma tabela. A chave estrangeira é o mecanismo que permite a implementação de relacionamentos em um banco de dados relacional (HEUSER, 2009).

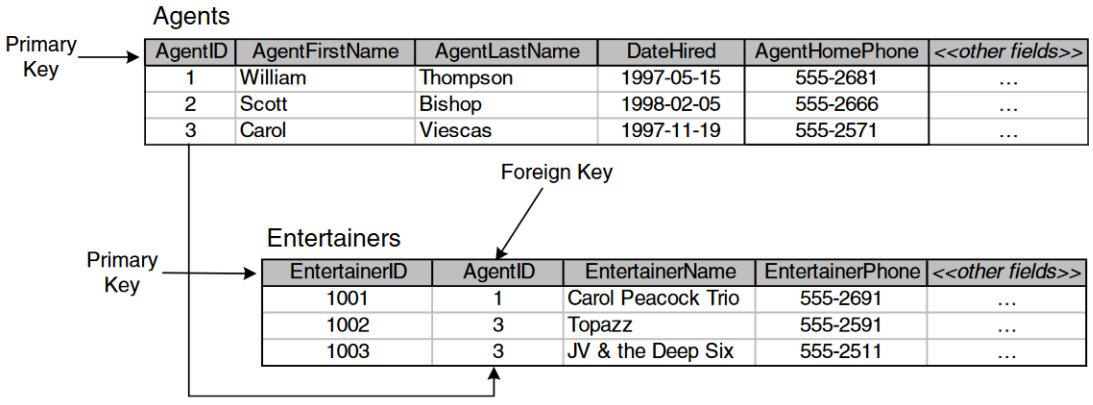

Figura 7 – Chaves primária e estrangeira (VIESCAS; HERNANDEZ, 2007)

2.3.3 Relacionamentos

Viescas e Hernandez (2007) dizem que se um registro de uma tabela pode ser associado de alguma maneira a registros de outra tabela, pode se dizer então que existe um relacionamento entre essas tabelas. A maneira em que o relacionamento é estabelecido

depende do tipo de relacionamento. Podem existir três tipos de relacionamento entre duas tabelas, um-para-um, um-para-muitos e muitos-para-muitos.

2.3.3.1 Um-para-um

Um par de tabelas possui um relacionamento um-para-um (1:1) quando cada registro de uma tabela pode ser relacionado com apenas um registro da outra tabela. Na Figura 8 existe um relacionamento 1:1 entre as tabelas "Agents" e "Compensation", a coluna "AgentID" é a chave-primária nas duas tabelas, e na tabela "Compensation" ela também serve como chave-estrangeira Nesse tipo de relacionamento uma tabela é referida como *tabela primária* e a outra como *tabela secundária*. O relacionamento é estabelecido ao pegar a chave-primária da tabela primária e a inserir na tabela secundária, onde ela se torna uma chave-estrangeira (VIESCAS; HERNANDEZ, 2007).

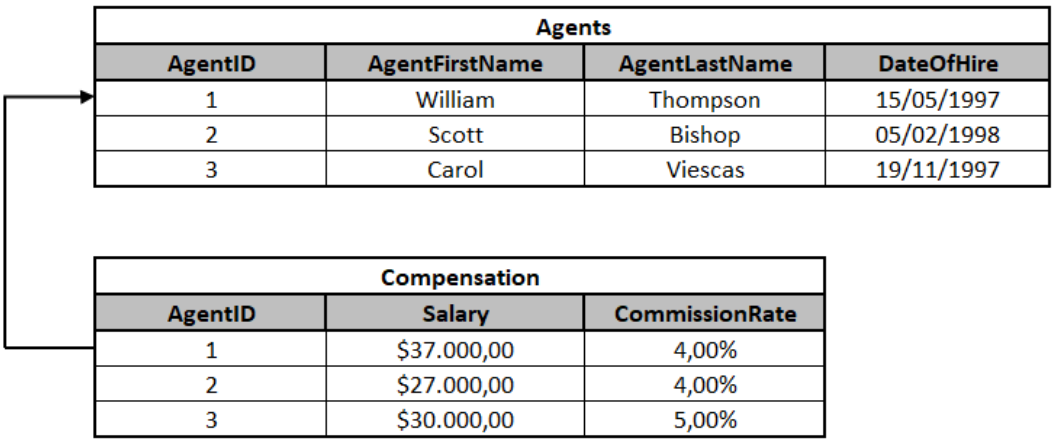

Figura 8 – Relacionamento 1:1 (baseado no trabalho de (VIESCAS; HERNANDEZ, 2007))

2.3.3.2 Um-para-Muitos

O relacionamento um-para-muitos acontece quando o registro de uma tabela pode ser relacionado a muitos registros de uma outra tabela, mas cada registro da segunda tabela pode referenciar apenas um registro da primeira. Esse relacionamento é estabelecido ao pegar a chave-primária da tabela do lado "um" e inserindo-a na tabela do lado "muitos", onde ela se torna uma chave-estrangeira (VIESCAS; HERNANDEZ, 2007).

A Figura 9 representa um relacionamento um-para-muitos entre as tabelas *Entertainers* e *Engagements*. Na tabela "Engagements" a coluna *EntertainerID* referencia a coluna *EntertainerID* da tabela *Entertainers*, agindo como chave-estrangeira.

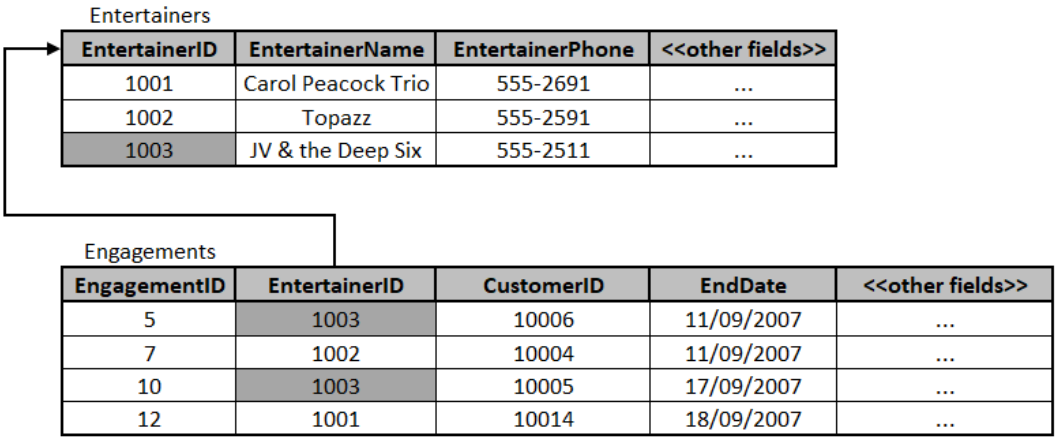

Figura 9 – Relacionamento 1:N (VIESCAS; HERNANDEZ, 2007)

2.3.3.3 Muitos-para-Muitos

Um par de tabelas possui um relacionamento muitos-para-muitos quando um registro da primeira tabela pode ser relacionado com múltiplos registros da segunda tabela, e quando um registro da segunda tabela pode ser relacionado com múltiplos registros da primeira tabela (VIESCAS; HERNANDEZ, 2007).

Para estabelecer este relacionamento é necessário criar uma tabela de ligação. Essa tabela proporciona uma forma fácil de associar dados de duas tabelas e ajuda a garantir que não haverá problemas ao adicionar, remover ou alterar qualquer dado do relacionamento (VIESCAS; HERNANDEZ, 2007).

A tabela de ligação é definida ao pegar uma cópia da chave-primária de cada tabela no relacionamento e usá-las para formar a estrutura da nova tabela. Esses atributos servem para dois propósitos distintos: Juntos formam uma chave-primária composta, e separados servem como chave-estrangeira (VIESCAS; HERNANDEZ, 2007).

A Figura 10 mostra como é definida uma relação muitos-para-muitos entre duas tabelas. Observa-se que na tabela *Engagements* as colunas *EntertainerID* e *Customer* são chaves-estrangeiras, referenciando a chave primária das tabelas *Entertainers* e *Customers*. As colunas com fundo cinza mostram a ocorrência de múltiplos relacionamentos entre as tabelas *Entertainers* e *Customers* através da tabela intermediária *Engagements*.

2.4 Banco de dados não relacional

O termo NoSQL foi usado pela primeira vez em 1998 para um banco de dados relacional que omitia o uso de SQL (STROZZI, 2010). O termo entretanto, passou a identificar inúmeros bancos de dados que possuem as seguintes características (BUTGEREIT, 2016):

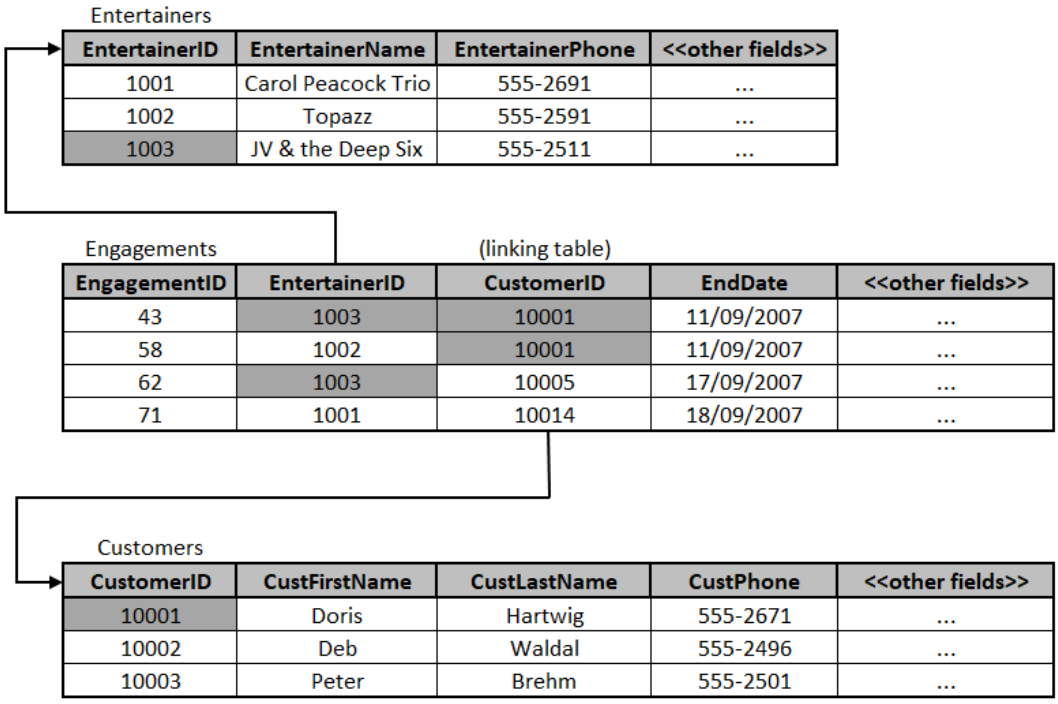

Figura 10 – Relacionamento N:M (baseado no trabalho de (VIESCAS; HERNANDEZ, 2007))

- Não utilizam SQL, mas suas próprias linguagens de consulta que podem ser similares ao SQL;
- São principalmente *open-source*, podendo existir versões comerciais; e
- O design é orientado a particionamento.

Bancos de dados relacionais utilizam transações ACID para lidar com a consistência dos dados. Isso conflita com um ambiente particionado, portanto bancos de dados NoSQL oferecem uma gama de opções para a consistência e distribuição (SADALAGE; FOWLER, 2013).

O termo ACID significa: *Atomicidade*, *Consistência*, *Isolamento* e *Durabilidade*, ou seja, todas as operações em uma transação serão completadas ou nenhuma (A), o banco de dados estará em um estado consistente quando a transação começar e terminar (C), a transação irá se comportar como se fosse a única operação em execução no banco de dados (I) e ao completar a transação, a operação não será revertida (D) (PRITCHETT, 2008).

Por terem como uma das principais aplicações a execução em ambiente particionado, os bancos de dados NoSQL normalmente sacrificam as propriedades ACID. Essa restrição foi demonstrada no teorema CAP. O teorema CAP foi proposto pelo professor

Eric Brewer. O teorema diz que um sistema de banco de dados distribuído não pode garantir todas as três propriedades, Consistência (C), Disponibilidade (A, do Inglês *Availability*) e Tolerância a particionamento (P, *Partition tolerance*), ao mesmo tempo (PRITCHETT, 2008).

Na Figura 11 observa-se a relação entre as propriedades do teorema CAP. Como o teorema diz que não é possível garantir as três propriedades simultaneamente, é possível garantir qualquer combinação entre duas das três propriedades. A imagem ilustra essa interação com a intersecção dos círculos representado as propriedades.

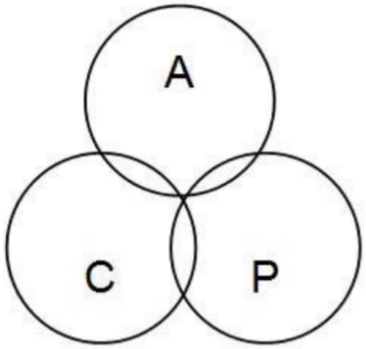

Figura 11 – O teorema CAP (WANG; TANG, 2012)

Como resultado do teorema CAP, Pritchett (2008) diz que se as propriedades ACID são a escolha para bancos de dados particionados, então, a alternativa para conseguir a disponibilidade no lugar da consistência é através do teorema BASE (WANG; TANG, 2012).

O teorema BASE é um produto do teorema CAP. BASE é a abreviação de *Basicamente Disponível (BA)*, *Estado-leve (S)* e *Eventualmente consistente (E)*. Isto significa que a aplicação funciona basicamente o tempo todo e poderá em algum momento estar dessincronizada [inconsistente] (estado-leve) (WANG; TANG, 2012).

O impacto do teorema CAP no projeto de sistemas de bancos de dados distribuídos (SBDD) é menor do que é frequentemente percebido. Uma outra troca entre consistência e latência tem maior e mais direta influência em diversos SBDDs conhecidos. A proposta PACELC unifica essa troca com o teorema CAP (Abadi, 2012).

O PACELC (do Inglês Partição Disponibilidade e Consistência Ou Latência e Consistência) expande o conceito do teorema CAP e diz que se houver um particionamento de rede (P), como o sistema fará a troca entre disponibilidade (A) e consistência (C), ou, na ausência de particionamento (E), como se dará a troca entre latência (L) e consistência (C) (Abadi, 2012).

Sadalage e Fowler (2013) dividem os bancos de dados NoSQL em 4 tipos, de acordo com seu modelo de dados.

2.4.1 Chave-valor

Um banco de dados do tipo chave-valor permite que o usuário armazene dados sem um esquema específico. Os dados são normalmente de um tipo definido em uma linguagem de programação ou um objeto. As informações consistem de duas partes, uma string representando a chave e o dado armazenado referido como valor, assim formando o par chave-valor. Riak[1] e DynamoDB[2] são exemplos de bancos de dados do tipo chave-valor (NAYAK; PORIYA; POOJARY, 2013).

2.4.2 Orientado a colunas

Bancos de dados orientados a colunas (ou família de colunas) armazenam dados como linhas que possuem várias colunas associadas a uma chave de linha (*row key*). Famílias de colunas são grupos de dados relacionados que são frequentemente acessados juntos. Um exemplo desse tipo de banco de dados é o Cassandra[3] (SADALAGE; FOWLER, 2013).

De acordo com Sadalage e Fowler (2013) a diferença entre bancos de dados orientados a colunas e bancos de dados relacionais é a forma em que os dados são armazenados. No banco de dados orientado a colunas, são armazenados grupos de colunas para todas as linhas.

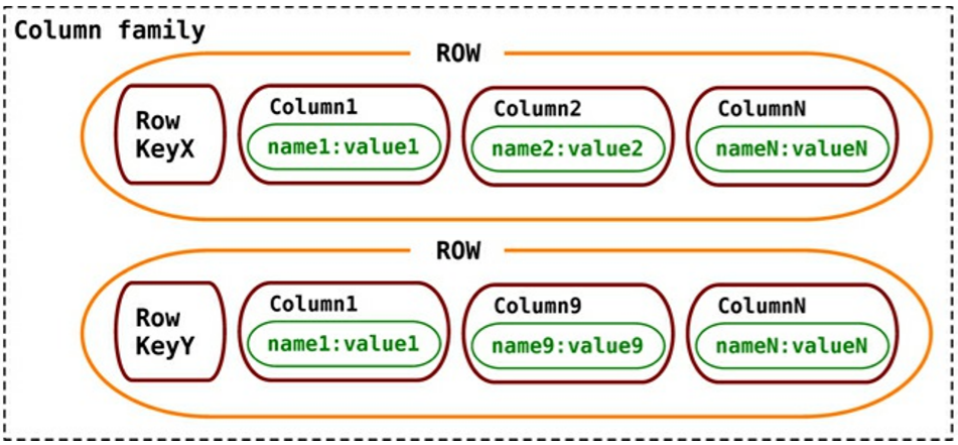

Figura 12 – Famílias de colunas (SADALAGE; FOWLER, 2013)

Na Figura 12 observa-se que cada linha (Row) é composta do atributo *Row Key* e de grupos de colunas, chamados famílias de colunas. Cada família de colunas pode ser composta de uma ou mais colunas, e cada linha pode ser composta de uma ou mais famílias de colunas que não precisam ser a(s) mesma(s) família(s) de colunas das outras linhas.

[1] http://basho.com/products/riak-kv/
[2] https://aws.amazon.com/pt/dynamodb/
[3] http://cassandra.apache.org/

2.4.3 Documentos

Armazenar dados em formato de documentos, sem a necessidade de garantir rígidas restrições de integridade, oferece grande desempenho e escalabilidade horizontal.

Os documentos são similares aos registros em um banco de dados relacional, mas são mais flexíveis por não possuírem esquema. Os documentos são armazenados em um formato padronizado, como XML, PDF, JSON, etc. Esses documentos são associados a uma chave única, que pode ser uma string ou uma string que se refere a uma URI. O MongoDB[4] e o CouchDB[5] são exemplos de bancos de dados NoSQL orientados a documentos (NAYAK; PORIYA; POOJARY, 2013).

2.4.4 Grafos

Um banco de dados do tipo grafo permite armazenar entidades e relacionamentos entre essas entidades. As entidades são conhecidas como "nós", que possuem propriedades. Os relacionamentos são arestas que possuem propriedades. A direção em que as arestas apontam tem significado, dando sentido ao relacionamento entre as entidades conectadas por ela. (Exemplo: Neo4j) (SADALAGE; FOWLER, 2013).

2.5 Gerenciamento de Modelos

Gerenciamento de modelos é uma abordagem genérica para resolver problemas de programabilidade de dados onde é necessário o desenvolvimento de mapeamentos precisos (BERNSTEIN; MELNIK, 2007). Uma razão pela qual a programação de dados não é fácil é que muitas vezes requer mapeamentos complexos entre diferentes representações de dados. Essas diferentes representações surgem por dois motivos: (i) heterogeneidade e (ii) *impedance mismatch* (do inglês, diferença entre dois paradigmas). A heterogeneidade surge da integração de fontes de dados independentes que foram desenvolvidas por pessoas diferentes e propósitos diferentes. As fontes de dados podem utilizar diferentes modelos de dados e esquemas (BERNSTEIN; MELNIK, 2007). *Impedance mismatch* é em parte conceitual: as linguagens de dados e de programação podem suportar diferentes paradigmas. A outra parte é estrutural, quando as linguagens não suportam os mesmos tipos de dados (COPELAND; MAIER, 1984).

Um sistema de gerenciamento de modelos (*Model Management System* ou MMS) é um componente que permite a criação, compilação, reuso, evolução e execução de mapeamentos entre esquemas representados por uma variedade de meta modelos (BERNSTEIN; MELNIK, 2007).

[4] https://www.mongodb.com/
[5] http://couchdb.apache.org/

De acordo com Bernstein e Melnik (2007) as principais abstrações suportadas por um MMS são esquemas e mapeamentos. Melnik (2004) define mapeamento como a correspondência semântica entre modelos. Bernstein e Melnik (2007) define um esquema como uma expressão que define um conjunto de instâncias possíveis, isto é, estados de bancos de dados.

A principal vantagem de um MMS é a possibilidade de realizar operações de escrita e leitura de maneira uniformizada. O desenvolvedor trabalha em uma camada mais abstrata, e o MMS se responsabiliza pela tradução das operações para as diferentes fontes de dados, utilizando para isso as informações dos esquemas e mapeamentos. De acordo com Bernstein e Melnik (2007) um MMS tem um objetivo de simplificar o desenvolvimento e manutenção de aplicações que executam programação de dados.

2.6 Álgebra ER de Parent e 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 aplicado em um modelo Entidade-Relacionamento. Apesar de ser um trabalho antigo e não se propor a resolver um problema dos atuais bancos NoSQL, a álgebra proposta permite que sejam especificadas consultas de forma independente do banco de dados utilizado. Ainda que seja um trabalho teórico, é possível implementar a semântica dessa álgebra com foco em um banco NoSQL, como será descrito na próxima seção.

A álgebra ER inclui os mesmos operadores básicos na álgebra relacional: união, diferença de conjuntos, produto cartesiano, projeção e seleção. Porém, estes se aplicam a Entidades e Relacionamentos, e não em relações (ou tabelas). Também 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.

2.6.0.1 Junção (*Relationship join*)

Assumindo que *E1, E2,...,En* são tipos de entidades associadas através de um relacionamento R, conforme mostra a Figura 13, a *junção* de E1,...,En através de R define um novo tipo de entidade E : $E = E_1 *_R (E_2, E_3, ..., E_n)$.

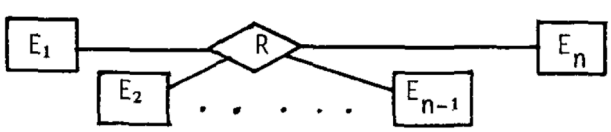

Figura 13 – Relacionamento R, relationship join (PARENT; SPACCAPIETRA, 1984)

Uma ocorrência de E consiste em uma ocorrência de E1 juntamente com todas as ocorrências (se houver) de R, com as entidades associadas E2,...,En nas quais a ocorrência 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 monovalorado, onde cada um desses atributos n, respectivamente, agrupa todos os atributos de R, E2,..., En.

O exemplo a seguir foi extraído do trabalho de Parent e Spaccapietra (1984): assumindo que o conteúdo da base de dados de exemplo representa os seguintes fatos:

- uma pessoa chamada SMITH tem um contrato com número 247 com uma empresa chamada LIFE-INS para o seu carro com número 1111 e outro contrato (número 3412) com a mesma empresa para o carro de número 2222;
- uma pessoa chamada CHEN tem um contrato de número 81 com LLOYD para o carro de número 1234.

Então, um novo tipo de entidade *PC* pode ser construída para armazenar os dados das pessoas juntamente com os dados de seus carros e empresa de seguro. A Figura 14 mostra o resultado da operação de *Junção*, onde "()" denotam valores para atributos complexos e "{}" denotam atributos multivalorados.

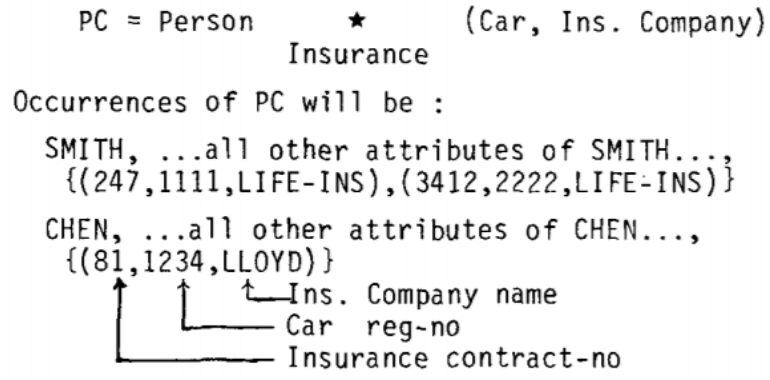

Figura 14 – Resultado da operação de Junção (PARENT; SPACCAPIETRA, 1984)

2.6.0.2 Produto cartesiano

Segundo (PARENT; SPACCAPIETRA, 1984) o produto cartesiano de duas entidades dos tipos E1, E2 define um novo tipo E: E = E1 * E2, cujo esquema é descrito na Figura 15.

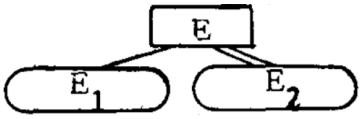

Figura 15 – Produto cartesiano entre duas entidades E1 e E2, produzindo uma nova entidade E (PARENT; SPACCAPIETRA, 1984)

Uma ocorrência de E consiste na ocorrência de E1 junto com todas as ocorrências de E2. Em outras palavras, E é um novo tipo de entidade que possui dois atributos:

- complexo, obrigatório, monovalorado, agrupando todos os atributos de E1
- complexo, opcional, multivalorado, agrupando todos os atributos de E2; o número de valores deste atributos é fixado e igual ao número de ocorrências de E2.

2.6.0.3 Projeção

O último operador que lida com a definição da estrutura da entidade resultante é o operador de projeção. Enquanto os dois operadores anteriores (junção e produto cartesiano) permitem a criação de entidades "maiores", a projeção é utilizada para "reduzir" uma entidade. A projeção consiste meramente na deleção de um grupo de atributos de uma entidade (PARENT; SPACCAPIETRA, 1984).

Sendo E1 uma entidade com os atributos a1,...,ai, ai+1, an. A entidade E projetada a partir de E1 é definida conforme Equação 2.3.

$$E = \Pi a_1, ..., a_i(E_1) \tag{2.3}$$

Todos os outros atributos (ai+1 até an) foram deletados. A entidade E herda todos os relacionamentos definidos em E1. Deve-se notar que a projeção pode deletar o atributo identificador de E1, o que pode criar instâncias idênticas de E (PARENT; SPACCAPIETRA, 1984).

2.6.0.4 Seleção

Seja E1 um tipo de entidade e c uma expressão de comparação no formato <operando operador operando>, onde o operando é o nome de um atributo ou valor ou conjunto de valores, e operador um operador de comparação aritmético (=, !=, >, >=, <, <=) ou um operador de conjunto. Ou também um conjunto de expressões de comparação ligadas por operadores lógicos (PARENT; SPACCAPIETRA, 1984).

$$E = \sigma_c(E_1) \tag{2.4}$$

A Equação 2.4 define um tipo de entidade E cuja descrição é idêntica a E1 e cuja população é restrita as ocorrências de E1 onde as condições expressas por c são satisfeitas (PARENT; SPACCAPIETRA, 1984).

2.6.0.5 Redução

Uma operação de seleção não modifica as ocorrências selecionadas. Em alguns casos o usuário pode desejar manter, no caso de atributos multivalorados, apenas aqueles que satisfazem determinados critérios. Este é o proposito do operador de redução (PARENT; SPACCAPIETRA, 1984).

$$E = \rho_{cr}(E_1) \tag{2.5}$$

De acordo com Parent e Spaccapietra (1984) a Equação 2.5 mostra que a operação de redução produz uma nova entidade E cujo esquema é idêntico ao definido por E1 e suas ocorrências são as mesmas de E1 exceto os registros que não passaram nas condições definidas, ou seja, o resultado é um subconjunto das ocorrências de E1 que atendam aos critérios expressos em "E".

2.6.0.6 Compressão

De acordo com com Parent e Spaccapietra (1984), duas ocorrências de um mesmo tipo de entidade são ditas idênticas se cada um de seus atributos possuem o mesmo valor.

Duas ou mais ocorrências podem ser idênticas, seja por definição ao inserir os dados ou como resultado de operações (especialmente projeções) em entidades bases. O operador de projeção é definido segundo a Equação 2.6 (PARENT; SPACCAPIETRA, 1984).

$$E = \gamma(E_1) \tag{2.6}$$

A equação define um novo tipo de entidade E, cuja descrição é idêntica a E1 e possui todas as ocorrências de E1, menos as que são idênticas a ocorrências de E1 que já foram consideradas (PARENT; SPACCAPIETRA, 1984).

Parent e Spaccapietra (1984) também diz que a ocorrência que causou a remoção de outra ocorrência, herda seus relacionamentos, limitando assim a perda semântica a apenas o número de ocorrências idênticas.

2.6.0.7 União e diferença

A operação de união mescla resultados de dois conjuntos em um único grupo e a operação de diferença remove todos os elementos de um grupo que também aparece em

outro grupo (PARENT; SPACCAPIETRA, 1984).

Para que essas operações possam ser aceitas, as duas entidades em questão devem ser compatíveis. Assumindo que a entidade E1 possui os atributos (a1,a2,...,an) e a entidade E2 possui os atributos (b1,b2,...,bn). E1 é dita compatível com E2 se ai e bi forem compatíveis, por exemplo, ambos atributos estão definidos no mesmo domínio e são simples ou complexos com atributos compatíveis (PARENT; SPACCAPIETRA, 1984).

Se E1 e E2 são compatíveis, então:

$$E(c_1, c_2, ..., c_n) = E_1 \cup E_2 \tag{2.7}$$

De acordo com a Equação 2.7 E define uma nova entidade, compatível com E1 e E2, cujos atributos são nomeados c1,...,cn, c possui o mesmo domínio de ai (PARENT; SPACCAPIETRA, 1984).

Parent e Spaccapietra (1984) definem a operação de diferença conforme a Equação 2.8:

$$E(c_1, c_2, ..., c_n) = E_1 - E_2 \tag{2.8}$$

"E" definindo uma entidade que representa o grupo de ocorrências de E1 que não possui ocorrência idêntica em E2.

O trabalho de Parent e Spaccapietra (1984) define a álgebra e a semântica das operações, porém não define uma sintaxe concreta para a escrita de consultas, e nem a semântica operacional necessária para a execução das consultas em um banco de dados real.

Pensando em uma implementação concreta, para bancos de dados relacionais, há uma limitação de cunho prático: não é possível criar atributos multivalorados em uma tabela, sendo necessária uma tabela adicional. Já para bancos NoSQL, essa limitação não existe, portanto a implementação de uma semântica operacional fica mais fácil.

2.6.0.8 Cardinalidade no modelo de Parent e Spaccapietra (1984)

Parent e Spaccapietra (1984) definem a operação de junção (*Relationship Join*) como a junção de N entidades através de um determinado relacionamento conforme expressão abaixo:

$$E = E_1 *_R (E_2, E_3, ..., E_n). \tag{2.9}$$

A composição de E se dá conforme as seguintes regras: (i) para cada ocorrência de *E1* há uma ocorrência de E, (ii) uma ocorrência de E consiste de uma ocorrência de

E1 com todas as ocorrências (se houver) de *R* na qual *E1* está envolvida, junto com as ocorrências associadas de *E2*, *E3*, ..., *En*.

Nota-se que a álgebra fala apenas em ocorrências, ou seja, a cardinalidade do relacionamento não é levada em consideração. A cardinalidade é importante para a tomada de decisão no momento do mapeamento entre o modelo conceitual (ER) e o modelo lógico pois especifica o número máximo de instâncias do relacionamento no qual uma entidade pode participar (ELMASRI et al., 2010). Além disso, Fraser et al. (2012) afirmam que a otimização de consultas dependem de um modelo de estimativa de cardinalidade preciso para produzir planos de execução confiáveis e eficientes. Essa afirmativa é importante em uma abordagem prática da álgebra (implementação). Como no trabalho de Parent e Spaccapietra (1984) a preocupação é apenas com o modelo conceitual (ER) e com a semântica das operações, e não com sua implementação, os autores assumem que o modelo ER não precisa conter informação relativa à cardinalidade dos relacionamentos.

Neste trabalho, no entanto, a informação de cardinalidade precisa ser considerada em algum momento, uma vez que está propondo a implementação da álgebra. Conforme discutido em maiores detalhes no Capítulo 4, essa decisão ficará por conta do projetista.

3 Trabalhos relacionados

Neste capítulo são apresentados trabalhos relacionados ao tema dessa pesquisa. São apresentados os trabalhos de Bernstein e Melnik (2007), Li, Gu e Zhang (2014), Liang, Lin e Ding (2015), Schreiner, Duarte e Mello (2015), Atzeni, Bugiotti e Rossi (2012), Parent e Spaccapietra (1984) e Noguera (2018). Estes trabalhos serão descritos em detalhes abaixo.

3.0.1 Bernstein e Melnik (2007)

O trabalho de Bernstein e Melnik (2007) traz uma revisão do conhecimento sobre o gerenciamento de modelos, faz também uma revisão da visão sobre o problema e identifica novos desafios para a pesquisa do tema abordado.

No trabalho são descritos dois cenários principais para a design / geração de mapeamentos: (i) presentes dois esquemas, um arquiteto de dados define um mapeamento entre eles, (ii) dado um esquema é gerado um esquema derivado junto com o mapeamento para o esquema original.

Para o primeiro cenário o processo de design de um mapeamento é descrito através de três passos:

- Correspondências: pares de elementos dos dois esquemas que acredita-se estarem relacionados.
- Tradução: as correspondências são traduzidas em regras/restrições de mapeamento.
- Transformação: em alguns casos a regras de mapeamento não são suficientes sendo necessário executar uma transformação.

No segundo cenário, o processo foi chamado de *ModelGen*, que é uma operação que traduz automaticamente um esquema definido por um metamodelo em outro esquema equivalente definido por um metamodelo diferente.

O trabalho também cita formas de automatizar a geração de transformações (para o design de mapeamentos) de acordo com o uso e contexto dos dados. Outro ponto abordado pelo trabalho é possíveis problemas durante a execução de um mapeamento, isto é, quando há necessidade de executar alguma ação sobre o esquema origem ou alvo.

Outro ponto abordado pelo trabalho é o problema de compatibilidade quando há mudanças no esquema utilizado. O trabalho diz que muitas abordagens sobre este problema requerem a manipulação de mapeamentos, e essas manipulações podem ser

abstraídas como uma sequencia de operações de gerenciamento de modelos, divididas da seguinte maneira:

- Composição: um novo mapeamento é construído a partir da composição do mapeamento entre o esquema antigo e o novo esquema.
- Diferença: essa operação utilizando o novo esquema e o mapeamento entre o esquema original e o novo, resultaria em um novo esquema contendo trechos do novo mapeamento que não estão no mapeamento de entrada e um novo mapeamento descrevendo a intersecção do novo esquema e do resultado da operação.
- Fusão (*merge*): combina dois esquemas utilizando a descrição dos esquemas e o mapeamento entre eles. O resultado é um novo esquema (fusão entre as entradas) incluindo mapeamentos para os esquemas utilizados na fusão.
- Inversão: operação capaz de obter o resultado original após uma transformação ser aplicada.

O trabalho conclui que ainda não há solução para o problema. Os autores dizem que ainda estamos no estágio de reutilizar algoritmos e designs de mapeamentos para solucionar cada novo problema prático e não no ponto de reutilizar componentes prontos para solucioná-los.

A abordagem de gerenciamento de modelos compartilha muitas similaridades com este trabalho, no sentido de estabelecer um mapeamento entre diferentes representações de dados. Pode-se dizer que este trabalho é de fato uma aplicação dos princípios de gerenciamento de modelos. Há algumas diferenças, no entanto. A primeira diferença é de ordem prática. O foco aqui é em NoSQL, ou seja, estamos aplicando um conceito já existente em uma tecnologia mais recente. Em teoria não há diferenças significativas, mas na prática as particularidades dos bancos NoSQL exigem um cuidado na implementação das operações das consultas, em particular no caso de atributos multivalorados e coleções embutidas.

A segunda diferença é que envolvemos apenas dois esquemas: o modelo ER, utilizado para análise dos requisitos de armazenamento e especificação de consultas; e o esquema de um banco de dados baseado no MongoDB, que é a implementação. Neste trabalho, adota-se uma abordagem muito próxima ao passo de correspondência entre pares de elementos dos dois esquemas. Porém, até o momento, não identificou-se necessidade de utilizar os passos de tradução e transformação, já que a correspondência foi suficiente para o propósito de facilitar a escrita das consultas. Esse foco em apenas dois esquemas torna esta solução menos abrangente, mas mais focada no problema da escrita das consultas, o que facilitou sua implementação.

A terceira diferença é que aqui adota-se uma abordagem generativa. Ou seja, as consultas são geradas. Após a geração, elas funcionam de maneira independente do sistema de gerenciamento de modelos (MMS). Podem inclusive ser adaptadas manualmente, caso necessário, apesar de não ser recomendado para manter o ciclo de evolução mais gerenciável. Isso dá uma liberdade maior para o desenvolvedor, tanto para ajustar as consultas como para execução: esta abordagem pode ser utilizada no MongoDB sem nenhum tipo de extensão ou necessidade de componente em tempo de execução.

A quarta e última diferença é que, neste trabalho explorou-se exclusivamente as operações de leitura, excluindo-se as operações de escrita, que estão previstas nos MMS. Porém, a princípio elas poderiam ser geradas da mesma forma, o que pode ser investigado em trabalhos futuros.

3.0.2 Li, Gu e Zhang (2014)

O trabalho de Li, Gu e Zhang (2014) propõe a transformação de dados representados por diagramas de classe para um formato compatível com o banco de dados HBase (NoSQL).

A proposta utiliza o metamodelo UML como meta-meta-modelo e subconjuntos UML como meta-modelos capazes de representar as estruturas de origem (diagrama de classes) e destino (HBase). A etapa seguinte do trabalho é definir as regras de transformação entre os meta-modelos de origem e de destino e por final, será gerado um modelo HBase a partir de um diagrama de classes UML através da transformação de modelos.

O objetivo principal deste trabalho é propor regras para transformação entre o metamodelo UML e o metamodelo HBase (subconjunto do metamodelo UML). O trabalho mostra que é possível essa transformação mas não considerou qualquer possibilidade de erro ou qualquer tipo de interferência que possa alterar o resultado da transformação. Além disso, a abordagem assume sempre o mesmo mapeamento entre classes e tabelas, ou seja, o conjunto de regras proposto não permite que o mapeamento seja alterado sem que seja necessário alterar o código de transformação. Nota-se também que não houve abordagem sobre o impacto da proposta nas consultas ao banco de dados.

3.0.3 Liang, Lin e Ding (2015)

O trabalho realizado por Liang, Lin e Ding (2015) propõe o uso de um modelo intermediário para a migração de dados de um banco relacional para um banco NoSQL, com o proposito de criar um processo genérico para migração de dados.

Na Figura 16 pode-se ver a estrutura do modelo. O modelo é formado por objetos. Cada objeto é uma coleção relacionada a uma entidade em um banco de dados relacional, incluindo as propriedades da entidade, relacionamentos e artefatos sobre os dados (*Data*

feature) e uma lista de consultas executadas (*Query feature*). Os artefatos representam características dos dados como, por exemplo: "frequentemente inseridos". A lista de consultas (*Query features*) representam uma lista das consultas mais utilizadas em relação ao modelo. Além do modelo, o trabalho sugere o uso de "bibliotecas de estratégia", que são responsáveis por realizar tarefas específicas a um banco de dados NoSQL.

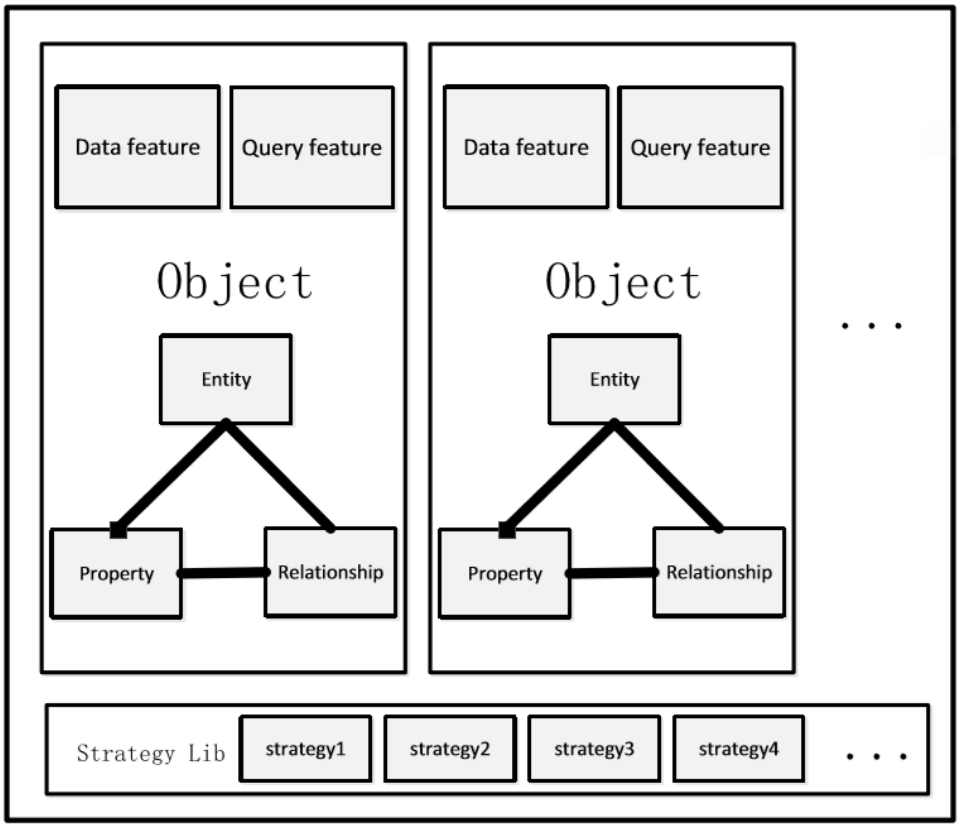

Figura 16 – Estrutura do modelo intermediário (LIANG; LIN; DING, 2015)

O processo de transição ocorre da seguinte maneira: o modelo intermediário recebe como entrada um modelo de dados relacional, podendo incluir diversas tabelas e relacionamentos. Será extraído um objeto para cada entidade, conforme Figura 17.

O trabalho mostra que é possível realizar a migração de dados entre uma base relacional e uma não relacional. Esse modelo pode ser utilizado como ferramenta para importar ou exportar dados entre diferentes tipos de bancos de dados e também como uma interface para desenvolvimento de outras ferramentas.

A proposta não permite que o resultado da transição de um banco relacional para um banco de dados NoSQL seja alterado de acordo com o modelo de entrada, ou seja, as regras definidas nas bibliotecas de estratégia serão aplicadas a todos os modelos de entrada e o resultado seguirá as mesmas diretrizes. O trabalho não deixa claro se há geração de consultas para acesso aos dados após a transição para o banco de dados NoSQL. Assume-se então que consultas à nova fonte de dados não é abordada pelo trabalho.

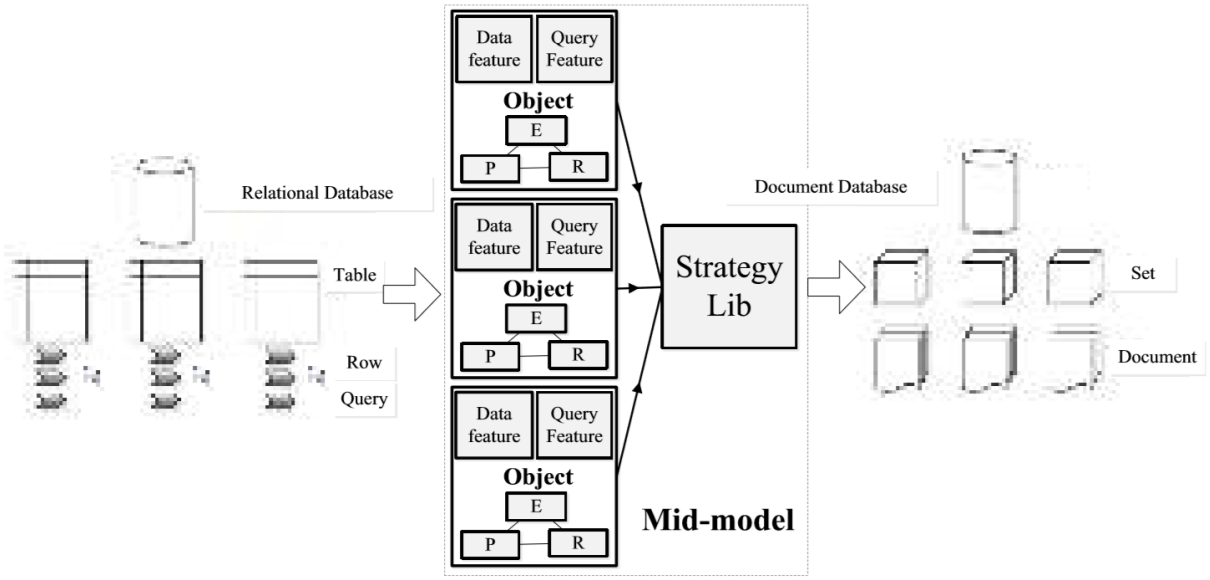

Figura 17 – Processo de transição utilizando o modelo intermediário (LIANG; LIN; DING, 2015)

3.0.4 Schreiner, Duarte e Mello (2015)

O trabalho de Schreiner, Duarte e Mello (2015) propõe a tradução de um esquema relacional e também comandos SQL para um banco de dados NoSQL dos tipos documentos, família de colunas e chave-valor.

Foi proposta a transformação de dados relacionais em um modelo canônico que será então transformado em um modelo compatível com um banco NoSQL. Os comandos SQL serão então convertidos em métodos de acesso, como uma API REST.

A Figura 18 mostra em (A) o modelo relacional, e em (B) sua representação no formato canônico.

O mapeamento do modelo canônico (Figura 18) para um modelo Documentos, definido na Figura 19 (NoSQL) procede da seguinte maneira: (i) o esquema canônico representa um banco de dados orientado a documentos; (ii) cada chave de primeiro nível pertencente ao esquema gera uma coleção de documentos; (iii) para cada chave de segundo nível que é filha de uma chave de primeiro nível gera-se um documento; (iv) para cada chave de terceiro nível gera-se um par atributo-valor pertencente ao documento.

As instruções DDL e DML também são mapeadas. Informações DDL são utilizadas para gerar e manipular um dicionário que contém os dados do esquema relacional, e suportam os seguintes comandos: *CREATE TABLE*, *ALTER TABLE* e *DROP TABLE*. As instruções DML realizam consultas no dicionário a fim de criar métodos compatíveis com uma API REST para ser executada no banco de dados NoSQL.

O trabalho é composto de 7 módulos, sendo uma interface de acesso que envia

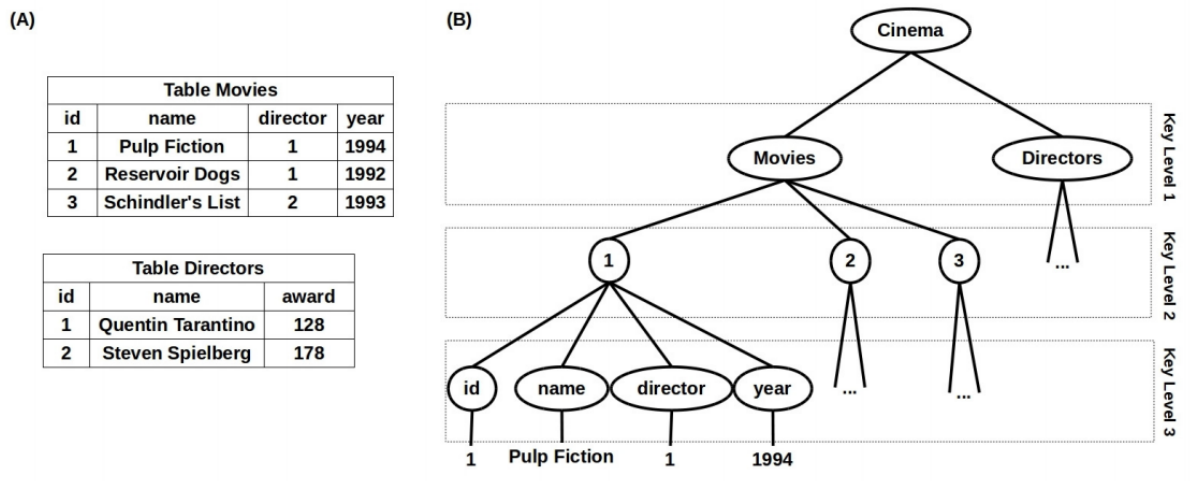

Figura 18 – Modelo relacional (A) e modelo canônico (B) (SCHREINER; DUARTE; MELLO, 2015)

instruções SQL para o SQL Parser e exibe o resultados gerados pelo módulo de execução. O parser envia os comandos para os módulos Translate e Query planner de acordo com o tipo de instrução. Os dados então são enviados para o módulo de execução e depois para o de comunicação que se conecta a um banco de dados NoSQL, conforme Figura 20.

O trabalho oferece uma forma de mapear comandos SQL para bancos de dados NoSQL dos tipos documento, família de colunas e chave-valor. O trabalho não oferece suporte para todas as instruções SQL, gerenciamento de índices e operações de junção.

É importante notar que o modelo canônico gerado seguirá sempre as mesmas regras independentemente do modelo de entrada.

3.0.5 Atzeni, Bugiotti e Rossi (2012)

Atzeni, Bugiotti e Rossi (2012) propuseram uma API onde bancos de dados não relacionais podem ser uniformemente definidos, consultados e acessados por uma aplicação. Para lidar com os dados semi-estruturados, a API utiliza uma representação intermediária implementada em JSON.

Na Figura 21 observa-se que a aplicação se comunica com a interface SOS (Save Our Systems) através de requisições HTTP (Hypertext Transfer Protocol), as requisições podem ser dos tipos PUT (inserir ou atualizar), DELETE (deletar) e GET (solicitar). Internamente os dados recebidos são serializados em JSON (Javascript Object Notation) para serem processados pela camada intermediária e depois convertidos em um formato compatível com o banco de dados em uso.

A proposta permite o acesso uniforme a bancos de dados NoSQL como Redis, HBase e MongoDB. É possível o uso de outras bases não relacionais com a interface SOS,

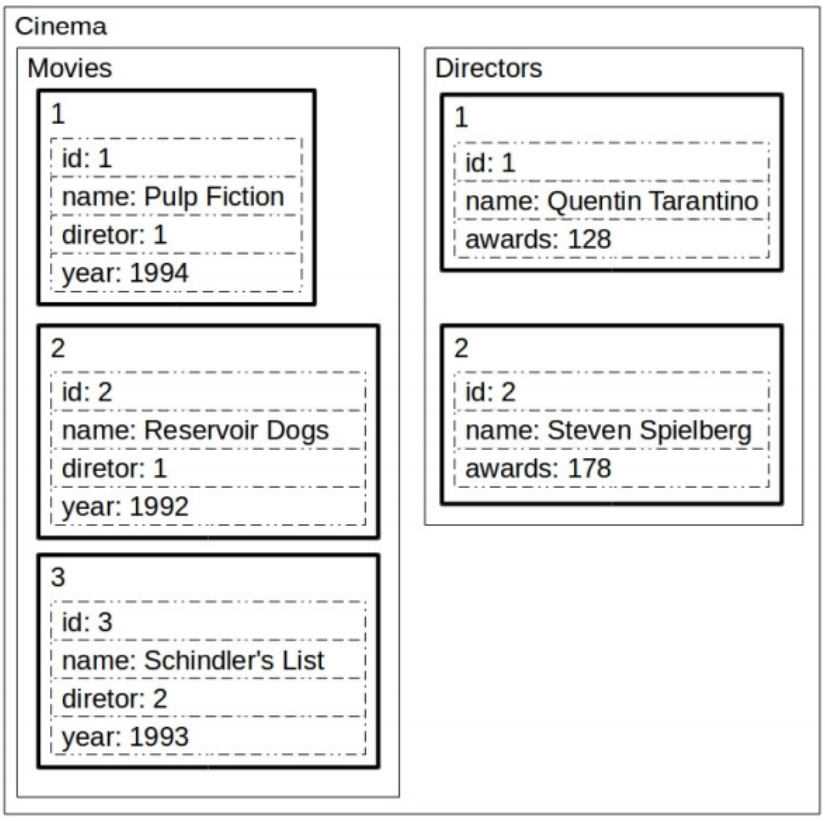

Figura 19 – Modelo documento-nosql gerado através do modelo canônico (SCHREINER; DUARTE; MELLO, 2015)

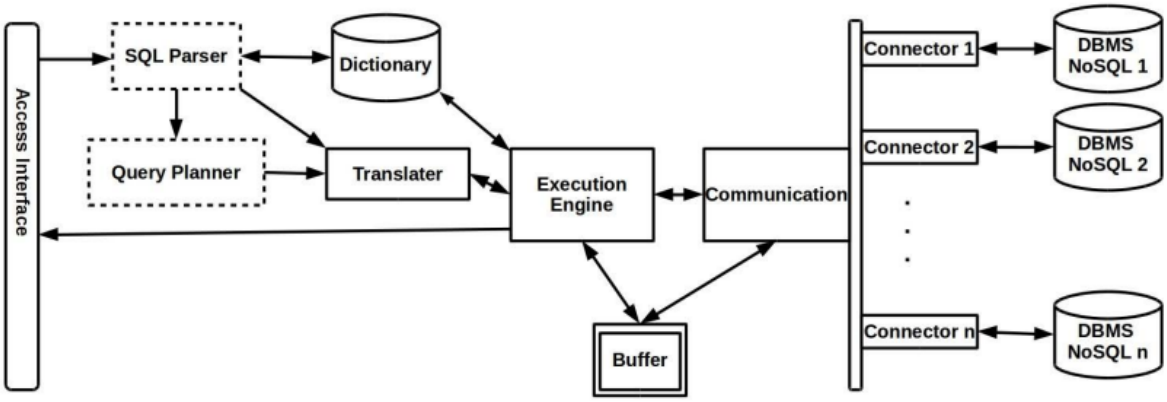

Figura 20 – Arquitetura SQLToKeyNoSQL (SCHREINER; DUARTE; MELLO, 2015)

sendo necessário a implementação de interpretadores para cada banco de dados.

Enquanto o trabalho dá suporte a operações básicas (PUT, GET e DELETE como exibido na Figura 21), operações mais complexas como a junção precisam ser construídas manualmente.

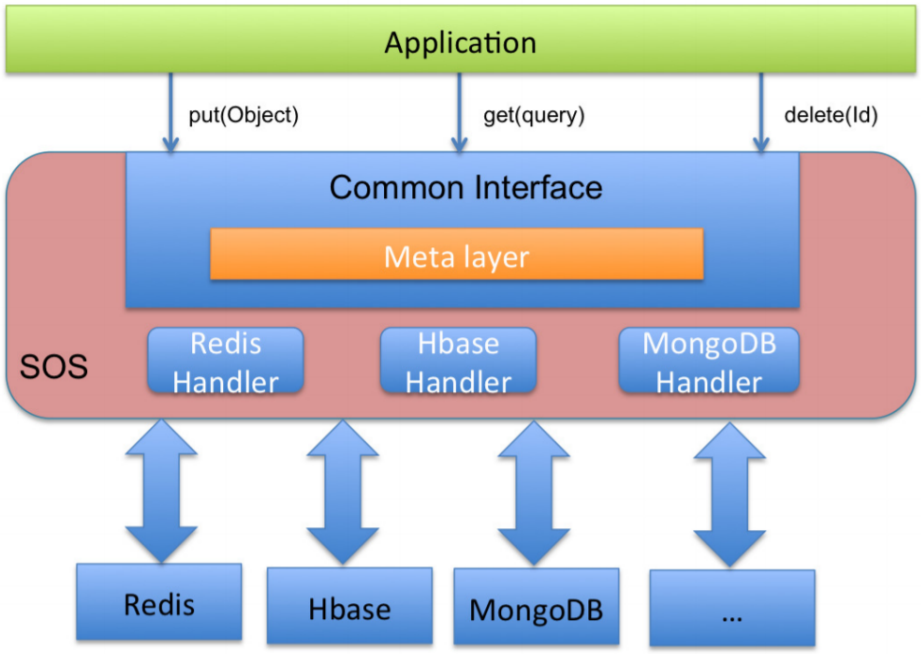

Figura 21 – Arquitetura SOS (Save Our Systems) (ATZENI; BUGIOTTI; ROSSI, 2012)

3.0.6 Noguera (2018)

O trabalho de Noguera (2018), Noguera e Lucrédio (2019) criou uma linguagem para escrita de consultas baseada na álgebra proposta por Parent e Spaccapietra (1984), tendo os seguintes objetivos: (i) permitir que desenvolvedores especifiquem consultas de forma independentemente do esquema utilizado; (ii) gerar consultas automaticamente em código nativo do banco de dados utilizado, considerando todos os possíveis mapeamentos e as formas em que as entidades e relacionamentos podem ser armazenados; (iii) sempre produzir o mesmo resultado, independentemente do mapeamento ou da implementação da consulta.

Noguera (2018) desenvolveu um algoritmo que gera consultas automaticamente de acordo com os mapeamentos possíveis entre o modelo ER e o banco de dados orientado a documentos (NoSQL) utilizado. Durante o desenvolvimento, foi descoberto que é possível gerar diferentes consultas para um mesmo mapeamento. O algoritmo gera todas as possibilidades, tomando cuidado para sempre produzir o mesmo resultado, como definido pela álgebra.

```
1    FROM Person RJOIN (Drives Car) SELECT *
```

Listagem 3.1 – Exemplo de consulta (NOGUERA, 2018)

Um importante diferencial do trabalho de Noguera (2018) em relação aos trabalhos relacionados é a possibilidade de se criar diferentes mapeamentos para um mesmo modelo conceitual. A Figura 22 mostra um exemplo de um modelo conceitual e possíveis mapeamentos para um banco de dados orientado a documentos (NoSQL). O mapeamento

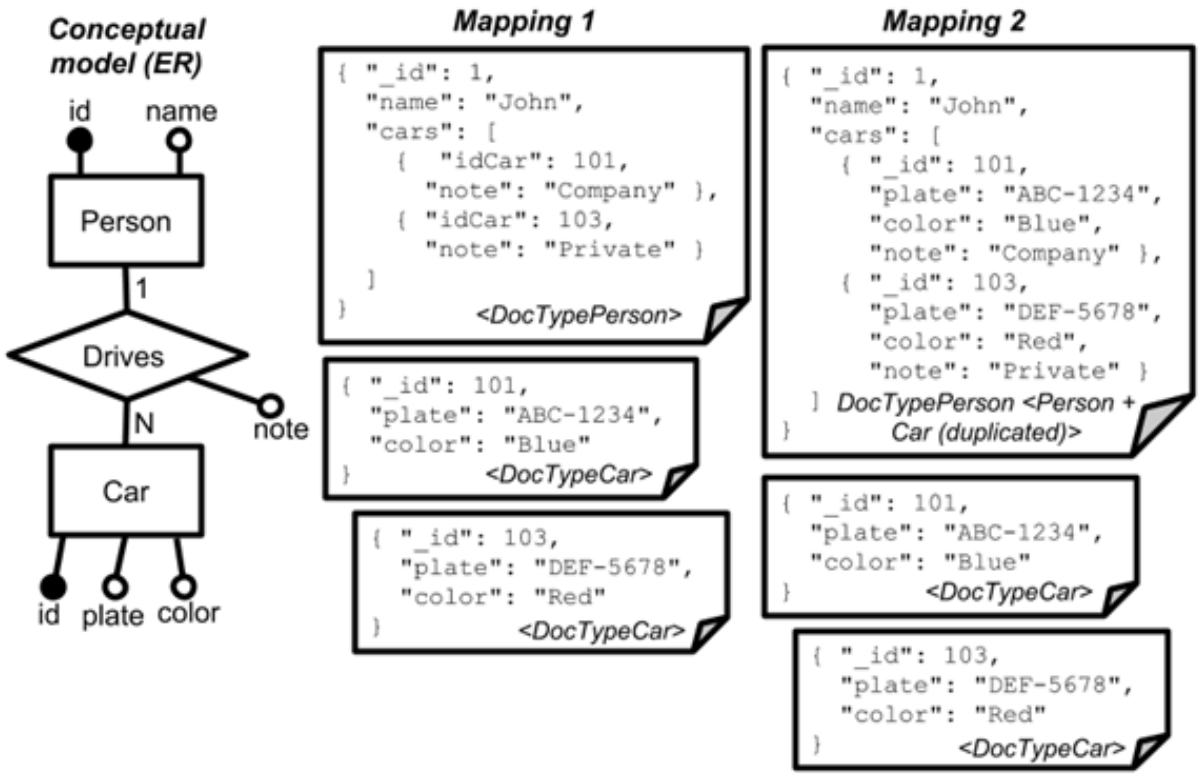

Figura 22 – Duas possíveis modelagens de um único modelo conceitual para um banco de dados orientado a documentos (NOGUERA, 2018)

1 (Mapping 1), por exemplo, define um tipo de documento para cada entidade. Já o mapeamento 2 (Mapping 2) define que a entidade "Car" é armazenada de forma duplicada, em dois tipos de documentos ("DocTypePerson" e "DocTypeCar"). Outros mapeamentos são possíveis, visando melhorar o desempenho em algumas consultas. A maior parte dos trabalhos relacionados trabalha em cima de um único tipo de mapeamento, que é fixo.

A Listagem 3.1 mostra um exemplo da sintaxe concreta definida por Noguera (2018) para a álgebra de Parent e Spaccapietra (1984). Em termos gerais, a principal diferença entre a linguagem proposta e outra mais conhecida, é que esta irá sempre resultar em uma única entidade. Se o termo "RJOIN" não for informado, o resultado é a própria entidade, caso contrario o resultado será uma entidade computada como definido por Parent e Spaccapietra (1984).

É importante mencionar que a sintaxe da operação "RJOIN" inclui o relacionamento que foi utilizado para estabelecer a junção, isso é necessário quando duas entidades possuem mais de um relacionamento entre elas (NOGUERA, 2018).

Segundo Noguera (2018) a ideia geral do algoritmo é analisar minuciosamente todas as possibilidades de mapeamento entre o modelo ER e os tipos de documentos do MongoDB. Até este momento foram analisados apenas relacionamentos binários, já que esta é a parte mais complexa da álgebra.

O algoritmo aceita três entradas, o modelo ER conceitual, o esquema MongoDB e o mapeamento entre eles. As entradas são especificadas como modelos orientados a objetos com referências cruzadas entre si. Baseado nessas informações o algoritmo tenta analisar para cada par de entidades relacionadas, em qual combinação elas se enquadram, então é gerada a consulta correspondente para o MongoDB (NOGUERA, 2018).

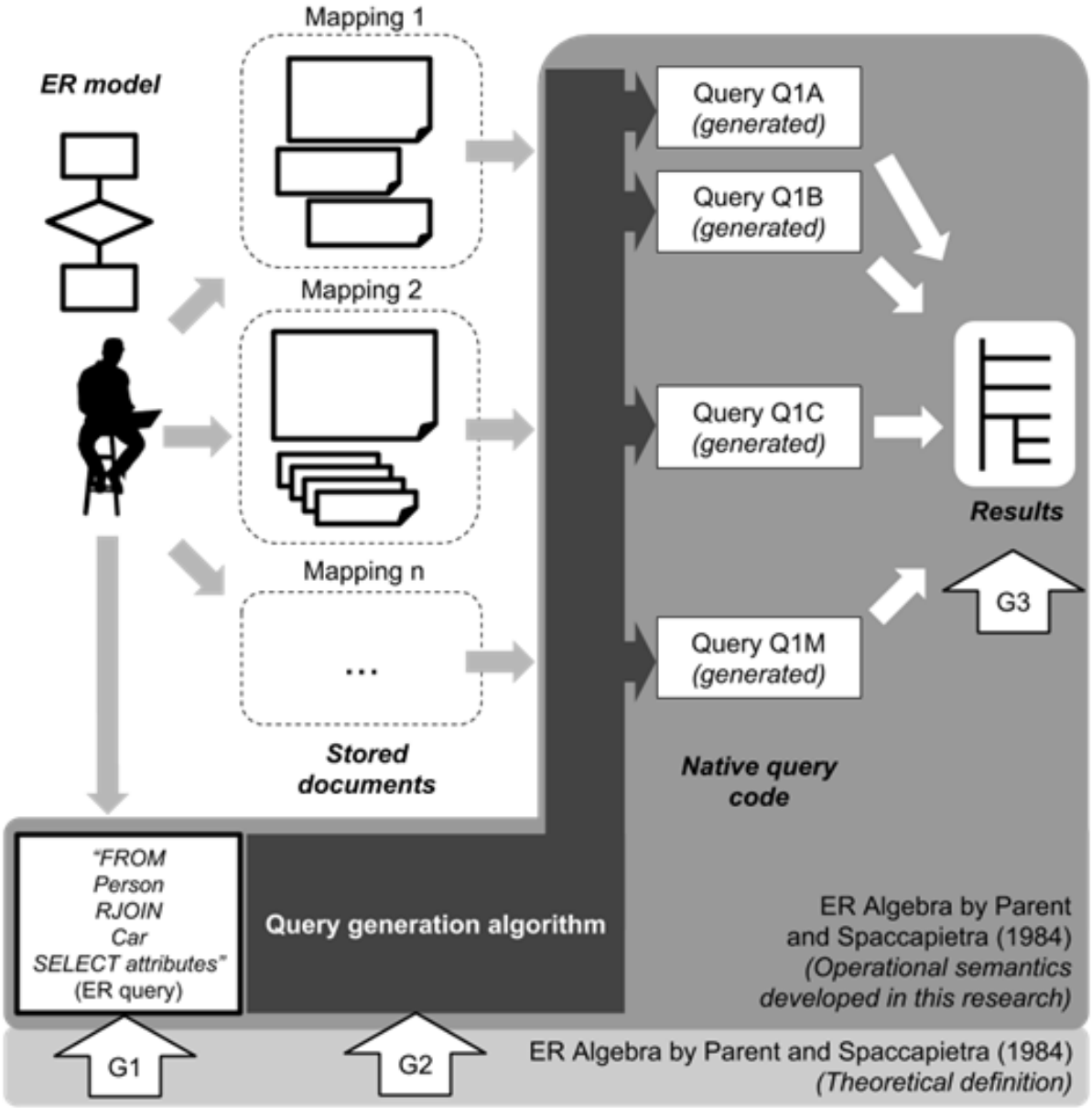

Figura 23 – Visão geral da proposta (NOGUERA, 2018)

A Figura 23 mostra a visão geral da proposta, desde a consulta na linguagem proposta até a geração de consultas para o banco de dados orientado a documentos.

De acordo com Noguera (2018) o projeto lidou com a operação de junção entre duas entidades, e os outros operadores devem ser implementados para que outros testes possam ser realizados e o desempenho das consultas geradas não foi considerado.

3.0.7 Considerações finais

Pode-se notar que os trabalhos focam em migração de dados para uma base NoSQL, como o trabalho de Li, Gu e Zhang (2014) que buscou o uso de modelos para transformar dados relacionais em um modelo HBase, e o trabalho de Liang, Lin e Ding (2015) que propôs o uso de um modelo intermediário para executar a migração entre bases relacionais e não relacionais.

Os trabalhos de Li, Gu e Zhang (2014), Liang, Lin e Ding (2015) e Schreiner, Duarte e Mello (2015) utilizam regras fixas para execução de seus processos, ou seja, independentemente da entrada de dados a forma de processar as informações é sempre a mesma, o que impossibilita que o desenvolvedor faça ajustes finos no processo para um determinado mapeamento de dados sem que seja necessário alterar as regras de execução.

É importante notar que os trabalhos de Curé et al. (BQL) e Sellami, Bhiri e Defude (ODBAPI) são mais flexíveis quanto a geração de consultas, não dependendo de regras fixas, mas são limitados a consultas simples (Curé et al. 2011, R. Sellami, S. Bhiri, B. Defude, 2014, apud (NOGUERA, 2018)).

Os trabalhos de Schreiner, Duarte e Mello (2015) e Atzeni, Bugiotti e Rossi (2012) buscaram uniformizar o acesso aos bancos de dados NoSQL, porém não abordaram o uso de consultas complexas.

Parent e Spaccapietra (1984) propuseram uma álgebra ER, definindo uma série de operadores, como junção, seleção e projeção que podem atuar em entidades e relacionamentos. O trabalho Noguera (2018) criou uma linguagem baseada na proposta de Parent e Spaccapietra (1984) e sugere a criação de um algoritmo para geração de consultas a partir de um modelo conceitual, o esquema MongoDB e o mapeamento entre o modelo e o esquema. Porém, nem todos os operadores foram implementados, o que é necessário para que se possa realizar mais testes e experimentos visando melhoria de desempenho, que é o objetivo final da pesquisa.

Para melhor contextualizar a relação entre os trabalhos citados e esta pesquisa, a Tabela 2 compara de forma resumida as contribuições de cada um dos trabalhos, em comparação com este.

Trabalho	Consultas complexas	Suporte a múltiplos bancos de dados	Suporte a todas operações CRUD	Flexibilidade de mapeamento	Análise de desempenho	Código gerado / Runtime
Li, Gu e Zhang (2014)	Trabalho trata apenas de correspondencia dos modelos UML e Hbase	Apenas Hbase	Não se aplica	Não	Não	Nâo há informações sobre geração de código
Liang, Lin e Ding (2015)	Não abordado pelo trabalho	Sim, permite implementação de bibliotecas para suportar diferentes bancos de dados	Sim	Não	Não	Não há informações sobre geração de código
Schreiner, Duarte e Mello (2015)	Apenas consultas simples (sem junções)	Qualquer banco de dados NoSQL dos tipos Chave-Valor, Família de colunas ou Documentos	Sim, incluindo instruções DDL	Não, modelo canônico sempre segue as mesmas regras	Não	Biblioteca runtime
Atzeni, Bugiotti e Rossi (2012)	Apenas consultas simples (sem junções)	Sim, permite implementação de bibliotecas para suportar diferentes bancos de dados	Sim	Não se aplica	Não	Biblioteca runtime
Noguera (2018)	Sim, mas apenas junções binárias	Apenas MongoDB	Apenas leitura	Sim	Não	Código gerado
Cabral (2020)	Sim, incluindo junções compostas	Apenas MongoDB	Apenas leitura	Sim	Sim, comparação com desempenho de consultas escritas à mão	Código gerado

Tabela 2 – Comparação entre os trabalhos

Este trabalho se assemelha aos conceitos de gerenciamento de modelos e em alguns pontos similares as propostas discutidas por Bernstein e Melnik (2007). As abordagens citadas por Bernstein e Melnik (2007) discutem desde criação de mapeamentos entre dois esquemas por um arquiteto de dados a maneiras de gerar automaticamente um mapeamento a partir de dois esquemas e também tópicos relacionados a execução de consultas através dos mapeamentos (incluindo alterações na definição de dados/esquemas) e também mecanismos de propagação da evolução de um esquema.

Este trabalho propõe a geração automatizada de consultas nativas para SGBD MongoDB. Essas consultas são geradas a partir de um esquema representado por um Modelo ER, um Esquema MongoDB, o mapeamento entre eles e de uma consulta escrita para o esquema representado pelo Modelo ER. Se o Esquema MongoDB é alterado (evolução) é necessário que o mapeamento seja modificado para refletir as alterações no esquema mas a consulta escrita para o esquema representado pelo Modelo ER não é alterada.

4 Uma álgebra ER para consultas em bancos de dados NoSQL: implementação de operadores adicionais e análise de desempenho

Como descrito no Capítulo 1 nota-se que é possível mapear um modelo ER de diferentes maneiras em um banco de dados NoSQL orientado a documentos, neste caso o MongoDB. Para cada representação do modelo ER no MongoDB é necessário que as consultas sejam escritas de acordo com a estrutura presente no MongoDB, o que pode causar grande esforço e retrabalho caso seja necessária alguma alteração nesse mapeamento.

O trabalho anterior de Noguera (2018) aborda esse tema ao propor uma linguagem para escrita de consultas a partir de um modelo ER que tem como base a álgebra proposta por Parent e Spaccapietra (1984). O objetivo desta linguagem é permitir que dado um modelo ER uma consulta irá produzir o mesmo resultado independentemente do esquema de dados do banco de dados utilizado. No entanto, o trabalho anterior de Noguera (2018) apresentava algumas limitações.

Este trabalho, partindo da proposta de Noguera (2018) teve como objetivo completar os resultados obtidos anteriormente, acrescentando mais operadores da álgebra ER e realizando mais análises. Os resultados obtidos são divididos em quatro partes, conforme descrito a seguir, com destaque para as semelhanças e diferenças entre este trabalho e o anterior:

1. Desenvolvimento de um conjunto de metamodelos para a especificação de um modelo conceitual (ER), um modelo representando o esquema no MongoDB (MDB) e um modelo intermediário conectando os elementos do modelo ER ao esquema MongoDB (MAP). Esses metamodelos já haviam sido desenvolvidos por Noguera (2018) e foram feitas algumas alterações como a remoção da cardinalidade no metamodelo ER, detalhes serão discutidos na próxima seção.

2. Desenvolvimento de um algoritmo que, a partir de uma consulta Q e utilizando os modelos ER, MDB e MAP, gera código JavaScript. Este, quando executado no MongoDB, produz o mesmo resultado independentemente do mapeamento informado. Um algoritmo já havia sido desenvolvido por Noguera (2018) mas havia o suporte para operações de junção entre duas entidades apenas. Nesta pesquisa o algoritmo foi aprimorado e estendido para incluir os operadores propostos por Parent e Spaccapietra (1984) e que não haviam sido contemplados: (i) junção composta (entre mais de duas entidades), (ii), projeção, (iii) seleção e (iv) produto cartesiano;

3. Análise automatizada de consistência do resultado do código gerado pelo algoritmo. No trabalho de Noguera (2018) já haviam sido feito testes, mas de forma manual. Além disso, devido ao limitante nos operadores, não haviam sido feitos testes com consultas mais complexas, apenas aquelas envolvendo duas entidades. Aqui foram feitos mais testes e análises, com consultas mais completas, elevando a confiança nas evidências obtidas.

4. Análise do desempenho do código gerado pelo algoritmo comparado com o desempenho de consultas escritas por um engenheiro de software. No trabalho de Noguera (2018) não houve preocupação com o desempenho do código gerado. Neste trabalho foi analisado o impacto da estrutura de resultado proposta pela álgebra no desempenho da consulta no banco de dados.

A seguir apresentam-se mais detalhes sobre estes resultados.

4.1 Metamodelos

São utilizados três metamodelos para representar o modelo Entidade-Relacionamento (ER), o esquema MongoDB e o mapeamento entre os modelos ER e Esquema MongoDB. Estes metamodelos já haviam sido desenvolvidos por Noguera (2018) e foram reutilizados parcialmente neste trabalho. As mudanças em relação aos metamodelos criados por Noguera (2018) são:

- Metamodelo ER: Entidade e Relacionamento são derivados de um mesmo elemento chamado *BaseERElement* que compõem um *ERModel*. Foram removidas referências à cardinalidade de relacionamentos. Atributos não são mais separados entre e simples e composto.

- Metamodelo Esquema MongoDB: Foram removidas as referências à tipagem de atributos, os atributos não são mais separados em *atômico* (simples) ou incorporado (*embedded*). O esquema (*MongoSchema*) é composto de coleções (*MongoDBCollection*) que são compostas de um esquema documento.

- Metamodelo de Mapeamento: este metamodelo foi simplificado para conter uma sequencia de regras (*MapRule*) interligando um elemento do modelo ER e uma coleção do MongoDB. Cada regra possui uma lista de regras que ligam um atributo do elemento ER a um atributo da coleção MongoDB.

A seguir apresenta-se os metamodelos para facilitar o entendimento das contribuições desta pesquisa.

4.1.1 Metamodelo ER

A Figura 24 representa o metamodelo ER. Este metamodelo contempla a existência de um Modelo ER (*ERModel*), que é composto por múltiplos *BaseERElements*. Um *BaseERElement* possui um nome, um apelido (*Alias*) e atributos (*Attributes*). *Entity* e *Relationship* são especializações de *BaseERElement*. Os atributos (*DataAttribute*) possuem nome (*Name)*, um valor indicando se é um identificador (*IsIdentifier)* e um valor indicando se é multivalorado (*IsMultiValued)*. O relacionamento (*Relationship*) contém pelo menos duas ocorrências de *RelationshipEnd*, que expressam o vínculo de uma entidade (*Entity*) com o relacionamento.

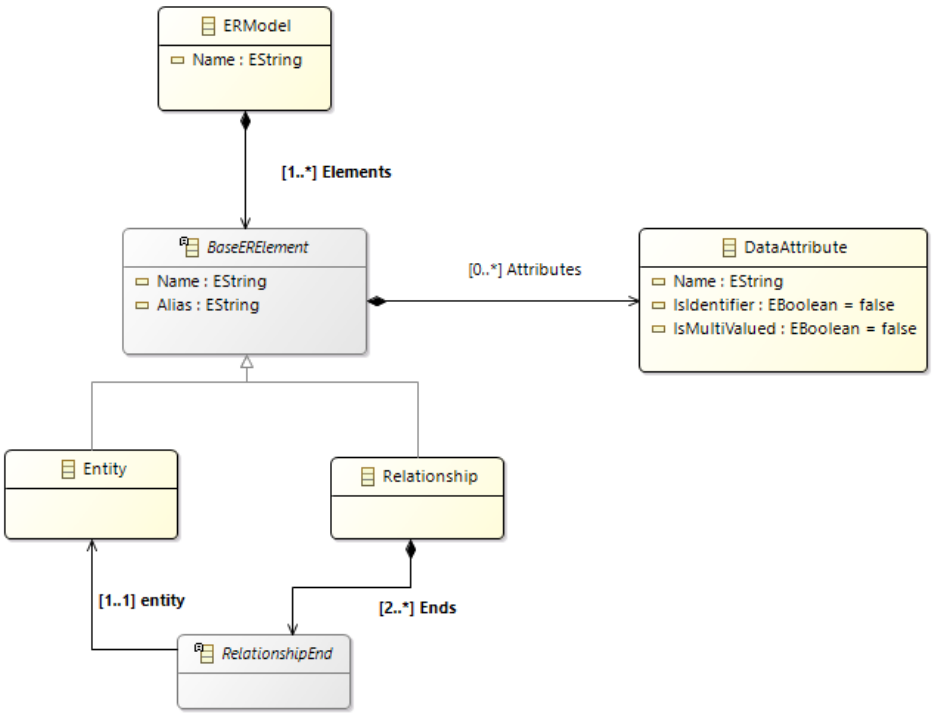

Figura 24 – Metamodelo ER

Um aspecto importante do metamodelo ER adotado é a ausência da informação de cardinalidade dos relacionamentos. Optou-se por adotar essa abordagem pois a álgebra de Parent e Spaccapietra (1984) também não contempla a cardinalidade em sua definição de modelo ER. Nesta abordagem, a relação de cardinalidade será contemplada no mapeamento, conforme discutido mais adiante neste capítulo.

4.1.2 Metamodelo Esquema MongoDB

Na Figura 25 está representado o metamodelo Esquema MongoDB. Neste metamodelo um Esquema MongoDB (*MongoSchema*) contém uma ou mais coleções (*MongoDBCollection*). Cada coleção é composta por um nome e um esquema de documento (*Document*). O documento é composto por um conjunto de atributos, onde cada atributo

tem um nome e indicativos se o mesmo é identificador (*IsIdentifier*) e se é multivalorado (*IsMultiValued*).

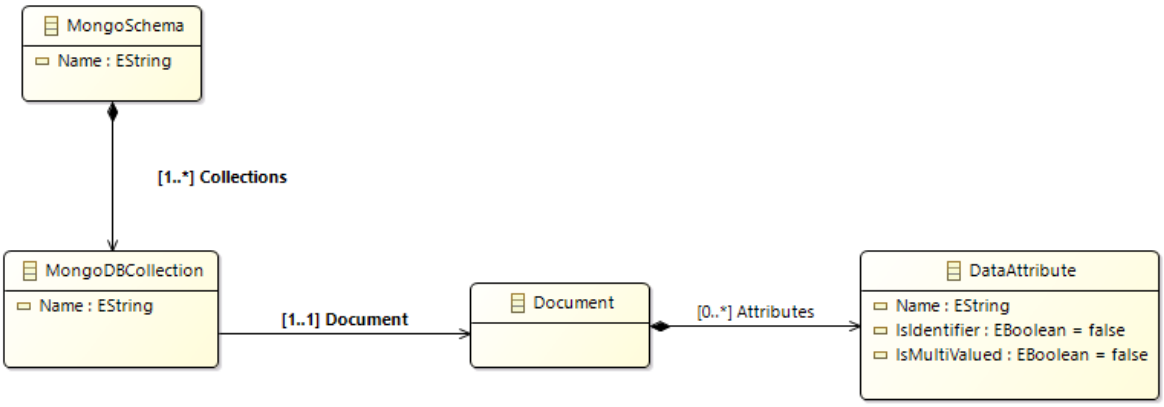

Figura 25 – Metamodelo Esquema MongoDB

4.1.3 Metamodelo de Mapeamento

A Figura 26 representa o metamodelo de mapeamento, que define como é feito o mapeamento entre um modelos ER e um esquema MongoDB. Um mapeamento (*Model-Mapping*) é composto por um conjunto de regras (*MapRule*). Cada regra cria um vínculo entre um elemento do modelo ER (*BaseERElement*) e uma coleção MongoDB (*MongoDB-Collection*), há também um atributo denominado *Rules* que define como cada atributo de um modelo é relacionado com um atributo do outro modelo, através de uma relação chave-valor (*Dictionary*).

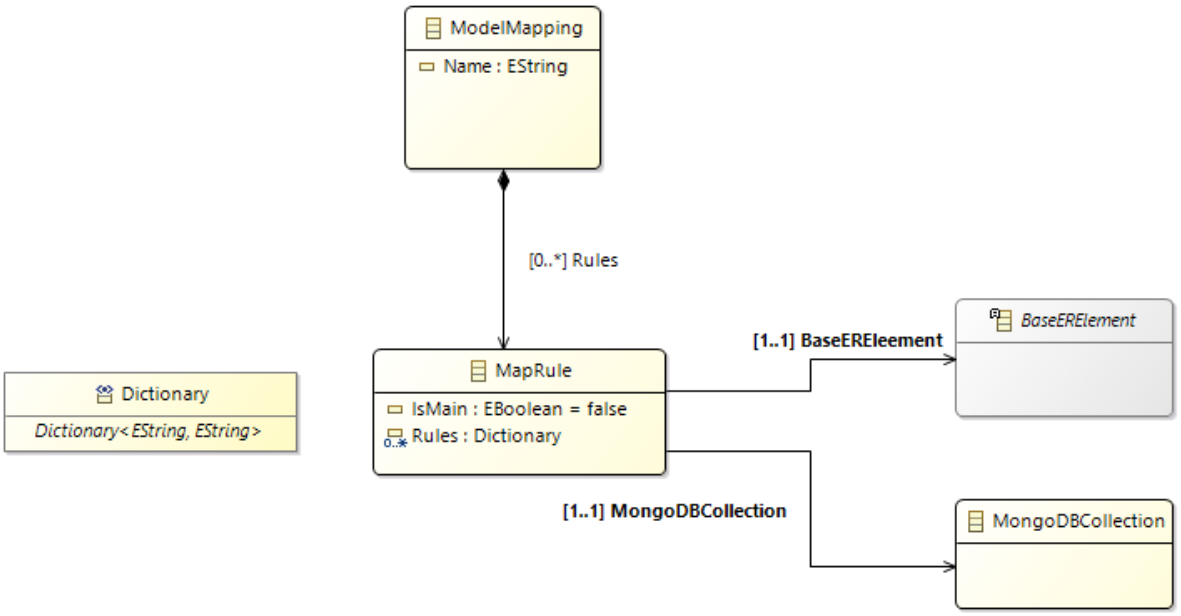

Figura 26 – Metamodelo Model-Mapping

Cada elemento do modelo ER deve ser mapeado, com a exceção de relacionamentos que não possuem atributos. Um mapeamento com a propriedade *IsMain* definida como verdadeiro indica que o elemento do modelo ER está mapeado para uma coleção dedicada a representar esse elemento. Em outras palavras, *IsMain* indica se aquela coleção é a fonte principal de informações referentes à entidade mapeada, e é ela quem deve ser usada em consultas que partem desta entidade. Coleções mapeadas com *IsMain* = falso indicam que as informações das entidades ali contidas são parciais, ou réplicas das informações principais, e devem ser utilizadas apenas em consultas interessadas nessas réplicas.

De acordo com Mason (2015) a modelagem de dados para um banco de dados orientado a documentos é diferente. Entidades separadas podem ser misturadas (desnormalizadas) em um único documento e o conceito de chave estrangeira é suportado através de uma referência. Para decidir se é necessário embutir ou manter a referência para outa entidade Hoberman (2014) faz a seguinte sugestão:

- Dados de múltiplas entidades que são frequentemente acessados ao mesmo tempo podem ser misturados (embutidos) em um único documento.
- Entidades que consideradas independentes podem ser embutidas em outra entidade.
- Se existe um relacionamento de cardinalidade 1:1 entre duas entidades, pode-se embutir um entidade na outra.
- Entidades que são inseridas, atualizadas e removidas com frequência similar podem ser embutidas.
- Entidades que não são entidade-chave, mas que possuem relacionamento com uma entidade-chave, podem ser referenciadas e não embutidas.

O mapeamento dos atributos se dá conforme as seguintes regras:

- Um atributo de uma Entidade/Relacionamento é mapeado a apenas um atributo da coleção.
- Atributos podem ser mapeados a atributos complexos, ou um documento embutido, seguindo a forma: *<raiz>.<atributo>*.

A Figura 27 demonstra de maneira simplificada o mapeamento entre uma entidade e uma coleção, e também as regras de mapeamento dos atributos. A estrutura *Address.Street* representa uma notação de hierarquia entre Pai e Filho, ou seja, o argumento à esquerda do ponto "." se refere a um atributo que contém o argumento à direita como filho.

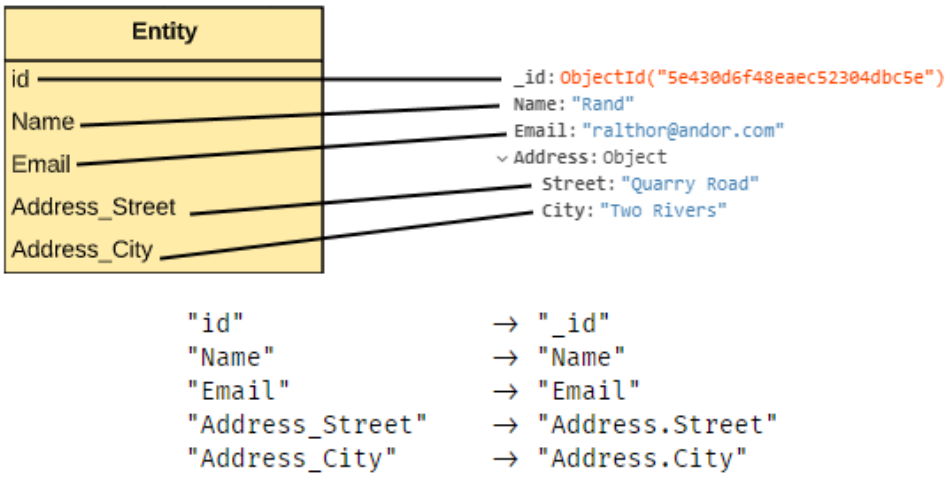

Figura 27 – Mapeamento entre Modelo ER e Esquema MongoDB

Expandindo o exemplo da Figura 27, consideram-se duas entidades *Person* e *Address*, e também um relacionamento *Lives*. A Figura 28 contém o mapeamento das duas entidades com o atributo "IsMain = true", ou seja, cada entidade foi mapeada para a sua própria coleção *PersonCollection* e *AddressCollection* respectivamente.

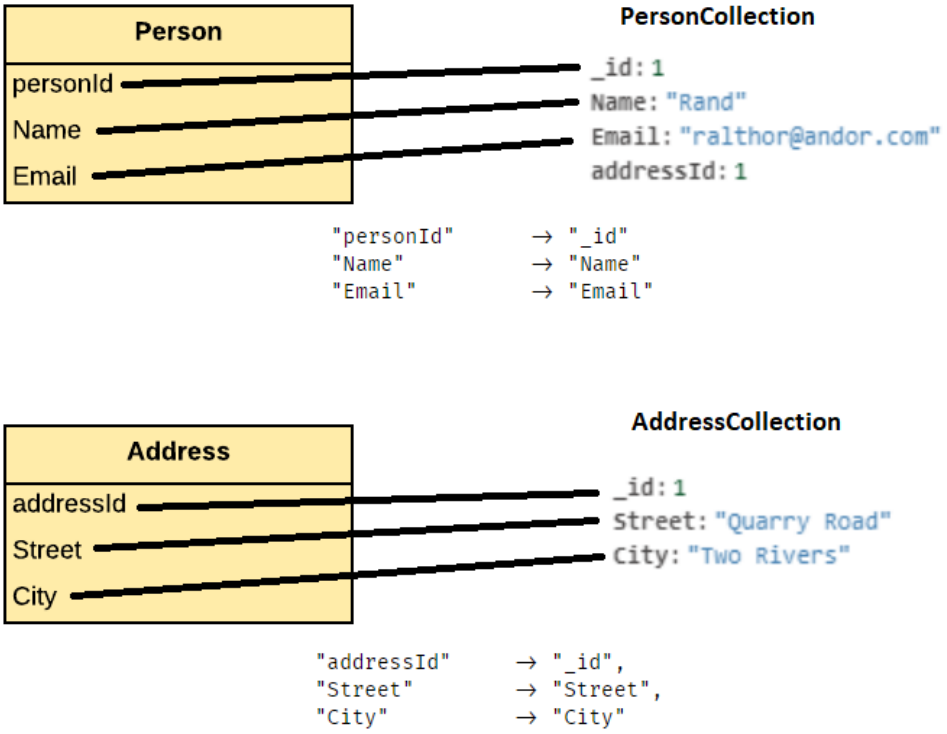

Figura 28 – Mapeamento "IsMain = true" para as entidades *Person* e *Address*

Nota-se que na Figura 28 existe um atributo na coleção *PersonCollection* chamado *addressId* que não foi mapeado por nenhuma entidade. Na Figura 29 esse atributo será mapeado pela entidade *Address*.

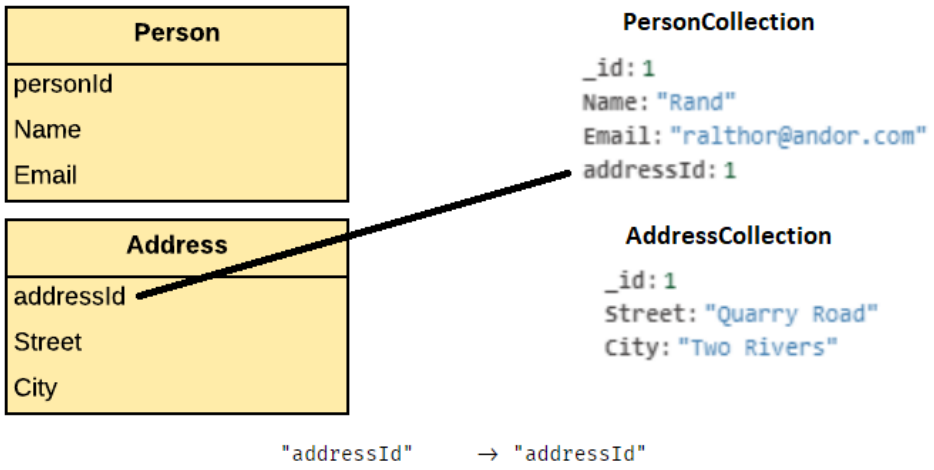

Figura 29 – Mapeamento "IsMain = false" para a entidade *Address*

Na Figura 29 está representado o mapeamento com atributo "IsMain = false" para a entidade *Address*, isto quer dizer que um ou mais atributos da entidade foram mapeados para uma coleção que não é destinada exclusivamente para tal entidade.

As entidades *Person* e *Address* estão relacionadas através do relacionamento *Lives* que não possui nenhum atributo. Mais uma vez, nota-se que não há qualquer menção à cardinalidade do relacionamento entre as entidades.

Observando as Figuras 28 e 29 pode-se dizer que a cardinalidade do relacionamento entre as entidades é 1:1, ou seja, uma ocorrência de *Person* está relacionada a apenas uma ocorrência de *Address*. Pode-se alterar o mapeamento para o exemplo da Figura 30, mapeando todos os atributos da entidade *Address* para a coleção *PersonCollection*.

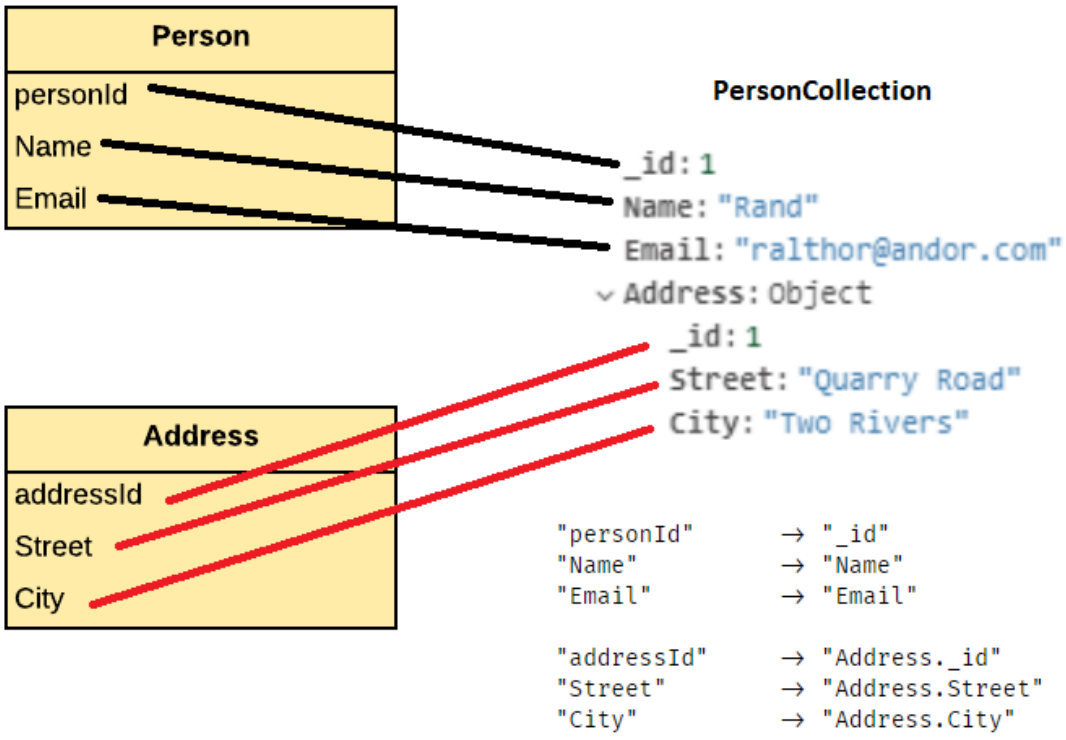

Figura 30 – Mapeamento "IsMain = true" para a entidade *Person* e "IsMain = false" para a entidade *Address*

O mapeamento da Figura 30 preserva as características do relacionamento *Lives*, ou seja, apesar do mapeamento ser diferente do citado anteriormente a cardinalidade do relacionamento é a mesma. Cita-se de forma insistente a cardinalidade para mostrar que assim como na álgebra o algoritmo não requer qualquer informação relativa à cardinalidade para gerar o código correspondente a junção de duas ou mais entidades.

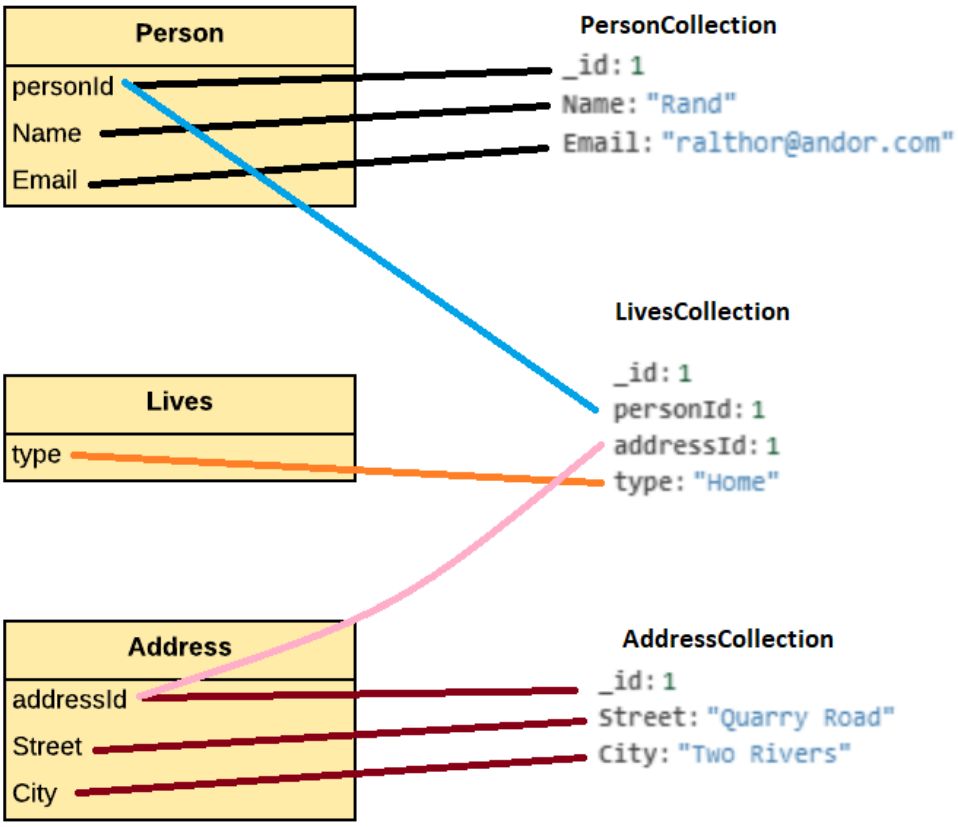

Figura 31 – Mapeamentos para as entidades *Person* e *Address* e o relacionamento *Lives*

Para demonstrar a flexibilidade da álgebra a Figura 31 ilustra novos mapeamentos para as entidades *Person* e *Address* e também o relacionamento *Lives* que para este exemplo possui um atributo chamado *type*. Nota-se que todos os elementos possuem um mapeamento "IsMain = true" (linhas com as cores preta [*Person -> PersonCollection*], laranja [*Lives -> LivesCollection*] e vermelho [*Address -> AddressCollection*]. As linhas rosa e azul representam mapeamentos "IsMain = false" para as entidades *Address* e *Person* respectivamente, ambos mapeamentos apontando para *LivesCollection*. No mapeamento da Figura 31 o algoritmo irá construir a junção entre as entidades *Person* e *Address* através da coleção intermediara para a qual o relacionamento *Lives* foi mapeado.

4.1.4 Representação textual dos metamodelos

Uma outra evolução em relação ao mapeamento desenvolvido por Noguera (2018) foi a criação de uma gramática livre de contexto para representar os três metamodelos de forma textual. Essa gramática[1] possibilitou a criação de um analisador sintático que automaticamente gera instâncias dos metamodelos para serem utilizadas na prática. Como exemplo, a Listagem 4.1 mostra o exemplo das Figuras 28 e 29 em forma textual, conforme a gramática desenvolvida. As linhas 5 a 19 mostram o modelo ER. As linhas 21 a 36

[1] Disponível em https://github.com/slowvoid/mongoQueryGenerator

mostram o esquema MongoDB já com o mapeamento indicado em cada nome de coleção e atributo. Por exemplo, a linha 28 indica que o atributo *addressId*, da entidade *Address* foi mapeado para a coleção *PersonCollection* que está vinculada a entidade *Person*.

```
1  Solution: "Person Sample"
2  Description: "Person Lives in Address (1:1)"
3  Version: "1.0"
4
5  ##############  ERModel  ################
6
7  Person {
8      personId: int
9      Name: string
10     Email: string
11 }
12
13 Address {
14     addressId: int
15     Street: string
16     City: string
17 }
18
19 Lives (Person, Address)
20
21 #############  MongoDBSchema  ##############
22
23 PersonCollection < Person*, Address >
24 {
25     _id: string     < Person.personId >
26     Name: string    < Person.Name >
27     Email:          < Person.Email >
28     addressId: int  < Address.addressId >
29 }
30
31 AddressCollection < Address* >
32 {
33     _id: string     < Address.addressId >
34     Street: string  < Address.Street >
35     City: string    < Address.City >
36 }
```

Listagem 4.1 – Representação textual dos três metamodelos utilizando a gramática desenvolvida

Outro exemplo da representação do mapeamento entre as entidades, utilizando como base a Figura 30, a Listagem 4.2 contém a representação textual dos três metamodelos descritos (de forma simplificada) na Figura 30. Na linha 23 foi adicionada referência à entidade *Address*, indicando que naquele bloco existe um mapeamento "IsMain = false" para a entidade. Nas linhas 29, 30 e 31 estão definidos os mapeamentos para os atributos *addressId*, *Street* e *City* da entidade *Address*. Esses atributos foram embutidos em um atributo chamado *address* como descrito na linha 28.

```
1  Solution: "Person Sample"
2  Description: "Person Lives in Address (1:1 embedded)"
3  Version: "1.0"
4
5  ##############  ERModel  ################
6
7  Person {
8      personId: int
9      Name: string
10     Email: string
11 }
12
13 Address {
14     addressId: int
15     Street: string
16     City: string
17 }
18
19 Lives (Person, Address)
20
21 #############  MongoDBSchema  ##############
22
23 PersonCollection < Person*, Address >
24 {
25     _id: string      < Person.personId >
26     Name: string     < Person.Name >
27     Email:           < Person.Email >
28     address: {
29         addressId: int   < Address.addressId >
30         Street: string   < Address.Street >
31         City: string     < Address.City >
32     }
```

```
33 }
```

Listagem 4.2 – Representação textual dos três metamodelos utilizando a grámatica desenvolvida

Nos exemplos das Listagens 4.1 e 4.2 a cardinalidade do relacionamento *Lives* entre as entidades *Person* e *Address* é "um-para-um" (1:1). Na Figura 32 está representando um mapeamento entre das entidades *Person* e *Address*, neste caso a cardinalidade do relacionamento entre as entidades é "um-para-muitos" (1:N).

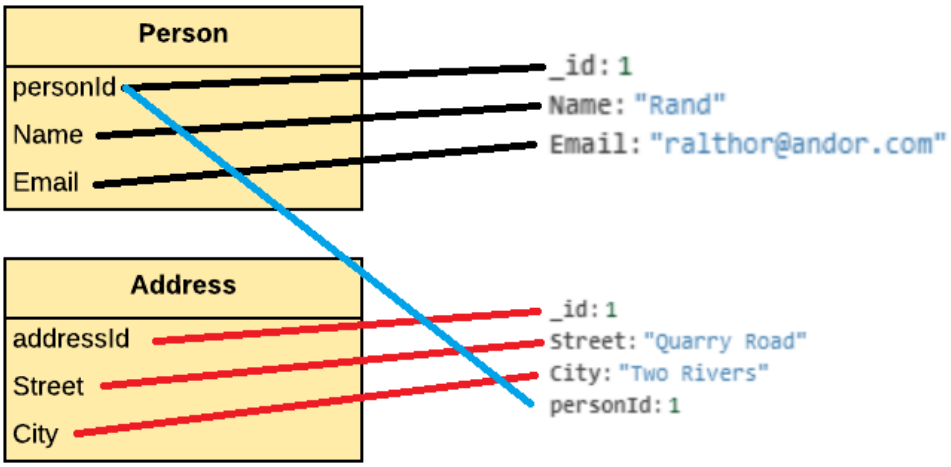

Figura 32 – Mapeamentos para as entidades *Person* e *Address*

Também pode-se representar a cardinalidade "um-para-muitos" (1:N) incorporando todos os atributos da entidade *Address* na coleção destinada à entidade *Person* utilizando um atributo multivalorado conforme demonstra a Figura 33.

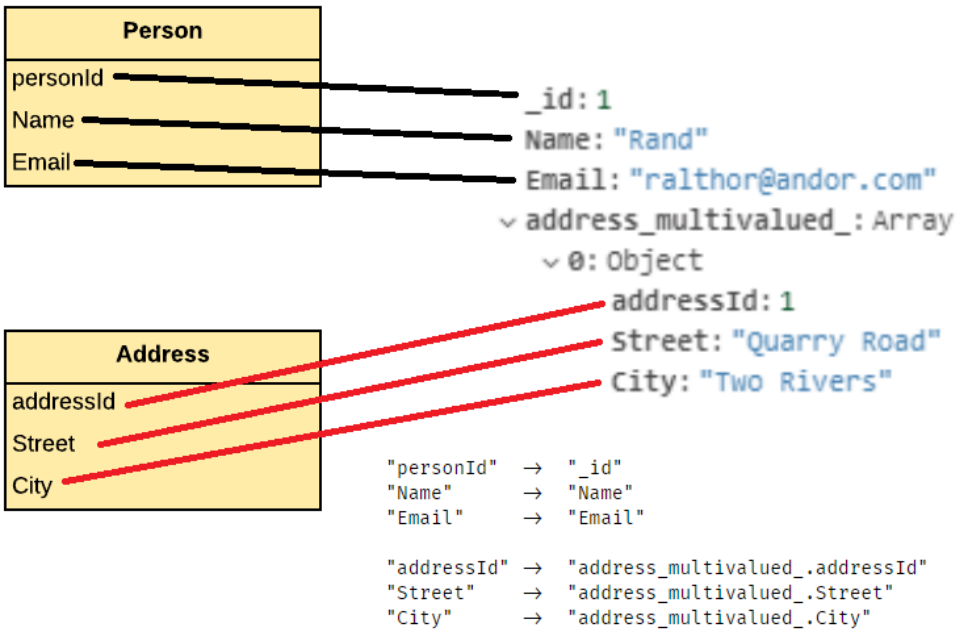

Figura 33 – Mapeamentos para as entidades *Person* e *Address*

A Listagem 4.3 descreve como é representado textualmente o mapeamento representado pela Figura 32. A linha 31 descreve que a coleção *AddressCollection* está vinculada à entidade *Address* como mapeamento "IsMain = true" (representado pelo asterisco) e que contém atributo(s) da entidade *Person* mapeada a ela. Na linha 35 nota-se que foi definido um mapeamento do atributo *personId* pertencente a entidade *Person*.

```
1  Solution: "Person Sample"
2  Description: "Person Lives in Address (1:N)"
3  Version: "1.0"
4
5  ##############  ERModel  ################
6
7  Person {
8      personId: int
9      Name: string
10     Email: string
11 }
12
13 Address {
14     addressId: int
15     Street: string
16     City: string
17 }
18
19 Lives (Person, Address)
```

```
20
21 ##############  MongoDBSchema  ###############
22
23 PersonCollection < Person* >
24 {
25     _id: string      < Person.personId >
26     Name: string     < Person.Name >
27     Email:           < Person.Email >
28 }
29
30 AddressCollection < Address*, Person >
31 {
32     _id: string      < Address.addressId >
33     Street: string   < Address.Street >
34     City: string     < Address.City >
35     personId: int    < Person.personId >
36 }
```

Listagem 4.3 – Representação textual dos mapeamentos da Figura 32

Na Listagem 4.4 está descrita representação textual do mapeamento da Figura 33. Nota-se que na linha 23 foi adicionada referência à entidade *Address* indicando que a coleção *PersonCollection* participa de uma mapeamento "IsMain = false" para a entidade *Address*. Nas linhas 29, 30 e 31 está definido o mapeamento dos atributos da entidade *Address*. É importante notar que os atributos estão vinculados a um outro atributo com o sufixo *__multivalued__* (linha 28) que indica ao algoritmo que este atributo é composto de diversas ocorrências de uma mesma estrutura (os atributos das linhas 29, 30 e 31).

```
1  Solution: "Person Sample"
2  Description: "Person Lives in Address (1:N embedded)"
3  Version: "1.0"
4
5  ##############  ERModel  ################
6
7  Person {
8      personId: int
9      Name: string
10     Email: string
11 }
12
13 Address {
14     addressId: int
15     Street: string
```

```
16     City: string
17 }
18
19 Lives (Person, Address)
20
21 #############  MongoDBSchema  ##############
22
23 PersonCollection < Person*, Address >
24 {
25     _id: string      < Person.personId >
26     Name: string     < Person.Name >
27     Email:           < Person.Email >
28     address_multivalued_: [
29         addressId: int   < Address.addressId >
30         Street: string   < Address.Street >
31         City: string     < Address.City >
32     ]
33 }
```

Listagem 4.4 – Representação textual dos mapeamentos da Figura 33

O mapeamento descrito pela Figura 31 mostra que nesse exemplo a cardinalidade do relacionamento entre as entidades *Person* e *Address* é "muitos-para-muitos" (N:M). A relação é feita através de uma coleção intermediária chamada *LivesCollection*. A Listagem 4.5 descreve a representação textual desse mapeamento. As linhas 38 até 44 contém a representação do mapeamento entre o relacionamento *Lives* e a coleção *LivesCollection*. Nota-se na linha 38 referências às entidades *Person* e *Address* significando que a coleção *LivesCollection* contém atributo(s) mapeado(s) dessas entidades. Nas linhas 41 e 42 estão mapeados os atributos *personId* e *addressId*.

```
1 Solution: "Person Sample"
2 Description: "Person Lives in Address (N:M muitos-para-muitos)"
3 Version: "1.0"
4
5 ##############  ERModel  ################
6
7 Person {
8     personId: int
9     Name: string
10    Email: string
11 }
12
13 Address {
```

```
    addressId: int
    Street: string
    City: string
}

Lives (Person, Address) {
    type: string
}
#############  MongoDBSchema  ##############

PersonCollection < Person* >
{
    _id: string     < Person.personId >
    Name: string    < Person.Name >
    Email:          < Person.Email >
}

AddressCollection < Address* >
{
    _id: string     < Address.addressId >
    Street: string  < Address.Street >
    City: string    < Address.City >
}

LivesCollection < Lives*, Person, Address >
{
    _id: int        < >
    personId: int   < Person.personId >
    addressId: int  < Address.addressId >
    type: string    < Lives.type >
}
```

Listagem 4.5 – Representação textual dos mapeamentos da Figura 31

O analisador sintático processa automaticamente esse arquivo textual e produz código que instancia objetos referentes aos elementos do modelo ER, do esquema MongoDB e do mapeamento. Esses objetos servirão de entrada para o algoritmo de geração de consultas, descrito a seguir.

4.2 Algoritmo para geração de consultas

Inicialmente pretendia-se aprimorar o algoritmo desenvolvido por Noguera (2018) que foi criado na linguagem Java, mas por preferência e maior afinidade com a linguagem C# optou-se então por reescrever os modelos e algoritmo na nova linguagem. Este processo de reescrita também ajudou a melhorar a estrutura e modularização do código-fonte original.

O objetivo do algoritmo é interpretar uma consulta escrita na linguagem proposta por Noguera (2018) e produzir uma consulta ao banco de dados MongoDB utilizando um modelo ER, um Esquema MongoDB e um Mapeamento ER-MongoDB como parâmetros de entrada.

É importante afirmar que o algoritmo não interpreta diretamente a linguagem de consulta, ela é utilizada como referência para a construção das instruções do algoritmo. Ou seja, na implementação atual, para especificar uma consulta, o desenvolvedor precisa especificar suas instruções em forma de código-fonte escrito em C#. Foi dado início ao desenvolvimento de um analisador sintático automático com base em uma gramática livre de contexto[2], mas devido a limitações de tempo, no momento da escrita desta dissertação a implementação ainda não estava concluída.

O algoritmo é composto por um conjunto de operações que representam os operadores propostos pela álgebra de Parent e Spaccapietra (1984). O trabalho de Parent e Spaccapietra (1984) propõe os seguintes operadores:

- Junção de relacionamento (Relationship Join)
- Produto cartesiano
- Projeção
- Seleção
- Redução
- Compressão
- União, Diferença

Devido à complexidade das operações, especialmente a junção de relacionamento, não foram implementados as operações de *Redução*, *Compressão* e *União, Diferença*. Foram implementadas apenas as operações *Junção de relacionamento (Relationship Join)*, *Produto cartesiano*, *Projeção* e *Seleção*.

[2] Disponível em `https://github.com/slowvoid/mongoQueryGenerator`

Para facilitar a leitura os modelos serão representados de forma simplificada e textual. A Listagem 4.6 representa um modelo Entidade-Relacionamento.

```
1  Person {
2      id: string
3      name: string
4      surname: string
5      salary: double
6  }
7  Car {
8      id: string
9      reg_no: int
10 }
11 InsuranceCompany {
12     id: string
13     name: string
14 }
15 Garage {
16     id: string
17     name: string
18     phone: string
19 }
20 Insurance (Person, Car, InsuranceCompany) {
21     contract: date
22 }
23 Drives (Person, Car) { }
24 Repaired (Car, Garage) {
25     date: date
26 }
```

Listagem 4.6 – Representação simplificada do Modelo ER

Na Listagem 4.6 as linhas de 1 a 6 descrevem uma entidade chamada *Person* e as linhas de 2 a 5 seus atributos. As linhas de 7 a 10, 11 a 14 e 15 a 19 representam as entidades *Car*, *InsuranceCompany* e *Garage* respectivamente. As linhas 20 a 22, 23 e 24 a 26 representam os relacionamentos *Insurance*, *Drives* e *Repaired*. É importante notar que nas linhas 20, 23 e 24, os nomes entre parênteses determinam quais entidades estão relacionadas através do relacionamento nomeado na mesma linha.

Aqui retoma-se a discussão sobre a cardinalidade. Como mostra a Listagem 4.6, na definição dos relacionamentos não é especificada a cardinalidade. O algoritmo utiliza os mapeamentos para construir o código nativo para MongoDB para as operações. Em particular, para o algoritmo, basta saber qual é a natureza mono/multivalorada dos atributos

para gerar o código corretamente.

Para ilustrar esse comportamento a Listagem 4.7 contém um fragmento do código do algoritmo para a construção da operação de junção.

```
1  if ( !Relationship.AreRelated( (Entity)SourceEntity.Element, (
       Entity)Target.Element ) )
2  {
3      ...
4  }
5
6  MapRule MainTargetRule = ModelMap.FindMainRule( Target.Element );
7
8  if ( MainTargetRule == null )
9  {
10     MapRule TargetRelationshipRule = ModelMap.FindRule( Target.
       Element, MainRelationshipRule.Target );
11
12     if ( TargetRelationshipRule == null )
13     {
14         ...
15     }
16
17     ...
18
19     foreach ( DataAttribute Attribute in Target.Element.
       Attributes )
20     {
21         ...
22     }
23 }
```

Listagem 4.7 – Fragmento de código do algoritmo para construção de uma operação *Junção de Relacionamento* (*Relationship Join*)

O fragmento de código da Listagem 4.7 descreve como o algoritmo constrói a operação de junção. Na linha 1 é verificado se as entidades são relacionadas através do relacionamento utilizado na operação. Depois é verificado a existência de um mapeamento "IsMain = true" para a entidade alvo da junção[3]. Caso não seja encontrado tal mapeamento (linha 8) o algoritmo busca por um mapeamento "IsMain = false" para a entidade alvo (linha 10) e então o algoritmo prossegue com o processamento de atributos da entidade alvo.

[3] Entidade alvo refere-se à entidade à direita da consulta. No exemplo de consulta [*FROM Person RJOIN <R> Address*], *Address* é a entidade alvo

```
1  MapRule MainTargetRule = ModelMap.FindMainRule( Target.Element );
2
3  if ( MainTargetRule == null )
4  {
5      MapRule TargetRule = ModelMap.FindRule( Target.Element,
       MainSourceRule.Target );
6
7      if ( TargetRule == null )
8      {
9          ...
10     }
11
12     bool IsRootMultivalued = TargetRule.
       BelongsToMultivaluedAttribute();
13 }
```

Listagem 4.8 – Fragmento de código do algoritmo para construção de uma operação *Junção de Relacionamento* (*Relationship Join*)

Uma outra possibilidade que é verificada pelo algoritmo, descrito pela Listagem 4.8 é verificar se a entidade alvo possui mapeamento "IsMain = false" (linha 5) e seus atributos foram mapeados para um atributo multivalorado (entidade incorporada) como descrito na linha 12. Nota-se com esses exemplos que, assim como a álgebra, o algoritmo não utiliza qualquer referência à cardinalidade para gerar código de uma junção entre entidades.

A Listagem 4.9 representa a definição do Esquema MongoDB e do Mapeamento entre o Modelo ER e o Esquema definido. As linhas de 1 a 7 indicam o nome da coleção e seus atributos. Na linha 1 é definido o nome da coleção *PersonCollection* e o elemento ER que está mapeado a ela (*Person*). O "*" indica que este mapeamento possui a propriedade *IsMain* como verdadeiro. As linhas de 3 a 6 demonstram os atributos da coleção e o seu tipo de dado e qual atributo do elemento ER é mapeado a ele (valor entre os sinais de menor e maior). Nas linhas de 8 a 12, 13 a 17 e 19 a 23 são definidas as coleções e mapeamentos para as coleções *CarCollection*, *InsuranceCompanyCollection* e *GarageCollection* respectivamente.

Nas linhas 24 a 31 da Listagem 4.9 é definida a coleção *InsuranceCollection*. Para essa coleção estão mapeados os atributos do relacionamento *Insurance* (definindo *IsMain=true*) e um atributo identificado das entidades *Person*, *Car* e *InsuranceCompany*. Nota-se que não existem atributos mapeados para "_id" nos relacionamentos. Isso é intencional: o campo "_id" é gerado automaticamente pelo MongoDB (obrigatório) e para este exemplo o valor representado não é relevante.

As linhas 32 a 37 e 38 a 44 da Listagem 4.9 representam o esquema e o mapeamento para os relacionamentos *Drives* e *Repaired*.

```
1  PersonCollection < Person* >
2  {
3      _id      : string    < Person.id >
4      fName    : string    < Person.name >
5      fSurname: string     < Person.surname >
6      fSalary  : double    < Person.salary >
7  }
8  CarCollection < Car* >
9  {
10     _id      : string    < Car.id >
11     fReg_no  : int       < Car.reg_no >
12 }
13 InsuranceCompanyCollection < InsuranceCompany* >
14 {
15     _id      : string    < InsuranceCompany.id >
16     fName    : string    < InsuranceCompany.name >
17 }
18 GarageCollection < Garage* >
19 {
20     _id      : string    < Garage.id >
21     fName    : string    < Garage.name >
22     fPhone   : string    < Garage.phone >
23 }
24 InsuranceCollection < Insurance*, Person, Car, InsuranceCompany >
25 {
26     _id          : string    <>
27     fPersonId    : string    < Person.id >
28     fCarId       : string    < Car.id >
29     fInsCoId     : string    < InsuranceCompany.id >
30     contract     : date      < Insurance.contract >
31 }
32 DrivesCollection < Drives*, Person, Car >
33 {
34     _id          : string    <>
35     fPersonId    : string    < Person.id >
36     fCarId       : string    < Car.id >
37 }
38 RepairedCollection < Repaired*, Car, Garage >
39 {
```

```
40      _id          : string     <>
41      fCarId       : string     < Car.id >
42      fGarageId    : string     < Garage.id >
43      fDate        : date       < Repaired.date >
44  }
```

Listagem 4.9 – Definição do Esquema MongoDB e Mapeamento com o Modelo ER

No exemplo da Listagem 4.9 nota-se que a informação referente à cardinalidade dos relacionamentos acaba se manifestando. Nos três relacionamentos deste exemplo, optou-se por criar uma coleção intermediária para armazenar as ocorrências dos relacionamentos. Isto implica que é possível relacionar as entidades seguindo o modelo "muitos-para-muitos" sem causar duplicação das informações além dos atributos identificadores. Outros exemplos de mapeamento poderiam restringir essa cardinalidade. Caso os dados de *Car* estivessem embutidos dentro da coleção da entidade *Person* como um atributo monovalorado, por exemplo, isso implicaria um relacionamento do tipo "muitos-para-um" (uma pessoa dirige um único carro, mas cada carro pode ser dirigido por múltiplas pessoas).

Como o algoritmo de geração das consultas não leva em consideração as cardinalidades, apenas o mapeamento, isso significa que fica a cargo do projetista do banco de dados definir um mapeamento que atenda às restrições necessárias para os dados. Com a possibilidade de criar coleções embutidas, atributos multivalorados e desnormalização dos dados no MongoDB, a gama de possibilidades é bastante extensa.

Um mapeamento pode gerar mais de um código para MongoDB para uma mesma consulta. Isto pode ocorre quando uma ou mais entidades possuem mapeamento com o atributo "IsMain = true" e também foram incorporadas a uma outra entidade. Em outras palavras, existem diferentes "caminhos" para se escrever uma mesma consulta. O trabalho de Noguera (2018) considera todas as possíveis consultas para um mesmo mapeamento e gera um código para MongoDB equivalente para cada possibilidade, deixando para o desenvolvedor a tarefa de escolher qual consulta gerada utilizar. Ou seja, a relação entre quantidade de mapeamentos para a entidade (qME) e quantidade de consultas geradas (qC) pode ser expressa da seguinte maneira:

$$qC = qME_1 \times qME_2 \times ... \times qME_n$$

Neste trabalho decidiu-se que na presença de diferentes possibilidades para um mesmo mapeamento o algoritmo irá gerar apenas um código para MongoDB para cada mapeamento, tendo como preferência mapeamentos com o atributo "IsMain = true". O motivo de limitar o algoritmo a apenas um resultado por mapeamento deve-se ao grande número de combinações que isso poderia gerar, o que estava dificultando a implementação. Portanto, neste momento, optou-se por simplificar neste sentido.

Em um ponto importante a considerar, o trabalho de Noguera (2018) utiliza a operação "find" em todas as consultas geradas. Essa operação é indicada para consultas simples e seu uso para executar operações mais complexas como junções compostas não aproveita da capacidade do MongoDB em tentar otimizar o pipeline de uma junção (MONGODB, 2019). Por isso, para este trabalho optou-se por adotar o "aggregate" para todas as consultas geradas. Além disso, o "aggregate" permite a execução de uma lista de operações onde a saída de cada operação é utilizada como entrada para a operação seguinte e esse fato contribuiu para a decisão de utilizá-lo, pois essa característica é muito similar à proposta de álgebra que diz que a saída de cada operação deve ser compatível com o padrão de entrada para as outras operações.

4.2.1 Estrutura do resultado da consulta

Parent e Spaccapietra (1984) definem em seu trabalho que as operações *Junção*, *Produto Cartesiano* e *Projeção* são as únicas operações que alteram a estrutura do resultado da consulta. As operações *Junção* e *Produto Cartesiano* são semelhantes na forma como tratam o resultado final com a exceção dos dados do relacionamento que não faz parte do Produto Cartesiano.

Foi decidido por simplificar a estrutura de dados proposta por Parent e Spaccapietra (1984) para que ao invés de possuir um atributo agrupando todos os atributos da entidade de origem e um outro atributo multivalorado contendo todos os atributos do relacionamento e da entidade "alvo", os atributos da entidade origem serão preservados em primeiro nível e os atributos do relacionamento e da entidade "alvo" serão agrupados em um atributo multivalorado. O motivo dessa mudança é o fato de que é a estrutura final do documento será mais simples e a relação dos dados (quando houver uma junção por exemplo) é fácil de ser notada.

Na Listagem 4.10, o resultado de uma consulta contém todos os atributos da entidade de origem no mesmo nível (linha 2) e se estiver presente uma operação *Junção* ou *Produto Cartesiano* haverá um novo atributo chamado *data_Relacionamento_ou_Entidade* (linha 3) multivalorado e conterá os atributos do relacionamento (linhas 5 e 9) e da entidade alvo (linhas 6 e 10). A fim de evitar a sobreposição de atributos de mesmo nome, em uma operação de *Junção* os atributos do relacionamento e da entidade alvo são prefixados com o nome de seu elemento ER de origem (entidade ou relacionamento).

```
1 {
2     atributoEntidadeOrigem: 'valor',
3     data_Relacionamento_ou_Entidade: [
4         {
5             Relacionamento_atributoRelacionamento: 'valor_1',
6             EntidadeAlvo_atributoEntidadeAlvo_Ocorrencia1: 1
```

```
7           },
8           {
9               Relacionamento_atributoRelacionamento: 'valor_2',
10              EntidadeAlvo_atributoEntidadeAlvo_Ocorrencia2: 2
11          }
12      ]
13  }
```

Listagem 4.10 – Estrutura do documento resultante das consultas. Adaptado da proposta de Parent e Spaccapietra (1984).

4.2.2 Operação Junção (*Relationship Join*)

Como mencionado anteriormente, no estado atual da implementação, uma consulta deve ser especificada em código-fonte C#[4], como um conjunto de operações correspondendo àquelas propostas por Parent e Spaccapietra (1984). A Listagem 4.11 representa os argumentos da operação de junção entre entidades. A linha 1 é a inicialização de uma nova instância da operação *Junção*. As linhas 2 a 5 são os argumentos da operação, que são respectivamente uma Entidade de origem (*Person*), o Relacionamento (*Drives*), uma lista com as entidades a serem juntadas, neste exemplo o relacionamento apenas relaciona *Person* e *Car*, e finalmente o mapeamento entre o Modelo ER e Esquema MongoDB (*Map*). Neste código, os objetos *Person*, *Drives*, *Car* e *Map* são gerados automaticamente a partir do modelo textual descrito na Seção 4.1.4.

```
1 RelationshipJoinOperator RJoinDrives = new
     RelationshipJoinOperator(
2     Person,
3     Drives,
4     new List<QueryableEntity>() { Car },
5     Map );
```

Listagem 4.11 – Código C# representando uma chamada da operação *Junção*.

Antes de entrar em detalhes sobre essa operação, se faz necessário introduzir uma composição de entidades chamada *Computed Entity* (Entidade Computada) (PARENT; SPACCAPIETRA, 1984). Uma entidade computada é a composição de uma entidade com outras entidades através de um relacionamento, pode-se considerar essa estrutura como uma representação prévia da operação *Junção* e é aceita como valor de entrada para a mesma, permitindo então a criação de operações de junção (*Junção*) aninhadas, como descrita na Listagem 4.12.

[4] Como já discutido anteriormente, atualmente este código precisa ser escrito à mão, mas a ideia é que no futuro ele seja automaticamente gerado a partir de um analisador sintático. Devido às limitações de tempo, esta etapa não foi concluída a tempo para a escrita desta dissertação.

```
1     FROM Person
2     RJOIN <Drives> ( Car RJOIN <Repaired> ( Garage ) )
3     SELECT *
```

Listagem 4.12 – Exemplo de consulta com junções aninhadas

Na Listagem 4.12, a linha 2 indica que serão realizadas duas operações de junção, uma que reúne a entidade de origem definida na linha 1 (*FROM Person*) e uma outra operação reunindo as entidades *Car* e *Garage* através do relacionamento *Repaired*. Essa junção é representada no algoritmo como uma *Computed Entity*, que a fim de simplificar a escrita, será chamada de *CarGarage*. A Listagem 4.13 mostra como a consulta definida na Listagem 4.12 é interpretada pelo algoritmo.

```
1     FROM Person
2     RJOIN <Drives> ( CarGarage )
3     SELECT *
```

Listagem 4.13 – Exemplo de consulta indicando uma Entidade Computada como argumento

Nota-se que na linha 2 da Listagem 4.13 a entidade a ser juntada não é mais escrita como uma operação, e sim como uma composição de entidades.

O primeiro passo do algoritmo é localizar o mapeamento da entidade de origem (*Person*) e o mapeamento do relacionamento (*Drives*). Nas Listagens 4.6 e 4.9 nota-se que o relacionamento *Drives* não possui atributos mas relaciona as entidades *Person* e *Car* em uma coleção própria, e por isso possui um mapeamento definido.

Uma vez identificada a presença ou não de um mapeamento explícito para o relacionamento, o algoritmo irá analisar cada entidade presente na lista informada na entrada da operação *Junção*. Então o algoritmo identifica se existe uma relação entre a entidade de origem e a entidade a ser juntada através do relacionamento e depois identifica o mapeamento da entidade alvo e então cria os operadores necessários para concretizar a junção.

A operação ao ser processada resulta em um conjunto de operadores (*MongoDBOperator*). Uma única operação *Junção* pode resultar em dois ou mais operadores, dependendo do mapeamento das entidades.

Alguns operadores, em especial o *LookupOperator*, podem ser compostos por múltiplos operadores. A Listagem 4.14 contém os operadores gerados pelo algoritmo como resultado da consulta da Listagem 4.13.

```
1 LookupOperator -> [
2     MatchOperator
3     LookupOperator
```

```
4     UnwindOperator
5     AddFieldsOperator
6     ProjectOperator
7 ]
```

Listagem 4.14 – Lista de operadores que são o resultado da operação *Junção* definida na Listagem 4.13.

Na Listagem 4.14, a linha 1 indica que o operador *LookupOperator* é composto dos operadores denominados nas linhas 2 a 6. A função de cada operador no caso do *LookupOperator* é buscar informações de outra coleção e possui suporte para execução de uma sequência de operadores junto à coleção informada antes de retornar os documentos, ou seja, permite a execução de uma consulta secundária. O operador *MatchOperator* é a base para uma comparação lógica. Neste caso é definida aqui a condição para relacionar a entidade de origem e a coleção para a qual o relacionamento está mapeado. O próximo *LookupOperator* fará a junção entre o registro do relacionamento e um registro da entidade alvo. O operador *AddFieldsOperator* adiciona novos atributos ao documento a ser retornado pelo banco de dados e o operador *ProjectOperator* executa uma operação de projeção, podendo adicionar/remover atributos ou atribuir um valor diferente aos atributos existentes.

A Listagem 4.15 é a consulta JavaScript para MongoDB gerada pelo algoritmo a partir dos modelos ER, Esquema MongoDB e Mapeamento. O MongoDB possui uma funcionalidade chamada *Aggregation Pipeline* que é uma sequência de operações executadas sobre uma coleção. Na Listagem 4.14 foram exibidos de forma simplifica os operadores, na Listagem 4.15 (linhas 2 até 38) está definido o operador *Lookup*, que por sua vez é composto por um conjunto de operadores (linhas 9 a 37). Os operadores citados na Listagem 4.14 como aninhados ao *LookupOperador* estão definidos nas linhas 9 a 13 (*MatchOperator*), 15 a 20 (*LookupOperator*), 22 a 25 (*UnwindOperator*), 27 a 30 (*AddFieldsOperator*) e 32 a 36 (*ProjectOperator*).

```
1 db.PersonCollection.aggregate([{
2     $lookup: {
3         from: "DrivesCollection",
4         as: "data_Drives",
5         let: {
6             match_Person: "$_id"
7         },
8         pipeline: [{
9             $match: {
10                $expr: {
11                    $eq: ["$fPersonId", "$$match_Person"]
12                }
```

```
            }
        }, {
            $lookup: {
                from: "CarCollection",
                as: "data_Car",
                foreignField: "_id",
                localField: "fCarId"
            }
        }, {
            $unwind: {
                path: "$data_Car",
                preserveNullAndEmptyArrays: true
            }
        }, {
            $addFields: {
                Car_id: "$data_Car._id",
                Car_reg_no: "$data_Car.fReg_no"
            }
        }, {
            $project: {
                data_Car: false,
                fPersonId: false,
                fCarId: false
            }
        }]
    }
}])
```

Listagem 4.15 – Consulta JavaScript para MongoDB gerada pelo algoritmo a partir da consulta definida na Listagem 4.13

4.2.3 Produto Cartesiano

A operação *Produto Cartesiano* une a entidade origem a todas as ocorrências da entidade alvo. É uma operação similar a *Junção* excluindo a necessidade das entidades estarem relacionadas. A Listagem 4.17 mostra a consulta gerada pelo algoritmo a partir da consulta definida na Listagem 4.16[5].

```
FROM Person
CARTESIANPRODUCT Garage
SELECT *
```

Listagem 4.16 – Exemplo de consulta utilizando a operação *Produto Cartesiano*

[5] Neste exemplo e nos próximos, omite-se o código C# referente às consultas, visando facilitar a leitura.

Na Listagem 4.16 a linha 2 define a operação utilizando como argumento uma entidade qualquer. Como dito anteriormente, para a operação de *Produto Cartesiano* não é necessário qualquer tipo de relação entre as entidades.

```
1 db.PersonCollection.aggregate([{
2     $lookup: {
3         from: "GarageCollection",
4         as: "data_Garage",
5         let: {},
6         pipeline: []
7     }
8 }])
```

Listagem 4.17 – Consulta JavaScript para MongoDB gerada pelo algoritmo a partir da consulta definida na Listagem 4.16

A Listagem 4.17 é a consulta JavaScript para MongoDB gerada pelo algoritmo a partir da consulta definida na Listagem 4.16. Nota-se que é utilizado apenas um operador *Lookup* (linhas 2 a 7). Na linha 6 o atributo *pipeline* é definido como um conjunto vazio, isso indica ao MongoDB que ao acessar a coleção especificada em *from* (linha 3) não execute nenhuma ação, apenas retorne os documentos contidos na coleção.

4.2.4 Projeção

A operação de *Projeção* tem como objetivo controlar quais atributos serão exibidos. Na Listagem 4.18 está descrita uma consulta utilizando projeção. Na linha 3 são especificados quais atributos serão mantidos no resultado da consulta.

```
1 FROM Person
2 RJOIN <Drives> ( Car )
3 SELECT Person.name, Person.surname, Car.reg_no
```

Listagem 4.18 – Exemplo de consulta utilizando a operação de *Projeção*

Na Listagem 4.18 a operação de projeção definida na linha 3 segue o padrão *SELECT <Entidade ou Alias>.<Atributo>*. Neste caso apenas os atributos *name* que pertence à entidade *Person*, *surname* que também pertence à entidade *Person* e *reg_no* que pertence à entidade *Car* serão incluídos no resultado.

```
1 db.PersonCollection.aggregate([{
2     $lookup: {
3         from: "DrivesCollection",
4         as: "data_Drives",
5         let: {
6             match_Person: "$_id"
```

```
7           },
8           pipeline: [
9               ...
10          ]
11      }
12  }, {
13      $project: {
14          "fName": true,
15          "fSurname": true,
16          "data_Drives.Car_reg_no": true
17      }
18  }])
```

Listagem 4.19 – Consulta JavaScript para MongoDB gerada pelo algoritmo a partir da consulta definida na Listagem 4.18

A Listagem 4.19 é o código JavaScript gerado pelo algoritmo a partir da consulta definida pela Listagem 4.18. Para simplificar a compreensão do código, foram omitidos os códigos dos operadores vinculados ao operador *Lookup* entre as linhas 8 e 10. Nota-se que a operação de projeção é descrita pelo operador *Project* nas linhas 13 a 17. Nas linhas 14, 15 e 16 ao atribuir o valor *true* o banco de dados entende que apenas esses atributos serão incluídos no resultado com a exceção do campo "_id" que deve ser explicitamente removido. No momento o algoritmo não remove o campo "_id" automaticamente, essa restrição a remoção é válida apenas para o campo "_id" da coleção de origem ou durante a execução de uma sub-consulta (em um operador *Lookup*).

4.2.5 Seleção

A operação de *Seleção* tem como objetivo filtrar as ocorrências de uma entidade de acordo com os critérios de comparação descritos na consulta. A Listagem 4.20 contém uma consulta utlizando uma operação de *Seleção*.

```
1 FROM Person
2 WHERE Person.surname = 'Smith'
3 SELECT *
```

Listagem 4.20 – Exemplo de consulta utilizando a operação de *Seleção*

Na Listagem 4.20 a linha 2 define o critério a ser utilizado na operação de *Seleção*. Neste exemplo o critério definido é selecionar apenas as ocorrências que possuírem o valor do atributo *surname* igual a *Smith*.

```
1 db.PersonCollection.aggregate([{
2     $match: {
```

```
3           $expr: {
4               $eq: ["$fSurname", "Smith"]
5           }
6       }
7   }])
```

Listagem 4.21 – Consulta JavaScript para MongoDB gerada pelo algoritmo a partir da consulta definida na Listagem 4.20

A Listagem 4.21 demonstra a consulta JavaScript gerada pelo algoritmo a partir da consulta definida na Listagem 4.20. Na linha 2 é iniciado o operador *Match* que é composto de uma expressão (linhas 3 a 5) que compara se o atributo *fSurname* é igual a *Smith*.

4.3 Considerações finais

Neste capítulo foi apresentado o trabalho de implementação e desenvolvimento realizado, sendo a adaptação de metamodelos representando um modelo entidade-relacionamento (ER), um esquema para o banco de dados MongoDB e para o mapeamento entre o modelo ER e o MongoDB. Foi ampliada a implementação da operação de *Junção* para permitir a junção de várias entidades e junções compostas (junção com o produto de uma junção). Também foram implementadas as seguintes operações propostas pela álgebra de Parent e Spaccapietra (1984); (i) *Produto cartesiano*, (ii) *Projeção* e (iii) *Seleção*.

O funcionamento da operação *Junção* é muito complexo, pois existem diversas formas de representar uma mesma relação entre entidades no MongoDB. Um relacionamento de cardinalidade 1:N, por exemplo, pode ser representado das seguintes maneiras: (i) identificador da entidade origem mapeado para a entidade alvo, (ii) embutir os identificadores da entidade alvo em um atributo multivalorado na entidade origem, (iii) embutir os registros da entidade alvo na entidade origem, ou (iv) entidade origem embutida em um outra entidade qualquer, entre outras opções. Essas são apenas algumas variações possíveis para um relacionamento de cardinalidade 1:N. É importante afirmar que não foi possível implementar o algoritmo de maneira a cobrir todas as possíveis variações. Decidiu-se então aplicar as seguintes restrições à operação *Junção*:

- A entidade de origem não pode estar embutida em outra entidade.
- Não é permitido embutir apenas o identificador da entidade alvo na entidade de origem.
- A entidade alvo em um relacionamento N:M não pode estar embutida na coleção do relacionamento.

Essas restrições possuem impacto na complexidade do código gerado para MongoDB. A primeira limitação que implica em embutir a entidade de origem a uma outra entidade poderia gerar uma junção onde a coleção de origem também é a coleção alvo, essa limitação deve-se à implementação do algoritmo, não há limitação conceitual para relacionamento entre ocorrências de uma mesma entidade.

A segunda restrição evita que algoritmo falhe na localização dos atributos de uma entidade. O uso de mapeamentos parciais pode ser considerado inválido pelo algoritmo por não fornecer referência adequada para a construção de uma operação ou por não incluir mapeamento para atributos utilizados nas operações de *projeção* e/ou *seleção*.

A terceira limitação tem como objetivo evitar o incremento da complexidade do código gerado, evitando operações adicionais para manipular a estrutura do documento e também evitar que em consultas complexas envolvendo *junções de junções* o algoritmo falhe devido a primeira restrição.

Uma outra limitação relativa à operação *Junção* é a falta de suporte à herança de relacionamentos após a junção das entidades como foi proposto por Parent e Spaccapietra (1984). Como a álgebra define uma estrutura para os resultados, esta estrutura agrupa as entidades de acordo com suas relações e após cada operação formam um novo tipo de entidade.

```
1 FROM Person
2 RJOIN <Drives> (Car)
3 RJOIN <Repaired> (Garage)
4 SELECT *
```

Listagem 4.22 – Exemplo de consulta com a herança de relacionamentos

A consulta definida na Listagem 4.22 só é possível de ser executada porque a operação da linha 2 resulta em uma nova entidade composta por *Person* e *Car*, e porque *Car* está relacionada a *Garage*. O resultado desta consulta está descrito na Listagem 4.23.

```
1  {
2      name: 'John',
3      surname: 'Smith',
4      salary: 3000,
5      data_Drives: [
6          {
7              Car_id: 1,
8              Car_reg_no: 12687921
9          }
10     ],
11     data_Repaired: [
12         {
```

```
13              Repaired_date: '01/01/2020',
14              Garage_id: 1,
15              Garage_name: 'Moe AutoRepair',
16              Garage_phone: '555 3423-0987'
17          }
18      ]
19  }
```

Listagem 4.23 – Estrutura do resultado da consulta da Listagem 4.22

Na Listagem 4.23 observa-se que os dados da entidade *Car* e *Garage* estão deslocados, apesar de serem relacionadas através de *Repaired*. Um outro ponto importante a se considerar é que para executar essa operação é necessário atualizar o mapeamento das entidades e a composição dos relacionamentos.

O trabalho de Noguera (2018) citado anteriormente mostra que para uma consulta e um mapeamento podem haver diversas possibilidades de geração de código e seu algoritmo foi construído de forma a gerar todas as possíveis variações. Neste trabalho decidiu-se por gerar apenas um código para cada par de consulta/mapeamento, dando preferência ao uso de mapeamentos com o atributo “IsMain = true”, a razão de não gerar todas as possíveis variações para um par de consulta/mapeamento é simplificar a saída do algoritmo.

Para validar a abordagem do trabalho, foram realizados estudos de casos utilizando bancos de dados de aplicações reais. Estes estudados serão vistos com detalhes no próximo capítulo.

5 Avaliação

No Capítulo 2 foram apresentados os fundamentos teóricos utilizados neste trabalho e no Capítulo 4 foram discutidos detalhes da pesquisa. Neste capítulo são expostos os critérios e objetivos de avaliação e os resultados obtidos.

Em resumo, os objetivos deste trabalho foram: (i) implementar a operação de junção para considerar múltiplas entidades, (ii) implementar os operadores *produto cartesiano*, *seleção* e *projeção* e (iii) analisar o desempenho das consultas geradas.

Para avaliar se os objetivos citados foram atingidos, foram definidas três questões para a avaliação, sendo:

Q1. Dado um único modelo ER, uma única consulta Q e diferentes mapeamentos M1, M2, ... Mn, as consultas geradas Q1, Q2, ... Qn produzem resultados R1, R2, ... Rn que são consistentes entre si e com a álgebra proposta por Parent e Spaccapietra (1984)?

Q2. Dado um único modelo ER, uma única consulta Q e diferentes mapeamentos M1, M2, ... Mn, as consultas geradas Q1, Q2, ... Qn tem desempenho significativamente diferente entre si?

Q3. O desempenho das consultas geradas pelo algoritmo é significativamente diferente quando comparado a consultas escritas à mão?

A questão Q1 tem propósito de validação, ou seja, confirmar que a implementação realizada está correta e de fato atende ao que é definido na álgebra. As questões Q2 e Q3 tem propósito de avaliar o impacto desta pesquisa na prática. Em particular, a questão Q2 irá indicar se é possível melhorar o desempenho de uma consulta alterando apenas o mapeamento, que é o objetivo maior desta pesquisa iniciada no trabalho de Noguera (2018), Noguera e Lucrédio (2019). E a questão Q3 analisa se o processo de geração de código é capaz de entregar código nativo cujo desempenho seja aceitável.

Para responder essas perguntas foram utilizadas as ferramentas *Newtonsoft.JSON* e *FluentAssertions* para validar a igualdade dos documentos resultantes das consultas. A igualdade é atingida quando os pares chave-valor dos documentos são iguais, sendo que a ordem de apresentação dos pares não é levada em consideração. A avaliação do desempenho é feita a partir da medição do tempo de execução de consulta. Para minimizar quaisquer efeitos de otimização do banco de dados foi medida a média de 100 execuções para cada consulta. Essa medição foi realizada através do método “explain” disponível no

MongoDB. Esse método descreve detalhes da execução da consulta, incluindo o tempo de execução.

Descritas as perguntas que deseja-se responder e os critérios/métodos utilizados para obter tais respostas, espera-se que:

- Todos os documentos sejam iguais, para cada consulta, independentemente do mapeamento utilizado;
- Espera-se que o desempenho das consultas esteja diretamente relacionado com o mapeamento utilizado, ou seja, se o mapeamento exige mais operações espera-se que o tempo de execução dessa consulta seja maior;
- Para as consultas escritas à mão, espera-se obter tempos similares, ainda que um leve ganho das consultas escritas à mão não seja surpreendente pois o algoritmo não faz nenhum tipo de otimização no código gerado;
- A diferença de desempenho possa ser explicada de acordo com a quantidade e complexidade das operações executadas nos casos mais lentos.

A avaliação foi construída utilizando dois sistemas reais: o *Progradweb* e o *MKCMS*, que foram recortados (no sentido de serem utilizados apenas em parte devido ao tamanho do sistema) e adaptados para que fosse possível analisar todas as possibilidades previstas pelo algoritmo. Os testes dos sistemas foram separados em duas seções descrevendo o modelo ER utilizado, os mapeamentos, as consultas e os códigos gerados.

5.1 Validação das operações

De todas as operações suportadas pelo algoritmo, apenas a operação *Produto Cartesiano* é gerada da mesma maneira, ou seja, independentemente da consulta essa operação gera um código similar ao apresentado na Listagem 4.17.

Para as operações *Junção*, *Seleção* e *Projeção* foram criados testes automatizados que comparam o resultado de uma consulta gerada pelo algoritmo com o resultado de uma consulta escrita por um engenheiro de software. Os testes validam diferentes combinações de consultas e mapeamentos.

Apenas para a operação *Junção* foram executados 22 (vinte e dois) testes. Cada teste foi executado utilizando consultas diferentes e mapeamentos diferentes (apenas um mapeamento por teste). O objetivo dos testes é validar o comportamento do algoritmo para diversas combinações de relacionamentos, mapeamentos e diferentes cardinalidades. Para cada teste foi criado um Modelo ER, um banco de dados MongoDB e um mapeamento exclusivamente para o teste. O banco de dados foi populado utilizando um gerador

de dados para testes de acordo com o modelo e mapeamento referentes ao teste. Para cada entidade foram criados aproximadamente dez registros.

Para a execução das consultas geradas e das consultas manuais foi desenvolvido um script que executa uma consulta JavaScript no MongoDB através do driver do banco de dados para C#. A Listagem 5.1 contém o código do teste para validar a geração de consultas utilizando a operação *Junção* entre duas entidades relacionadas por meio de um relacionamento do tipo 1:N. A entidade alvo foi mapeada para a mesma coleção da entidade origem. Na linha 1 são carregados os modelos ER, Esquema MongoDB e mapeamento, na linha 3 é carregada a consulta escrita de maneira nativa, usando JavaScript, pelo engenheiro de software. Nas linhas 7 até 21 é especificada a consulta utilizando a abordagem ER desta pesquisa. Nas linhas 5, 23, 30 e 31 são realizadas comparações para verificar se as consultas não são nulas e o resultado do banco de dados não é nulo. As linhas 27 e 28 mostram a execução das consultas no banco de dados. A linha 33 executa a comparação dos dois resultados e verifica se as respostas do banco de dados são idênticas.

```
1  RequiredDataContainer ModelData =
       OneToManyRelationshipsDataProvider.OneToManyEmbedded();
2
3  string HandcraftedQuery = Utils.ReadQueryFromFile( "
       HandcraftedQueries/personCarOneToMany_2.js" );
4
5  Assert.IsNotNull( HandcraftedQuery );
6
7  RelationshipJoinOperator RJoinOp = new RelationshipJoinOperator(
8      new QueryableEntity( (Entity)ModelData.
       EntityRelationshipModel.FindByName( "Person" ), "person" ),
9      (Relationship)ModelData.EntityRelationshipModel.FindByName( "
       Drives" ),
10     new List<QueryableEntity> {
11         new QueryableEntity( (Entity)ModelData.
       EntityRelationshipModel.FindByName( "Car" ), "car" )
12     },
13     ModelData.ERMongoMapping );
14
15 List<AlgebraOperator> OpList = new List<AlgebraOperator> {
       RJoinOp };
16 FromArgument StartArg = new FromArgument( new QueryableEntity(
       ModelData.EntityRelationshipModel.FindByName( "Person" ) ),
17     ModelData.ERMongoMapping );
18
19 QueryGenerator QueryGen = new QueryGenerator( StartArg, OpList );
```

```
20
21 string GeneratedQuery = QueryGen.Run();
22
23 Assert.IsNotNull( GeneratedQuery );
24
25 QueryRunner Runner = new QueryRunner( "mongodb://localhost
      :27017", "researchDatabaseOneToMany" );
26
27 string HandcraftedResult = Runner.GetJSON( HandcraftedQuery );
28 string GeneratedResult = Runner.GetJSON( GeneratedQuery );
29
30 Assert.IsNotNull( HandcraftedResult );
31 Assert.IsNotNull( GeneratedResult );
32
33 Assert.IsTrue( JToken.DeepEquals( JToken.Parse( HandcraftedResult
       ), JToken.Parse( GeneratedResult ) ) );
```

Listagem 5.1 – Código do teste para validar uma consulta utilizando a operação *Junção* e um relacionamento com cardinalidade 1:N e entidade alvo mapeada para a mesma coleção da entidade de origem.

Todos os testes foram desenvolvidos de maneira similar ao exemplo da Listagem 5.1 e estão disponíveis em um repositório no GitHub[1].

Foram realizados dois estudos de caso a fim de validar os resultados do algoritmo frente a duas bases de dados reais.

A primeira base de dados utilizada é o Prograd Web. Trata-se de um sistema de controle acadêmico que foi utilizado durante muitos anos na UFSCar. Este sistema é de grande porte. A estrutura do banco de dados possui 155 tabelas e 27 “views”. Para ser utilizado nesta pesquisa, o modelo ER foi extraído da seguinte forma: foram analisadas as “views” (que é a visualização do resultado uma consulta no banco de dados) e foram escolhidas as consultas de variados níveis de complexidade, desde uma simples junção entre duas entidades a uma operação mais complexa envolvendo múltiplas junções. A partir do modelo extraído, foram selecionadas 6 entidades e 5 relacionamentos.

A segunda base de dados pertence a um sistema de marketing digital ao qual deu-se o nome *MKCMS*. Essa base de dados possui apenas 4 tabelas e seu modelo representa toda a base de dados, totalizando apenas 4 entidades e 3 relacionamentos.

Um ponto importante é que para ambos os sistemas, apenas o modelo ER estava disponível, os dados para teste foram gerados através de ferramentas de geração de dados

[1] https://github.com/slowvoid/mongoQueryGenerator/tree/master/MongoQueryGenerator/UnitTests

para teste (popularmente chamadas de *data fakers*) e scripts criados nas linguagens C# e JavaScript.

Para o sistema Progradweb foram gerados 50 registros para a entidade *Aluno*, 50 registros para a entidade *Endereço*, 50 registros para a entidade *Matricula*, 10 registros para a entidade *Enfase*, 11 registros para a entidade *Curso* e 50 registros para a entidade *Disciplina*. A quantidade de registros foi definida a fim de gerar dados em quantidade suficiente para expressar a complexidade do modelo.

Para o sistema MKCMS, foram gerados 50 registros para a entidade *User*, 50 registros para a entidade *Product*, 17 registros para a entidade *Category* e 20 registros para a entidade *Store*.

5.2 Estudo de caso: Sistema Progradweb

O sistema Progradweb é software de controle acadêmico que foi utilizado pela UFSCar. O modelo ER desse sistema é extenso, decidiu-se então utilizar apenas uma parte do modelo original. No total foram utilizadas 9 (nove) consultas contemplando todas as entidades e relacionamentos presentes no modelo utilizado. O modelo ER resultante é suficiente para validar a utilização do algoritmo para gerar consultas utilizando as operações propostas, pois inclui o uso de todas as operações suportadas pelo algoritmo.

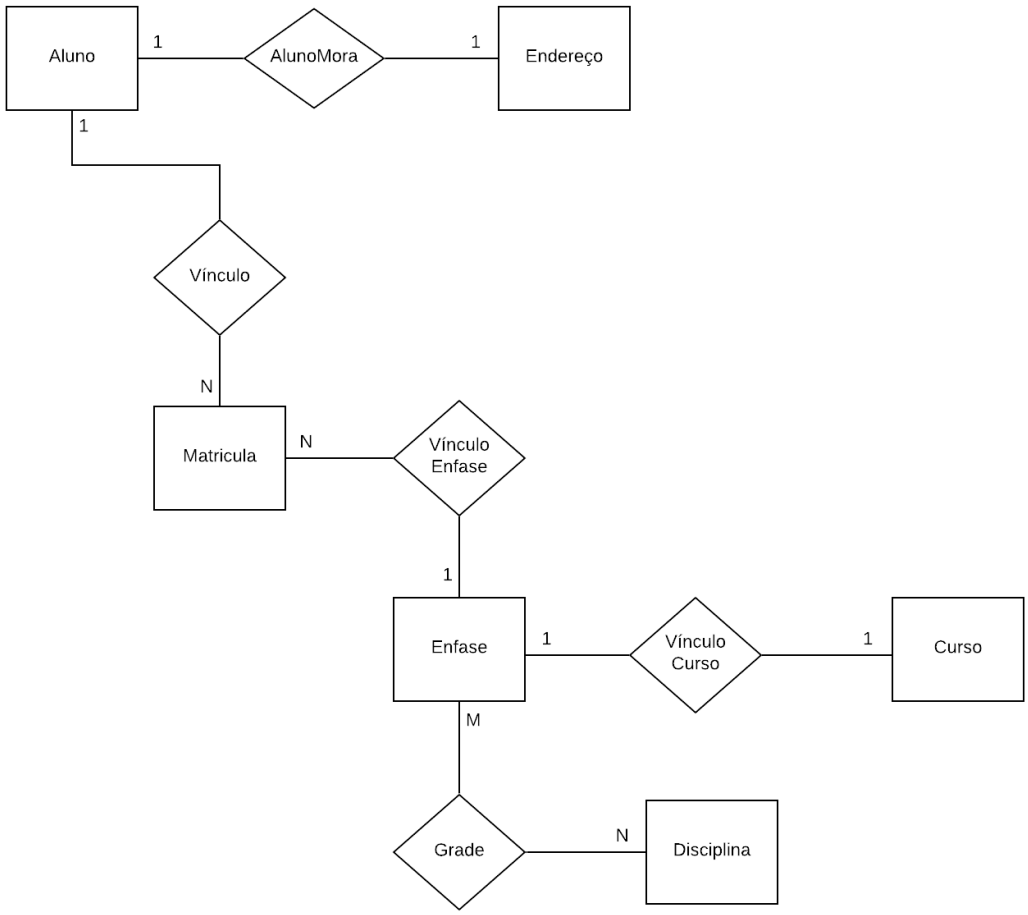

Figura 34 – Modelo ER do sistema Progradweb

A Figura 34 representa o modelo ER para o sistema Progradweb. Os testes para este sistema foram desenvolvidos com base em nove consultas reais utilizadas pelo sistema, as consultas foram extraídas das "views" do banco de dados do sistema. Para cada consulta foram utilizados mapeamentos diferentes, no total foram criados 9 (nove) mapeamentos, os mapeamentos foram criados de modo a respeitar as limitações citadas no capítulo 4.

```
Aluno {
    codalu_alug: int
    nomealu_alug: string
    cpf_alug: string
}

Endereco {
    codend_end: int
    logradouro_end: string
```

```
    bairro_end: string
    compl_end: string
    cep_end: string
    codcidade_end: string
}

Matricula {
    codmatr_matr: int
    anoini_matr: int
    semiini_matr: int
}

Disciplina {
    coddiscip_discip: int
    nome_discip: string
}

Enfase {
    codenf_enf: int
    nomeenf_enf: string
    siglaenf_enf: string
}

Curso {
    codcur_cur: int
    sigla_cur: string
    nome_cur: string
}

AlunoMora {}

Vinculo {}

VinculoEnfase {}

VinculoCurso {}

Grade {
    gradegrad_id: int
    perfil_grd: int
    userid_grd: int
```

```
}
```

Listagem 5.2 – Modelo ER para o Sistema Progradweb com detalhes de atributos

A Listagem 5.2 representa o modelo ER do sistema Progradweb (apresentado de forma simplificada na Figura 34) de forma compatível com o algoritmo. Os mapeamentos serão apresentados nas próximas seções de acordo com as consultas discutidas.

5.2.1 Consulta 1

Com base no modelo ER da Figura 34, foi testada a consulta descrita na Listagem 5.3. A consulta realiza a junção das entidades *Aluno* e *Endereco* através do relacionamento *AlunoMora*.

```
FROM Aluno a
RJOIN <AlunoMora> (Endereco e)
SELECT *
```

Listagem 5.3 – Consulta Teste entre as entidades *Aluno* e *Endereco*

Para testar a consulta definida na Listagem 5.3 foram utilizados dois mapeamentos diferentes. Uma vez que o relacionamento *AlunoMora* possui cardinalidade 1:1, os seguintes mapeamentos foram escolhidos:

```
Aluno < Aluno*, Endereco >
{
    _id             : int       < Aluno.codalu_alug >
    nomealu_alug    : string    < Aluno.nomealu_alug >
    cpf_alug        : string    < Aluno.cpf_alug >
    endid_alug      : int       < Endereco.codend_end >
}

Endereco < Endereco* >
{
    _id:            : int       < Endereco.codend_end >
    logradouro_end  : string    < Endereco.logradouro_end >
    bairro_end      : string    < Endereco.bairro_end >
    compl_end       : string    < Endereco.compl_end >
    cep_end         : string    < Endereco.cep_end >
    codcidade_end   : string    < Endereco.codcidade_end >
}
```

Listagem 5.4 – Mapeamento entre modelo ER e MongoDB utilizado na consulta da Listagem 5.3

Na Listagem 5.4 está descrito um dos mapeamentos utilizados para a consulta da Listagem 5.3. Nota-se que ambas as entidade possuem mapeamento com *IsMain=True* (linhas 1 e 10) e nota-se também que existe uma referência da entidade *Endereco* mapeada para a entidade *Aluno* (linha 6). Esta referência indica a cardinalidade 1:1 na relação entre as duas entidades.

```
1  Aluno < Aluno*, Endereco >
2  {
3      _id              : int       < Aluno.codalu_alug >
4      nomealu_alug     : string    < Aluno.nomealu_alug >
5      cpf_alug         : string    < Aluno.cpf_alug >
6      endereco {
7          endid_alug       : int       < Endereco.codend_end >
8          logradouro_end   : string    < Endereco.logradouro_end >
9          bairro_end       : string    < Endereco.bairro_end >
10         compl_end        : string    < Endereco.compl_end >
11         cep_end          : string    < Endereco.cep_end >
12         codcidade_end    : string    < Endereco.codcidade_end >
13     }
14 }
```

Listagem 5.5 – Segundo mapeamento entre modelo ER e MongoDB utilizado na consulta da Listagem 5.3

A Listagem 5.5 mostra uma outra forma de realizar o mapeamento entre as entidades *Aluno* e *Endereco*. Nota-se que a entidade *Endereco* foi incorporada em um atributo complexo da entidade *Aluno*, o que é compatível com a cardinalidade do relacionamento 1:1.

O teste da consulta da Listagem 5.3 está completamente reproduzido na Listagem 5.6.

```
1  RequiredDataContainer Container1 = PrograduateWebDataProvider.
       MapEntitiesToCollections();
2  RequiredDataContainer Container2 = PrograduateWebDataProvider.
       MapEntitiesToCollectionsEnderecoEmbedded();
3
4  QueryableEntity Aluno = new QueryableEntity( Container1.
       EntityRelationshipModel.FindByName( "Aluno" ) );
5  QueryableEntity Endereco = new QueryableEntity( Container1.
       EntityRelationshipModel.FindByName( "Endereco" ) );
6
7  RelationshipJoinOperator RJoinOp1 = new RelationshipJoinOperator(
8      Aluno,
```

```
9     (Relationship)Container1.EntityRelationshipModel.FindByName(
      "AlunoMora" ),
10    new List<QueryableEntity>() { Endereco },
11    Container1.ERMongoMapping );
12
13 SortArgument SortArg1 = new SortArgument( Aluno, Aluno.Element.
      GetIdentifierAttribute(), MongoDBSort.Ascending );
14 SortStage SortOp1 = new SortStage( new List<SortArgument>() {
      SortArg1 }, Container1.ERMongoMapping );
15
16 List<AlgebraOperator> OperatorList1 = new List<AlgebraOperator>()
      { RJoinOp1, SortOp1 };
17
18 FromArgument FromArg1 = new FromArgument( Aluno, Container1.
      ERMongoMapping );
19
20 RelationshipJoinOperator RJoinOp2 = new RelationshipJoinOperator(
21    Aluno,
22    (Relationship)Container2.EntityRelationshipModel.FindByName(
      "AlunoMora" ),
23    new List<QueryableEntity>() { Endereco },
24    Container2.ERMongoMapping );
25
26 SortArgument SortArg2 = new SortArgument( Aluno, Aluno.Element.
      GetIdentifierAttribute(), MongoDBSort.Ascending );
27 SortStage SortOp2 = new SortStage( new List<SortArgument>() {
      SortArg2 }, Container2.ERMongoMapping );
28
29 List<AlgebraOperator> OperatorList2 = new List<AlgebraOperator>()
      { RJoinOp2, SortOp2 };
30
31 FromArgument FromArg2 = new FromArgument( Aluno, Container2.
      ERMongoMapping );
32
33
34 QueryGenerator QueryGen1 = new QueryGenerator( FromArg1,
      OperatorList1 );
35 QueryGenerator QueryGen2 = new QueryGenerator( FromArg2,
      OperatorList2 );
36
37 string Query1 = QueryGen1.Run();
38 string Query2 = QueryGen2.Run();
```

```
39
40 Assert.IsNotNull( Query1, "Query1 cannot be null" );
41 Assert.IsNotNull( Query2, "Query2 cannot be null" );
42
43 QueryRunner QueryRunner1 = new QueryRunner( "mongodb://localhost
      :27017", "progradweb_1" );
44 QueryRunner QueryRunner2 = new QueryRunner( "mongodb://localhost
      :27017", "progradweb_2" );
45
46 string QueryResult1 = QueryRunner1.GetJSON( Query1 );
47 string QueryResult2 = QueryRunner2.GetJSON( Query2 );
48
49 Assert.IsNotNull( QueryResult1, "QueryResult1 cannot be null" );
50 Assert.IsNotNull( QueryResult2, "QueryResult2 cannot be null" );
51
52 JToken JSONResult1 = JToken.Parse( QueryResult1 );
53 JToken JSONResult2 = JToken.Parse( QueryResult2 );
54 JSONResult1.Should().BeEquivalentTo( JSONResult2 );
```

Listagem 5.6 – Código em C# para teste da consulta definida na Listagem 5.3

Nas linhas 1 e 2 da Listagem 5.6 são geradas as informações necessárias para o algoritmo, sendo a linha 1 referente ao mapeamento da Listagem 5.4 e a linha 2 referente ao mapeamento da Listagem 5.5. A classe *RequiredDataContainer* agrupa as seguintes estruturas, Modelo ER, Esquema MongoDB e o Mapeamento ER-MongoDB. Nas linhas 4 e 5 são criadas uma instância para a entidade *Aluno* e uma para a entidade *Endereco*. O tipo *QueryableEntity* é uma estrutura derivada de *Entity* que fornece ao algoritmo funcionalidades/facilidades para lidar com uma entidade. Nas linhas 7, 8, 9, 10 e 11 é definida a operação *Junção* para o mapeamento da Listagem 5.4. As linhas 13 e 14 definem uma operação de ordenação pelo atributo identificador da entidade *Aluno* de forma ascendente. A ordenação foi utilizada para simplificar a comparação e evitar falso-positivos devido a ordem dos resultados. Nas linhas 16 e 18 são definidas a lista de operações a sem executada para o mapeamento e o argumento de entrada da consulta a ser gerada (Esse argumento é sempre a entidade definida depois de *FROM* na consulta).

As linhas 20 a 24, 26, 27, 28 e 31 repetem os mesmos passos citados anteriormente mas utilizando o mapeamento da Listagem 5.5. Nas linhas 34 e 35 é inicializado o algoritmo para a geração das consultas informando a entidade de partida (*Person*) e a lista de operações a ser executada. Nas linhas 37 e 38 o algoritmo é executado, sendo retornada uma string contendo a consulta JavaScript para MongoDB gerada. As linhas 40 e 41 verificam se as consultas geradas não são nulas. As linhas 43 e 44 inicializam o código responsável por executar as consultas no MongoDB. As linhas 46, 47, 49 e 50

são, respectivamente, a execução das consulta para os mapeamentos e a verificação se os resultados informados são diferente de nulo. O resultado da execução da consulta no banco de dados é no formato *JSON*. As linhas 52 e 53 interpretam essas resultados e a linha 54 executa uma comparação entre ambos e caso seja detectado alguma diferença entre os resultados o teste é considerado falho.

A Listagem 5.7 é a consulta JavaScript para MongoDB gerada pelo algoritmo utilizando o mapeamento 5.4. Nas linhas 2 a 7 é realizada a busca pela entidade *Endereco* onde o valor do seu identificador é igual valor do mesmo atributo mapeado para a entidade *Aluno*. Como o relacionamento possui cardinalidade 1:1 utiliza-se o comando *$unwind* para transformar o valor de *data_Endereco_join* em monovalorado (As operações *lookup* do MongoDB por padrão sempre resultam em um atributo multivalorado). Nas linhas 14 a 21 é criado um novo atributo complexo para armazenar os dados da entidade *Endereco*. Nas linhas 23 a 27 é adicionado um novo atributo ao documento contendo os dados da entidade *Endereco* já no formato definido pela álgebra. Os outros comandos são operações de projeção que removem atributos não utilizados/criados apenas para processamento e ordenação.

```
1  db.Aluno.aggregate([{
2      $lookup: {
3          from: "Endereco",
4          as: "data_Endereco_join",
5          foreignField: "_id",
6          localField: "endid_alug"
7      }
8  }, {
9      $unwind: {
10         path: "$data_Endereco_join",
11         preserveNullAndEmptyArrays: true
12     }
13 }, {
14     $addFields: {
15         "data_Endereco.Endereco_codend_end": "$data_Endereco_join
       ._id",
16         "data_Endereco.Endereco_logradouro_end": "
       $data_Endereco_join.logradouro_end",
17         "data_Endereco.Endereco_bairro_end": "$data_Endereco_join
       .bairro_end",
18         "data_Endereco.Endereco_compl_end": "$data_Endereco_join.
       compl_end",
19         "data_Endereco.Endereco_cep_end": "$data_Endereco_join.
       cep_end",
```

```
20          "data_Endereco.Endereco_codcidade_end": "
    $data_Endereco_join.codcidade_end"
21      }
22 }, {
23      $addFields: {
24          data_AlunoMora: [{
25              $mergeObjects: ["$data_Endereco"]
26          }]
27      }
28 }, {
29      $project: {
30          data_Endereco: false,
31          data_Endereco_join: false
32      }
33 }, {
34      $sort: {
35          "_id": 1
36      }
37 }, {
38      $project: {
39          "endid_alug": false
40      }
41 }])
```

Listagem 5.7 – Código JavaScript para MongoDB gerado a partir da consulta da Listagem 5.3 utilizando o mapeamento da Listagem 5.4

A Listagem 5.8 contém o código JavaScript para MongoDB criado a partir do mapeamento da Listagem 5.5. Para este mapeamento não é necessário buscar dados em outras coleções, sendo necessário apenas manipular a estrutura do documento para o formato definido pela álgebra. Esse processo é realizado entre as linhas 2 e 15. Nas linhas de 17 a 28 são removidos os atributos não utilizados ou que foram criados temporariamente para processamento e o processo de ordenação dos dados.

```
1 db.Aluno.aggregate([{
2      $addFields: {
3          "data_Endereco.Endereco_codend_end": "$endereco.
    enderecoid",
4          "data_Endereco.Endereco_logradouro_end": "$endereco.
    logradouro",
5          "data_Endereco.Endereco_bairro_end": "$endereco.bairro",
6          "data_Endereco.Endereco_compl_end": "$endereco.
    complemento",
```

```
7         "data_Endereco.Endereco_cep_end": "$endereco.cep",
8         "data_Endereco.Endereco_codcidade_end": "$endereco.
    codcidade"
9     }
10 }, {
11     $addFields: {
12         data_AlunoMora: [{
13             $mergeObjects: ["$data_Endereco"]
14         }]
15     }
16 }, {
17     $project: {
18         endereco: false,
19         data_Endereco: false
20     }
21 }, {
22     $sort: {
23         "_id": 1
24     }
25 }, {
26     $project: {
27         "endereco": false
28     }
29 }])
```

Listagem 5.8 – Código JavaScript para MongoDB gerado a partir da consulta da Listagem 5.3 utilizando o mapeamento da Listagem 5.5

5.2.2 Consulta 2

A consulta que está definida na Listagem 5.9 envolve múltiplas operações de junção incluindo consultas secundárias e o uso de entidades computadas.

```
1 FROM Aluno a
2 RJOIN <Vinculo> ( Matricula m
3                   RJOIN <VinculoEnfase> ( Enfase e
4                                           RJOIN <VinculoCurso> (
    Curso c ) ) )
5 RJOIN <AlunoMora> ( Endereco end )
6 SELECT *
```

Listagem 5.9 – Consulta Teste entre as entidades *Aluno*, *Matricula*, *Enfase* e *Curso*

Para a execução do teste definido na Listagem 5.9 foram utilizados três mapeamentos diferentes.

O mapeamento definido na Listagem 5.10 mostra que cada entidade foi mapeada para uma coleção própria no esquema MongoDB, ou seja, possuem a propriedade *IsMain = true*. Nota-se também que o relacionamento *Grade* foi mapeamento para sua própria coleção, caracterizando um relacionamento de cardinalidade N:M (relacionando as entidades *Disciplina* e *Enfase*).

```
1  Aluno < Aluno*, Endereco >
2  {
3      _id              : int       < Aluno.codalu_alug >
4      nomealu_alug     : string    < Aluno.nomealu_alug >
5      cpf_alug         : string    < Aluno.cpf_alug >
6      endid_alug       : int       < Endereco.codend_end >
7  }
8
9  Curso < Curso* >
10 {
11     _id:             : int       < Curso.codcur_cur >
12     sigla_cur        : string    < Curso.sigla_cur >
13     nomecur_cur      : string    < Curso.nomecur_cur >
14 }
15
16 Disciplina < Disciplina* >
17 {
18     _id:             : int       < Disciplina.coddiscip_discip >
19     nome_discip      : string    < Disciplina.nome_discip >
20 }
21
22 Endereco < Endereco* >
23 {
24     _id:             : int       < Endereco.codend_end >
25     logradouro_end   : string    < Endereco.logradouro_end >
26     bairro_end       : string    < Endereco.bairro_end >
27     compl_end        : string    < Endereco.compl_end >
28     cep_end          : string    < Endereco.cep_end >
29     codcidade_end    : string    < Endereco.codcidade_end >
30 }
31
32 Enfase < Enfase*, Curso >
33 {
34     _id:             : int       < Enfase.codenf_enf >
```

```
35      nomeenf_enf      : string     < Enfase.nomeenf_enf >
36      siglaenf_enf     : string     < Enfase.siglaenf_enf >
37      codcur_enf       : int        < Curso.codcur_cur >
38  }
39
40  Grade < Grade*, Disciplina, Enfase >
41  {
42      _id              : int        < Grade.gradegrad_id >
43      perfil_grd       : int        < Grade.perfil_grd >
44      userid_grd       : int        < Grade.userid_grd >
45      discipgrad_id    : int        < Disciplina.coddiscip_discip >
46      enfgrad_id       : int        < Enfase.codenf_enf >
47  }
48
49  Matricula < Matricula*, Aluno, Enfase >
50  {
51      _id:             : int        < Matricula.codmatr_matr >
52      anoini_matr      : int        < Matricula.anoini_matr >
53      semiini_matr     : int        < Matricula.semiini_matr >
54      codalu_matr      : int        < Aluno.codalu_alug >
55      codenf_matr      : int        < Enfase.codenf_enf >
56  }
```

Listagem 5.10 – Mapeamento 1 entre o Modelo ER da Figura 34 e Esquema MongoDB

A Listagem 5.11 contém o mapeamento 2 entre o modelo ER (Figura 34) e esquema MongoDB. Esse mapeamento é muito semelhante ao mapeamento 1 (5.10), a diferença é que a entidade *Endereco* foi incorporada à coleção *Aluno* (Linhas 6 a 13).

```
1   Aluno < Aluno*, Endereco >
2   {
3       _id              : int        < Aluno.codalu_alug >
4       nomealu_alug     : string     < Aluno.nomealu_alug >
5       cpf_alug         : string     < Aluno.cpf_alug >
6       endereco {
7           endid_alug       : int        < Endereco.codend_end >
8           logradouro_end   : string     < Endereco.logradouro_end >
9           bairro_end       : string     < Endereco.bairro_end >
10          compl_end        : string     < Endereco.compl_end >
11          cep_end          : string     < Endereco.cep_end >
12          codcidade_end    : string     < Endereco.codcidade_end >
13      }
14  }
```

```
15
16 Curso < Curso* >
17 {
18     _id:            : int       < Curso.codcur_cur >
19     sigla_cur       : string    < Curso.sigla_cur >
20     nomecur_cur     : string    < Curso.nomecur_cur >
21 }
22
23 Disciplina < Disciplina* >
24 {
25     _id:            : int       < Disciplina.coddiscip_discip >
26     nome_discip     : string    < Disciplina.nome_discip >
27 }
28
29 Enfase < Enfase*, Curso >
30 {
31     _id:            : int       < Enfase.codenf_enf >
32     nomeenf_enf     : string    < Enfase.nomeenf_enf >
33     siglaenf_enf    : string    < Enfase.siglaenf_enf >
34     codcur_enf      : int       < Curso.codcur_cur >
35 }
36
37 Grade < Grade*, Disciplina, Enfase >
38 {
39     _id             : int       < Grade.gradegrad_id >
40     perfil_grd      : int       < Grade.perfil_grd >
41     userid_grd      : int       < Grade.userid_grd >
42     discipgrad_id   : int       < Disciplina.coddiscip_discip >
43     enfgrad_id      : int       < Enfase.codenf_enf >
44 }
45
46 Matricula < Matricula*, Aluno, Enfase >
47 {
48     _id:            : int       < Matricula.codmatr_matr >
49     anoini_matr     : int       < Matricula.anoini_matr >
50     semiini_matr    : int       < Matricula.semiini_matr >
51     codalu_matr     : int       < Aluno.codalu_alug >
52     codenf_matr     : int       < Enfase.codenf_enf >
53 }
```

Listagem 5.11 – Mapeamento 2 entre o Modelo ER da Figura 34 e Esquema MongoDB

O mapeamento definido na Listagem 5.12 também é similar ao mapeamento 1

(Listagem 5.10, a diferença sendo apenas a incorporação da entidade *Curso* a coleção *Enfase* (Linhas 30 a 34).

```
1  Aluno < Aluno*, Endereco >
2  {
3      _id              : int       < Aluno.codalu_alug >
4      nomealu_alug     : string    < Aluno.nomealu_alug >
5      cpf_alug         : string    < Aluno.cpf_alug >
6      endid_alug       : int       < Endereco.codend_end >
7  }
8
9  Disciplina < Disciplina* >
10 {
11     _id:             : int       < Disciplina.coddiscip_discip >
12     nome_discip      : string    < Disciplina.nome_discip >
13 }
14
15 Endereco < Endereco* >
16 {
17     _id:             : int       < Endereco.codend_end >
18     logradouro_end   : string    < Endereco.logradouro_end >
19     bairro_end       : string    < Endereco.bairro_end >
20     compl_end        : string    < Endereco.compl_end >
21     cep_end          : string    < Endereco.cep_end >
22     codcidade_end    : string    < Endereco.codcidade_end >
23 }
24
25 Enfase < Enfase*, Curso >
26 {
27     _id:             : int       < Enfase.codenf_enf >
28     nomeenf_enf      : string    < Enfase.nomeenf_enf >
29     siglaenf_enf     : string    < Enfase.siglaenf_enf >
30     curso: {
31         cursoId      : int       < Curso.codcur_cur >
32         sigla:       : string    < Curso.sigla_cur >
33         nome:        : string    < Curso.nomecur_cur >
34     }
35 }
36
37 Grade < Grade*, Disciplina, Enfase >
38 {
39     _id              : int       < Grade.gradegrad_id >
```

```
    perfil_grd      : int       < Grade.perfil_grd >
    userid_grd      : int       < Grade.userid_grd >
    discipgrad_id   : int       < Disciplina.coddiscip_discip >
    enfgrad_id      : int       < Enfase.codenf_enf >
}

Matricula < Matricula*, Aluno, Enfase >
{
    _id:            : int       < Matricula.codmatr_matr >
    anoini_matr     : int       < Matricula.anoini_matr >
    semiini_matr    : int       < Matricula.semiini_matr >
    codalu_matr     : int       < Aluno.codalu_alug >
    codenf_matr     : int       < Enfase.codenf_enf >
}
```

Listagem 5.12 – Mapeamento 3 entre o Modelo ER da Figura 34 e Esquema MongoDB

A construção dos testes é muito similar, tendo como principal diferença a forma como a operação de *Junção* é feita, caso seja utilizada. Na Listagem 5.13 é exibido apenas trecho do teste que constrói as operações de junção.

```
ComputedEntity EnfaseCursoCE = new ComputedEntity( "EnfaseCursoCE
   ",
    Enfase,
    (Relationship)Container1.EntityRelationshipModel.FindByName(
   "VinculoCurso" ),
    new List<QueryableEntity>() { Curso } );

ComputedEntity MatriculaEnfaseCE = new ComputedEntity( "
   MatriculaEnfaseCE",
    Matricula,
    (Relationship)Container1.EntityRelationshipModel.FindByName(
   "VinculoEnfase" ),
    new List<QueryableEntity>() { new QueryableEntity(
   EnfaseCursoCE ) } );

RelationshipJoinOperator RJoinOp1_A = new
   RelationshipJoinOperator(
    Aluno,
    (Relationship)Container1.EntityRelationshipModel.FindByName(
   "Vinculo" ),
    new List<QueryableEntity>() { new QueryableEntity(
   MatriculaEnfaseCE ) },
```

```
15     Container1.ERMongoMapping );
16
17 RelationshipJoinOperator RJoinOp1_B = new
       RelationshipJoinOperator(
18     Aluno,
19     (Relationship)Container1.EntityRelationshipModel.FindByName(
       "AlunoMora" ),
20     new List<QueryableEntity>() { Endereco },
21     Container1.ERMongoMapping );
```

Listagem 5.13 – Trecho de código C# utilizado no teste da consulta da Listagem 5.9

Na Listagem 5.13 é representada a construção da operação *Junção* incluindo as junções secundárias que compõe as linhas 2, 3 e 4 da consulta da Listagem 5.9. Para execução das junções secundárias é necessário construí-las de "dentro para fora", primeiramente faz-se a relação (entidade computada) entre as entidades *Enfase* e *Curso* (linhas 1 a 4), então uma nova entidade computada é criada fazendo a ligação entre a entidade *Matricula* e a composição entre *Enfase* e *Curso* (linhas 6 a 9). O próximo passo é construir a operação *Junção* unindo a entidade de origem (*Aluno*) com a entidade computada *MatriculaEnfaseCE* através do relacionamento *Vinculo* (linhas 11 a 15). Em sequência é construída a operação de junção para unir a entidade *Aluno* com a entidade *Endereco* (linhas 17 a 21).

A Listagem 5.14 contém o código gerado pelo algoritmo para os mapeamentos das Listagens 5.10 e 5.11, as linhas 34, 38 e 50 indicam que parte do código da consulta foi omitido. Nota-se nas linhas 8 e 21 o inicio de sub consultas (O termo *pipeline* dentro de uma operação *Lookup* especifica quais outras operações serão executas na coleção de destino). Neste caso o MongoDB irá executar uma sequência de consultas ao acessar as coleções *Matricula* e *Enfase*.

```
1  db.Aluno.aggregate([{
2      $lookup: {
3          from: "Matricula",
4          as: "data_Matricula",
5          let: {
6              source_codalu_alug: "$_id"
7          },
8          pipeline: [{
9              $match: {
10                 $expr: {
11                     $eq: ["$codalu_matr", "$$source_codalu_alug"]
12                 }
13             }
```

```
14          }, {
15              $lookup: {
16                  from: "Enfase",
17                  as: "data_Enfase",
18                  let: {
19                      source_codenf_enf: "$codenf_matr"
20                  },
21                  pipeline: [{
22                      $match: {
23                          $expr: {
24                              $eq: ["$_id", "$$source_codenf_enf"]
25                          }
26                      },
27                      {$lookup: {
28                          from: "Curso",
29                          as: "data_Curso_join",
30                          foreignField: "_id",
31                          localField: "codcur_enf"
32                      }
33                  },
34                  ...
35                  ]
36              }
37          }
38          ...
39          ]
40      }
41  }, {
42      $addFields: {
43          data_Vinculo: "$data_Matricula"
44      }
45  }, {
46      $project: {
47          data_Matricula: false
48      }
49  },
50  ...
51  ])
```

Listagem 5.14 – Código JavaScript para MongoDB gerado para a consulta da Listagem 5.9 utilizando os mapeamentos das Listagens 5.10 e 5.11

A Listagem 5.15 contém um fragmento do código gerado pela algoritmo utilizando

o mapeamento da Listagem 5.12. A diferença para o código da Listagem 5.14 é a forma como é feita a junção da entidade *Curso* que neste caso foi incorporada a entidade *Enfase*.

```
...
$lookup: {
    from: "Enfase",
    as: "data_Enfase",
    let: {
        source_codenf_enf: "$codenf_matr"
    },
    pipeline: [{
        $match: {
            $expr: {
                $eq: ["$_id", "$$source_codenf_enf"]
            }
        }
    }, {
        $addFields: {
            "data_Curso.Curso_codcur_cur": "$curso.cursoId",
            "data_Curso.Curso_sigla_cur": "$curso.sigla",
            "data_Curso.Curso_nomecur_cur": "$curso.nome"
        }
    }, {
        $addFields: {
            data_VinculoCurso: [{
                $mergeObjects: ["$data_Curso"]
            }]
        }
    }, {
        $project: {
            curso: false,
            data_Curso: false
        }
    }, {
        $project: {
            "curso": false
        }
    }, {
        $addFields: {
            Enfase_codenf_enf: "$_id",
            Enfase_nomeenf_enf: "$nomeenf_enf",
            Enfase_siglaenf_enf: "$siglaenf_enf"
```

```
40          }
41      }, {
42          $project: {
43              _id: false,
44              nomeenf_enf: false,
45              siglaenf_enf: false
46          }
47      }]
48  }
49  ...
```

Listagem 5.15 – Código JavaScript para MongoDB gerado para a consulta da Listagem 5.9 utilizando o mapeamento 5.12

Nota-se que não foi exibido o trecho de código gerado para a junção da entidade *Endereco* pois o código é similar ao exibido nas Listagens 5.7 e 5.8.

5.2.3 Consulta 3

Nesta consulta serão utilizadas as operações de *Projeção* e *Seleção*. A consulta está definida na Listagem 5.16.

```
1 FROM Aluno a
2 RJOIN <Mora> ( Endereco e )
3 WHERE e.bairro_end = 'Buckinghamshire'
4 SELECT a.nomealu_alug, e.logradouro_end, e.bairro_end, e.cep_end
```

Listagem 5.16 – Consulta teste entre as entidades *Aluno* e *Endereco* utilizando operações de *Projeção* e *Seleção*

Para gerar a consulta utilizou-se os mapeamentos das Listagens 5.10 e 5.11. O objetivo da consulta da Listagem 5.16 é unir as entidades *Aluno* e *Endereco* (linhas 1 e 2), selecionar apenas os registros que possuírem o valor do atributo *bairro_end* igual a "Buckinghamshire" (linha 3) e então projetar os atributos especificados (linha 4).

```
1  ...
2  {
3      $match: {
4          $expr: {
5              $in: ["Buckinghamshire", "$data_AlunoMora.
       Endereco_bairro_end"]
6          }
7      }
8  }, {
9      $project: {
```

```
10          "nomealu_alug": true,
11          "data_AlunoMora.Endereco_logradouro_end": true,
12          "data_AlunoMora.Endereco_bairro_end": true,
13          "data_AlunoMora.Endereco_cep_end": true
14      }
15  }
16  ...
```

Listagem 5.17 – Código JavaScript gerado a partir da consulta da Listagem 5.16

Nó código da Listagem 5.17 a operação de seleção é definida nas linhas 3 a 7, como neste caso o atributo utilizado para comparação pertence a uma lista (atributo multivalorado, conforme especificado pela álgebra) é necessário utilizar uma comparação do tipo "InArray" que procura o valor "*Buckinhamshire*" em uma lista contendo todos os valores para o atributo "*bairro_end*" da entidade *Endereco* (linha 5).

O código gerado para as operações de *Seleção* e *Projeção* são semelhantes para os dois mapeamentos utilizados, conforme mostra a Listagem 5.17. O código para os dois mapeamentos é semelhante pois a estrutura do documento após a operação *Junção* é a mesma.

5.2.4 Outras consultas

No total foram executados testes para nove consultas do sistema Progradweb, sendo que estas não são apresentadas neste texto pois o código gerado a partir destas consultas não apresenta novos elementos, ou seja, o código JavaScript MongoDB gerado apresenta pequenas variações das operações já apresentadas. Os motivos para essas variações são: (i) a cardinalidade do relacionamento utilizado; e (ii) uma entidade sendo incorporada a outra entidade.

No total, foram testadas nove consultas e nove mapeamentos, resultando em 9 testes e 14 asserções, cada teste é composto de uma ou mais asserções. Neste estudo, 100% dos testes forem bem sucedidos, ou seja, os resultados das diferentes consultas são consistentes entre si e com a álgebra ER.

5.3 Estudo de caso: Sistema MKCMS

O sistema MKCMS é um gerenciador de conteúdo para marketing digital. Seu objetivo é permitir que usuários do sistema pesquisem informações sobre determinados produtos, esse sistema foi mencionado no capítulo 1, o modelo ER apresentado nesta seção é uma variação do modelo da Figura 1. O Modelo ER para esse sistema está descrito na Figura 35. Para simplificar a visualização foram omitidos os atributos das entidades.

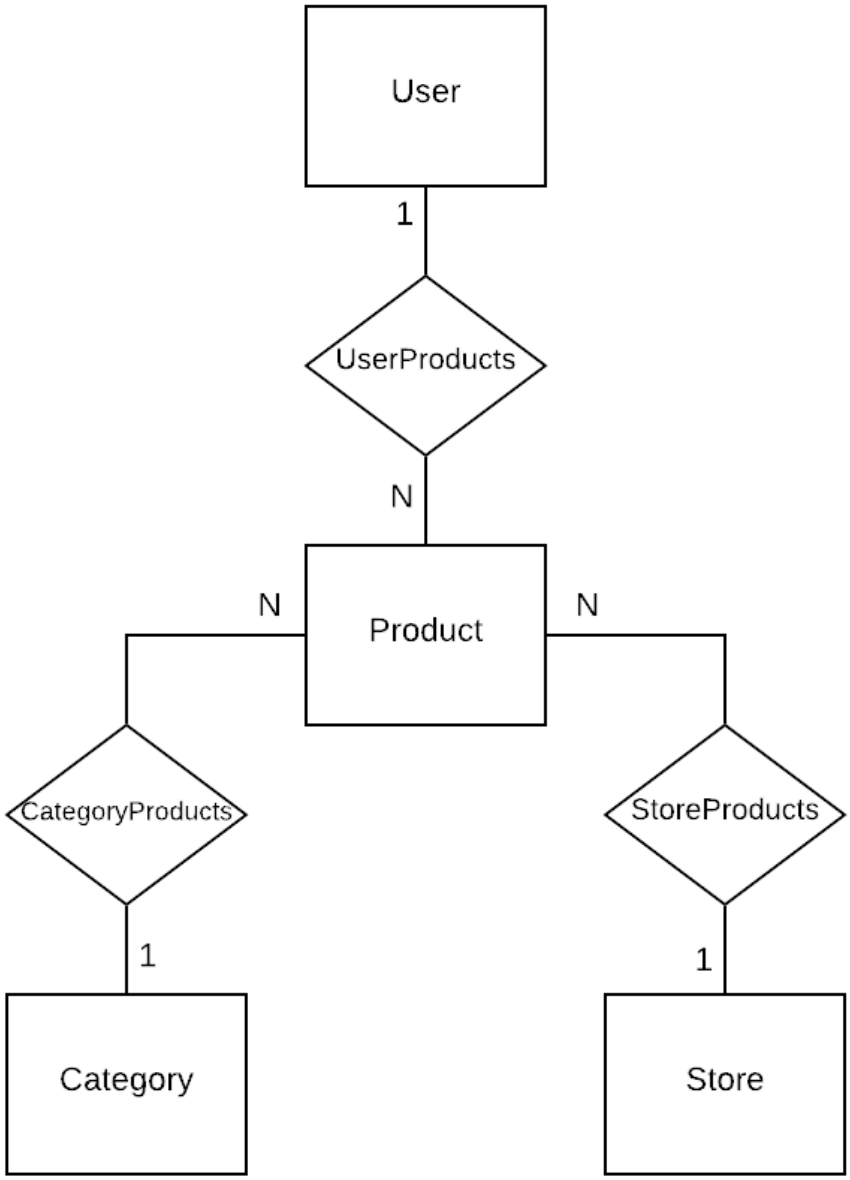

Figura 35 – Modelo ER para o sistema MKCMS

Para os testes foram utilizados cinco mapeamentos diferentes, definidos nas Listagens 5.18, 5.19, 5.20, 5.21 e 5.22.

No mapeamento da Listagem 5.18 cada entidade foi mapeada para uma coleção, ou seja, possuem o atributo "IsMain = true" (linhas 1, 8, 18 e 24).

```
1 User < User* >
2 {
3     _id              : int       < User.UserID >
4     UserName         : string    < User.UserName >
5     UserEmail        : string    < User.UserEmail >
6 }
```

```
7
8  Product < Product*, User, Category, Store >
9  {
10     _id              : int       < Product.ProductID >
11     Title            : string    < Product.Title >
12     Description      : string    < Product.Description >
13     UserID           : int       < User.UserID >
14     CategoryID       : int       < Category.CategoryID >
15     StoreID          : int       < Store.StoreID >
16 }
17
18 Category < Category* >
19 {
20     _id              : int       < Category.CategoryID >
21     CategoryName     : string    < Category.CategoryName >
22 }
23
24 Store < Store* >
25 {
26     _id              : int       < Store.StoreID >
27     StoreName        : string    < Store.StoreName >
28 }
```

Listagem 5.18 – Mapeamento 1 entre Modelo ER da Figura 35 e o Esquema MongoDB

No mapeamento da Listagem 5.19 apenas a entidade *Product* possui mapeamento com o atributo "IsMain = true". As entidades *User*, *Category* e *Store* foram incorporadas a entidade *Product* (linhas 6 até 18).

```
1  Product < Product*, User, Category, Store >
2  {
3      _id              : int       < Product.ProductID >
4      Title            : string    < Product.Title >
5      Description      : string    < Product.Description >
6      user: {
7          _id          : int       < User.UserID >
8          name         : string    < User.UserName >
9          email        : string    < User.UserEmail >
10     }
11     category: {
12         _id          : int       < Category.CategoryID >
13         name         : string    < Category.CategoryName >
14     }
```

```
15      store: {
16          _id          : int        < Store.StoreID >
17          name         : string     < Store.StoreName >
18      }
19  }
```

Listagem 5.19 – Mapeamento 2 entre Modelo ER da Figura 35 e o Esquema MongoDB

No mapeamento da Listagem 5.20, a entidade *Category* foi incorporada a entidade *Product* (linhas 15 a 18). As entidades *Product*, *User* e *Store* possuem mapeamento "IsMain = true" (linhas 1, 8 e 21).

```
1  User < User* >
2  {
3      _id              : int        < User.UserID >
4      UserName         : string     < User.UserName >
5      UserEmail        : string     < User.UserEmail >
6  }
7
8  Product < Product*, User, Category, Store >
9  {
10     _id              : int        < Product.ProductID >
11     Title            : string     < Product.Title >
12     Description      : string     < Product.Description >
13     UserID           : int        < User.UserID >
14     StoreID          : int        < Store.StoreID >
15     category: {
16         _id          : int        < Category.CategoryID >
17         name         : string     < Category.CategoryName >
18     }
19 }
20
21 Store < Store* >
22 {
23     _id              : int        < Store.StoreID >
24     StoreName        : string     < Store.StoreName >
25 }
```

Listagem 5.20 – Mapeamento 3 entre Modelo ER da Figura 35 e o Esquema MongoDB

No mapeamento da Listagem 5.21 as entidades *Product*, *Category* e *User* possuem o atributo "IsMain = true" (Linhas 1, 8 e 21). A entidade *Store* foi incorporada na entidade *Product* (linha 8).

```
1  User < User* >
```

```
2  {
3      _id              : int       < User.UserID >
4      UserName         : string    < User.UserName >
5      UserEmail        : string    < User.UserEmail >
6  }
7
8  Product < Product*, User, Category, Store >
9  {
10     _id              : int       < Product.ProductID >
11     Title            : string    < Product.Title >
12     Description      : string    < Product.Description >
13     UserID           : int       < User.UserID >
14     CategoryID       : int       < Category.CategoryID >
15     store: {
16         _id          : int       < Store.StoreID >
17         name         : string    < Store.StoreName >
18     }
19 }
20
21 Category < Category* >
22 {
23     _id              : int       < Category.CategoryID >
24     CategoryName     : string    < Category.CategoryName >
25 }
```

Listagem 5.21 – Mapeamento 4 entre Modelo ER da Figura 35 e o Esquema MongoDB

No mapeamento da Listagem 5.22 a entidade *User* foi incorporada a entidade *Product* (linha 1). As entidades *Product*, *Category* e *Store* possuem mapeamento com atributo "IsMain = true" (linhas 1, 15 e 21).

```
1  Product < Product*, User, Category, Store >
2  {
3      _id              : int       < Product.ProductID >
4      Title            : string    < Product.Title >
5      Description      : string    < Product.Description >
6      CategoryID       : int       < Category.CategoryID >
7      StoreID          : int       < Store.StoreID >
8      user: {
9          _id          : int       < User.UserID >
10         name         : string    < User.UserName >
11         email        : string    < User.UserEmail >
12     }
```

```
13 }
14
15 Category < Category* >
16 {
17     _id              : int        < Category.CategoryID >
18     CategoryName     : string     < Category.CategoryName >
19 }
20
21 Store < Store* >
22 {
23     _id              : int        < Store.StoreID >
24     StoreName        : string     < Store.StoreName >
25 }
```

Listagem 5.22 – Mapeamento 5 entre Modelo ER da Figura 35 e o Esquema MongoDB

5.3.1 Consulta 1

Esta consulta utiliza os mapeamentos das Listagens 5.18, 5.19, 5.20, 5.21 e 5.22.

A Listagem 5.23 descreve a consulta que faz a junção das entidades *User*, *Store* e *Category* através dos relacionamentos *UserProducts*, *StoreProducts* e *CategoryProducts* respectivamente.

```
1 FROM Product p
2 RJOIN <UserProducts> (User u)
3 RJOIN <StoreProducts> (Store s)
4 RJOIN <CategoryProducts> (Category c)
5 SELECT *
```

Listagem 5.23 – Consulta teste entre as entidades *Product*, *User*, *Category* e *Store*

A Listagem 5.24 contém fragmento do código gerado pelo algoritmo para o mapeamento da Listagem 5.18. A junção de cada entidade gera código similar ao gerado para a entidade *User* (linhas 2 a 31). Na linha 23 é utilizado um operador chamado *$mergeObjects*, esse operador une os atributos informados em um único objeto.

```
1 ...
2 {
3     $lookup: {
4         from: "User",
5         as: "data_User_join",
6         foreignField: "_id",
7         localField: "UserID"
8     }
```

```
}, {
    $unwind: {
        path: "$data_User_join",
        preserveNullAndEmptyArrays: true
    }
}, {
    $addFields: {
        "data_User.User_UserID": "$data_User_join._id",
        "data_User.User_UserName": "$data_User_join.UserName",
        "data_User.User_UserEmail": "$data_User_join.UserEmail"
    }
}, {
    $addFields: {
        data_UserProducts: [{
            $mergeObjects: ["$data_User"]
        }]
    }
}, {
    $project: {
        data_User: false,
        data_User_join: false
    }
}
...
```

Listagem 5.24 – Fragmento do código JavaScript para MongoDB gerado a partir da consulta da Listagem 5.23 e do mapeamento da Listagem 5.18

A Listagem 5.25 representa o código gerado para a junção da entidade *User* que está embutida na entidade *Product*. Esse padrão de código é gerado também para a junção das outras entidades (*Category* e *Store*). Para os outros mapeamentos o código gerado será uma composição dos fragmentos mostrados nas Listagens 5.24 e 5.25.

```
...
{
    $addFields: {
        "data_User.User_UserID": "$user._id",
        "data_User.User_UserName": "$user.name",
        "data_User.User_UserEmail": "$user.email"
    }
}, {
    $addFields: {
        data_UserProducts: [{
```

```
            $mergeObjects: ["$data_User"]
        }]
    }
}, {
    $project: {
        user: false,
        data_User: false
    }
}
...
```

Listagem 5.25 – Fragmento do código JavaScript gerado para MongoDB a partir da consulta da Listagem 5.23 e do mapeamento da Listagem 5.19

5.3.2 Consulta 2

Nesta consulta são realizadas operações de junção utilizando sub-consultas e operação de projeção. A consulta utiliza os mapeamentos das Listagens 5.18, 5.21 e 5.22.

Na Listagem 5.26 é realizada a junção das entidades *Category*, *Product* e *User* (linhas 2 e 3) partindo da entidade *Category*. Na linha 4 são definidos os atributos a serem exibidos no resultado da consulta.

```
1 FROM Category c
2 RJOIN <CategoryProducts> (Product p
3                           RJOIN <UserProducts> (User u))
4 SELECT Category.CategoryName, Product.Title, User.UserName, User.
    UserEmail
```

Listagem 5.26 – Consulta teste entre as entidades *Category*, *Product* e *User*

Na Listagem 5.27 está descrito o trecho do código gerado para a consulta responsável pela projeção dos atributos escolhidos na consulta. Nota-se o uso de uma técnica chamada “dot-notation” que permite o acesso a atributos complexos seguindo a hierarquia onde o identificador a esquerda é o atributo pai do identificador a direita.

```
1 ...
2 {
3     $project: {
4         "data_CategoryProducts.Product_Title": true,
5         "data_CategoryProducts.data_UserProducts.User_UserName":
      true,
6         "data_CategoryProducts.data_UserProducts.User_UserEmail":
      true,
7         "CategoryName": true
```

```
8       }
9   }
10  ...
```

Listagem 5.27 – Código JavaScript para MongoDB gerado a partir da consulta da Listagem 5.26

5.3.3 Consulta 3

Esta consulta tem como objetivo demonstrar uma operação de seleção. A consulta utiliza os mapeamentos das Listagens 5.18, 5.21 e 5.22.

```
1 FROM Category
2 WHERE CategoryName = 'Home'
3 SELECT *
```

Listagem 5.28 – Consulta teste utilizando *seleção* para a entidade *Category*

Na Listagem 5.28 serão selecionados apenas os registros que possuírem o valor "Home" para o atributo *CategoryName* (linha 2).

```
1 db.Category.aggregate([{
2     $match: {
3         $expr: {
4             $eq: ["$CategoryName", "Home"]
5         }
6     }
7 }])
```

Listagem 5.29 – Código JavaScript para MongoDB gerado a partir da consulta da Listagem 5.28

O código da Listagem 5.29 descreve como é feita a operação da seleção para a consulta da Listagem 5.28. É necessário apenas executar o uma operação *Match* (linhas 2 a 6). Nas linhas 3 a 5 é avaliada uma expressão que verifica a igualdade entre o valor do atributo *CategoryName* e o valor "Home".

5.3.4 Outras consultas

No total foram utilizadas doze consultas para testes do sistema MKCMS. As consultas descritas nas seções 5.3.1, 5.3.2 e 5.3.3 demonstram as principais operações suportadas pelo algoritmo. As demais consultas utilizam operações já demonstradas com pequenas variações devido ao mapeamento e cardinalidade dos relacionamentos mas sem alterar a essência do algoritmo.

Neste estudo de caso, assim como no anterior, 100% dos testes foram bem sucedidos, ou seja, os resultados são consistentes entre si e com a álgebra ER.

5.4 Avaliação de desempenho

Para realizar a avaliação do desempenho das consultas geradas pelo algoritmo o banco de dados para o sistema *MKCMS* foi alimentado com uma quantidade maior de dados, cada entidade possui o seguinte número de registros: *Category* 18 registros, *Product* 150000 registros, *Store* 100 registros e *User* 20000 registros. Também foram definidos alguns parâmetros para a execução dos códigos no MongoDB e foi definido que a métrica utilizada para comparação é o tempo de execução da consulta. Os parâmetros de execução são estes:

- Cada consulta foi executada utilizando o modo "explain" do MongoDB que descreve detalhes da execução da consulta.
- Cada consulta foi executada cem vezes a fim de se garantir que o resultado não seja influenciado pelo uso de cache pelo banco de dados e de outros fatores que possam alterar o tempo de execução das consultas.
- Para cada mapeamento foram geradas consultas pelo algoritmo e manualmente.
- Entidade *User* possui vinte mil registros.
- Entidade *Product* possui cento e cinquenta mil registros.

Ao executar uma consulta no modo "explain" do MongoDB, o banco de dados retorna um documento contendo detalhes da execução da consulta, como tempo de execução, uso de índices, entre outros detalhes que ajudam a explicar as diferenças no processamento. O tempo de geração das consultas não foi medido pois o objetivo da avaliação foi comparar o desempenho da consulta gerada com o de uma consulta escrita à mão desconsiderando qualquer interferência do processo de geração da consulta. A geração de código é um processo realizado esporadicamente, apenas no projeto inicial do sistema, quando há novas consultas a serem desenvolvidas, ou quando há mudanças no mapeamento. De qualquer forma, nos testes realizados não foi percebida nenhuma demora significativa na execução do gerador, que aparentemente produz os resultados em tempos na ordem de milissegundos.

Para a execução das consultas foi desenvolvida uma ferramenta para enviar ao MongoDB a consulta a ser executada e armazenar os resultados. As consultas são executadas uma por vez até atingir uma centena de repetições. Após a conclusão das repetições os resultados são armazenados em um banco de dados para análise.

Para cada consulta foi criado também um código correspondente de forma manual e que não segue o padrão de resultado exigido pela álgebra. O objetivo dessa consulta é analisar o impacto que a estrutura de resultados da álgebra tem no desempenho das consultas.

5.4.1 Consulta 1

Nesta consulta foi realizada a junção da entidade *Product* com as entidades *User*, *Category* e *Store*. A consulta está descrita na Listagem 5.30.

```
1 FROM Product p
2 RJOIN <CategoryProducts> (Category c)
3 RJOIN <StoreProducts> (Store s)
4 RJOIN <UserProducts> (User u)
5 SELECT *
```

Listagem 5.30 – Consulta 1: junção entre as entidades *Product*, *User*, *Category* e *Store*

Para a consulta da Listagem 5.30 foram utilizados os seguintes mapeamentos:

- Mapeamento 1 (Listagem 5.18): neste mapeamento cada entidade está mapeada para uma coleção no MongoDB.
- Mapeamento 2 (Listagem 5.19): neste mapeamento as entidades *Category*, *Store* e *User* foram incorporadas a entidade *Product*.
- Mapeamento 3 (Listagem 5.20): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *Category* que foi incorporada a *Product*.
- Mapeamento 4 (Listagem 5.21): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *Store* que foi incorporada a *Product*.
- Mapeamento 5 (Listagem 5.22): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *User* que foi incorporada a *Product*.

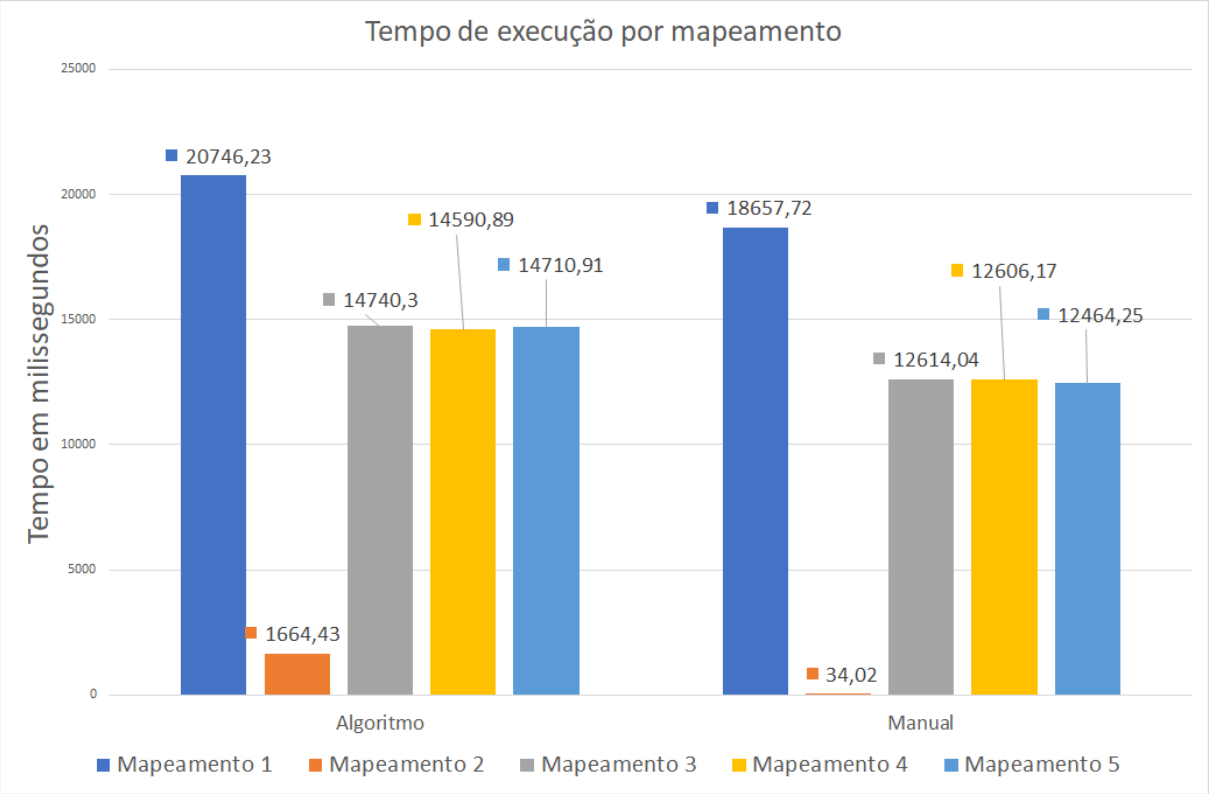

Figura 36 – Comparação de desempenho da consulta da Listagem 5.30

Na Figura 36 os resultados do lado esquerdo foram obtidos através da execução das consultas geradas pelo algoritmo e do lado direito estão os resultados das consultas geradas manualmente. Nota-se que a consulta gerada pelo algoritmo foi mais lenta em todos os casos, com uma diferença em torno de 10 a 15%.

A Tabela 3 contém o tempo de execução das consultas para cada mapeamento, sendo o tempo expresso em milissegundos, considerando a média das 100 execuções. Nota-se que para os mapeamentos 1, 3, 4 e 5 a diferença percentual entre o o código de algoritmo e o gerado manualmente é similar, a diferença do código para esses mapeamentos é a presença de operações adicionais para estruturar o documento de acordo com a álgebra. Para o mapeamento 2, o algoritmo utiliza o "aggregation framework" do MongoDB enquanto a consulta manual utiliza uma simples operação "find".

Mapeamento	**Média (ms)**		**Diferença**
	Algoritmo	**Manual**	
1	20746,23	18657,72	-10,07%
2	1664,43	34,02	-97,96%
3	14740,3	12614,04	-14,42%
4	14590,89	12606,17	-13,60%
5	14710,91	12464,25	-15,27%

Tabela 3 – Comparação do desempenho da consulta da Listagem 5.30

5.4.2 Consulta 2

Nesta consulta foi realizada a junção da entidade *Product* com a entidade *Category*.

```
1 FROM Category c
2 RJOIN <CategoryProducts> (Product p)
3 SELECT *
```

Listagem 5.31 – Consulta 2 junção entre as entidades *Product* e *Category*

Para o teste da consulta da Listagem 5.31 foram utilizados os seguintes mapeamentos:

- Mapeamento 1 (Listagem 5.18): neste mapeamento cada entidade está mapeada para uma coleção no MongoDB.
- Mapeamento 4 (Listagem 5.21): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *Store* que foi incorporada a *Product*.
- Mapeamento 5 (Listagem 5.22): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *User* que foi incorporada a *Product*.

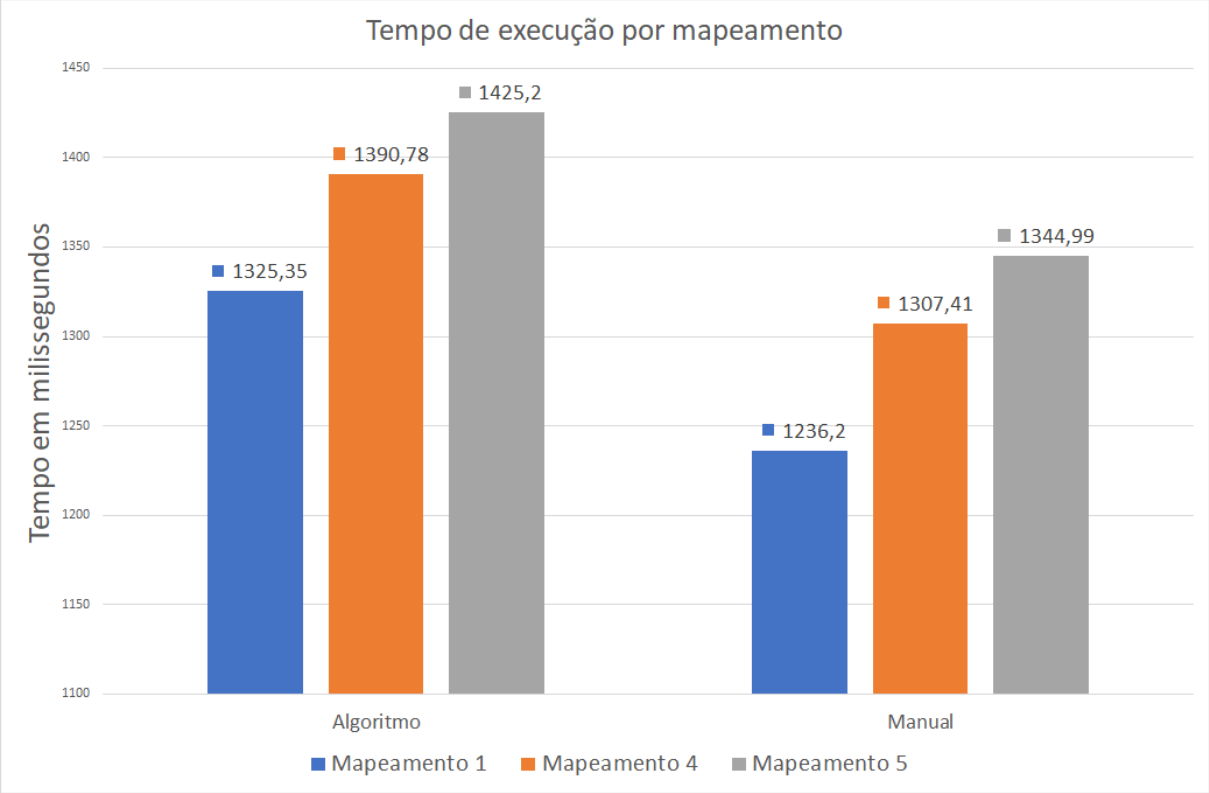

Figura 37 – Comparação de desempenho da consulta da Listagem 5.31

A Figura 37 compara os resultados do desempenho das consultas geradas pelo algoritmo e de consultas escritas a mão, nota-se que assim como na Figura 36 as consultas

do algoritmo foram mais lentas, porém a diferença é menor do que da consulta anterior (cerca de 5%).

Mapeamento	Média (ms)		Diferença
	Algoritmo	Manual	
1	1325,35	1236,2	-6,73%
4	1390,78	1307,41	-5,99%
5	1425,2	1344,99	-5,63%

Tabela 4 – Comparação do desempenho da consulta da Listagem 5.31

Na Tabela 4 nota-se que a variação do desempenho das consultas ao alterar o mapeamento é pequena, sendo aproximadamente 65 milissegundos entre os mapeamentos 1 e 4 e 100 milissegundos entre os mapeamentos 1 e 5. Entre os mapeamentos 4 e 5 a diferença foi de 35 milissegundos. Nota-se também que o código escrito manualmente foi aproximadamente 6% mais rápido do que o código gerado pelo algoritmo.

Os dados da Tabela 4 mostram desempenho próximos entre os mapeamentos utilizados, isso pode ser explicado pela similaridade dos mapeamentos.

5.4.3 Consulta 3

Esta consulta é similar a consulta da Listagem 5.31, incluindo uma outra entidade na junção de *Category* com *Product*. Foram utilizados os seguintes mapeamentos:

- Mapeamento 1 (Listagem 5.18): neste mapeamento cada entidade está mapeada para uma coleção no MongoDB.
- Mapeamento 4 (Listagem 5.21): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *Store* que foi incorporada a *Product*.
- Mapeamento 5 (Listagem 5.22): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *User* que foi incorporada a *Product*.

```
1 FROM Category c
2 RJOIN <CategoryProducts> ( Product p
3                            RJOIN <UserProducts> (User u))
4 SELECT Category.CategoryName, Product.Title, User.UserName, User.
    UserEmail
```

Listagem 5.32 – Consulta 3 junção entre as entidades *Category* e a junção de *Product* com *User*

A Listagem 5.32 descreve a junção da entidade *Category* com o resultado da junção entre as entidades *Product* e *User*, e projeta apenas os atributos definidos na linha 4.

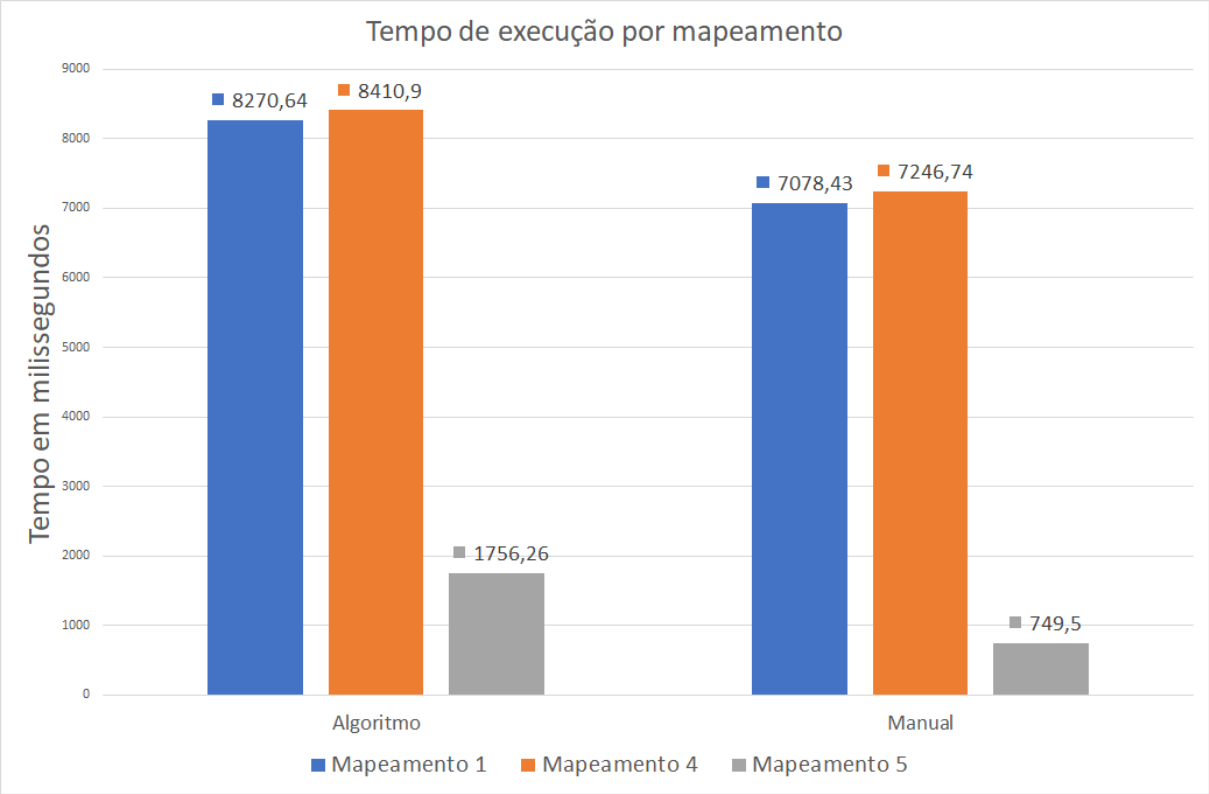

Figura 38 – Comparação de desempenho da consulta da Listagem 5.32

A Figura 38 mostra que para todos os mapeamentos o código criado pelo engenheiro de software teve melhor desempenho que o código gerado pelo algoritmo. A Tabela 5 mostra que o desempenho para os mapeamentos 1 e 4 foi similar, para estes mapeamentos as entidades envolvidas na consulta são representadas da mesma maneira. Nota-se que ao incorporar a entidade *User* (mapeamento 5) o desempenho da consulta é significativamente melhor, sendo aproximadamente 78,76% e 79,12% mais rápido quando comparado com os mapeamentos 1 e 4 respectivamente (código gerado pelo algoritmo) e respectivamente 89,51% e 89,66% para o código escrito manualmente.

Mapeamento	**Média (ms)**		**Diferença**
	Algoritmo	**Manual**	
1	8270,64	7078,43	-14,41%
4	8410,9	7246,74	-13,84%
5	1756,26	749,5	-57,32%

Tabela 5 – Comparação do desempenho da consulta da Listagem 5.32

Nota-se que segundo a Tabela 5 o código escrito pelo engenheiro de software foi 14,41% mais rápido que o gerado pelo algoritmo para o mapeamento 1, 13,84% para o mapeamento 4 e 57,32% mais rápido ao utilizar o mapeamento 5. É importante citar que no mapeamento 5 a entidade *User* foi incorporada a entidade *Product*, evitando assim utilizar uma operação *lookup* para a entidade *User*. Nota-se também que a diferença do tempo de

execução entre o algoritmo e a consulta "manual" é próxima de 1000 milissegundos para todos os mapeamentos.

5.4.4 Consulta 4

Nesta consulta foi realizada a junção da entidade *Product* com as entidades *Category*, *Store* e *User* selecionando apenas as ocorrências em que o valor do atributo *Price* é inferior a "5". A consulta está descrita na Listagem 5.33.

```
1 FROM Product p
2 RJOIN <CategoryProducts> ( Category c )
3 RJOIN <StoreProducts> ( Store s )
4 RJOIN <UserProducts> ( User u )
5 WHERE p.Price < 5
6 SELECT *
```

Listagem 5.33 – Consulta 4 junção entre as entidades *Product*, *Category*, *Store* e *User* utilizando uma operação de *seleção*

Para a consulta da Listagem 5.33 foram utilizados os seguintes mapeamentos:

- Mapeamento 1 (Listagem 5.18): neste mapeamento cada entidade está mapeada para uma coleção no MongoDB.
- Mapeamento 2 (Listagem 5.19): neste mapeamento as entidades *Category*, *Store* e *User* foram incorporadas a entidade *Product*.
- Mapeamento 3 (Listagem 5.20): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *Category* que foi incorporada a *Product*.
- Mapeamento 4 (Listagem 5.21): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *Store* que foi incorporada a *Product*.
- Mapeamento 5 (Listagem 5.22): neste mapeamento as entidades foram mapeadas para coleções distintas exceto *User* que foi incorporada a *Product*.

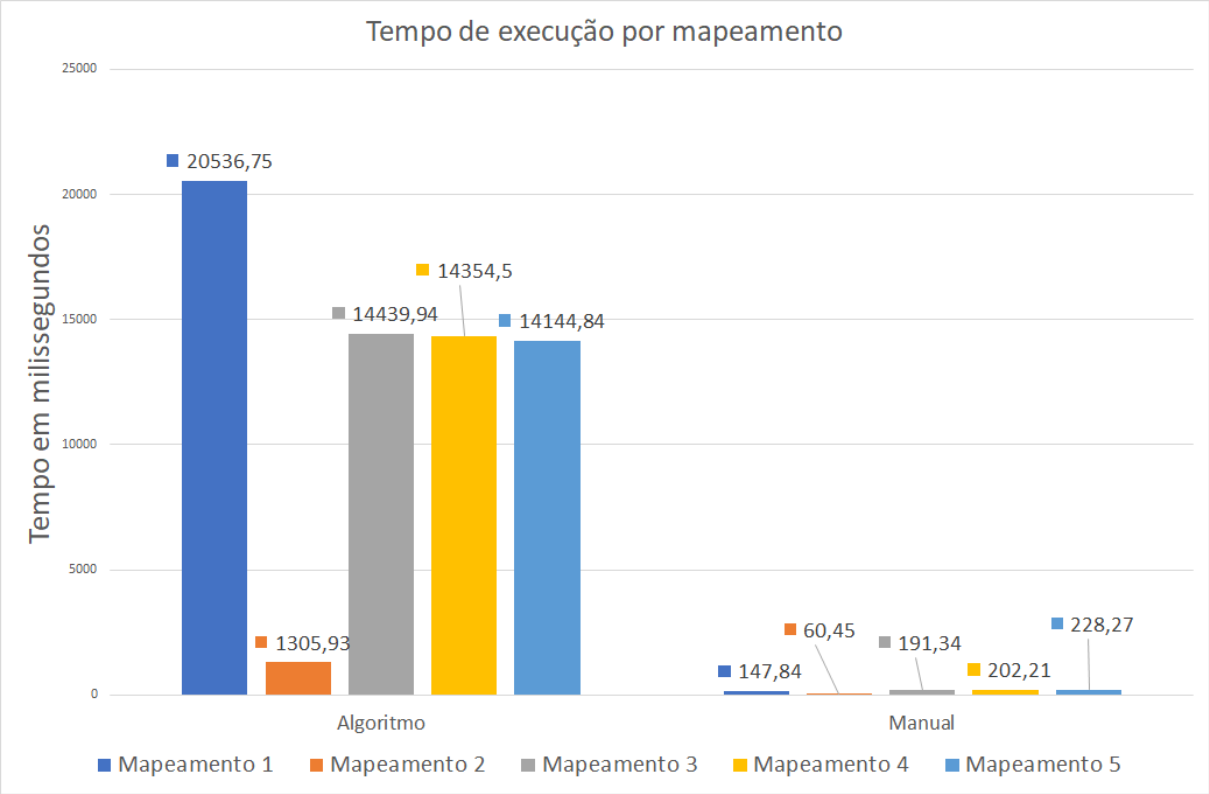

Figura 39 – Comparação de desempenho da consulta da Listagem 5.33

A Figura 39 mostra a comparação do desempenho das consultas para cada mapeamento, do lado esquerdo do gráfico estão os resultados das consultas geradas pelo algoritmo e do lado direito os resultados das consultas escritas manualmente. Nota-se que em todos os mapeamentos o tempo de execução das consultas geradas pelo algoritmo foram muito superiores às consultas geradas manualmente. A Tabela 6 permite melhor análise da diferença.

Mapeamento	**Média (ms)**		**Diferença**
	Algoritmo	**Manual**	
1	20536,75	147,84	-99,28%
2	1305,93	60,45	-95,37%
3	14439,94	191,34	-98,67%
4	14354,5	202,21	-98,59%
5	14144,84	228,27	-98,39%

Tabela 6 – Comparação do desempenho da consulta da Listagem 5.33

A Tabela 6 mostra com detalhes as diferenças do desempenho entre as consultas geradas pelo algoritmo e as geradas manualmente, em todos os casos a diferença foi superior a 95%. É importante notar que esta consulta é similar à consulta 1 (Listagem 5.30) e que a consulta manual foi em média 13,34% mais rápida (desconsiderando o mapeamento 2 que possui diferença acima de 95%) sendo que a única diferença é o uso

de uma operação de seleção na consulta da Listagem 5.33. Para detalhar a questão, as Listagens 5.34 e 5.35 exibem parte do código feito à mão e pelo algoritmo, respectivamente. Em ambas as listagens o código foi gerado utilizando o mapeamento 1 (Listagem 5.18).

```
1  db.Product.explain('executionStats').aggregate([
2      {$match: {
3          $expr: {
4              $lt: ['$Price', 5]
5          }
6      }},
7      {"$lookup": {
8          from: 'Category',
9          foreignField: '_id',
10         localField: 'CategoryID',
11         as: 'data_Category'
12     }},
13     ...
14 ])
```

Listagem 5.34 – Fragmento de código gerado manualmente para a consulta da Listagem 5.33

Nas Listagens 5.34 e 5.35 as linhas 13 da primeira listagem e as linhas 8 e 15 da segunda listagem indicam trechos de código que foram omitidos.

```
1  db.Product.explain('executionStats').aggregate([{
2      $lookup: {
3          from: "Store",
4          as: "data_Store_join",
5          foreignField: "_id",
6          localField: "StoreID"
7      }
8      ...
9  }, {
10     $match: {
11         $expr: {
12             $lt: ["$Price", 5]
13         }
14     },
15     ...
16 }]);
```

Listagem 5.35 – Fragmento de código gerado pelo algoritmo para a consulta da Listagem 5.33

As Listagens 5.34 e 5.35 mostram que a diferença principal entre os códigos gerados é a posição da operação de seleção, que está definida nas linhas 2 até 6 (Listagem 5.34 e nas linhas 10 até 14 (Listagem 5.35). O fato de realizar a operação de seleção antes de qualquer outra operação (Listagem 5.34) indica que as outras operações (junção das entidades *Category*, *Store* e *User*) foram realizadas apenas para as ocorrências que já satisfazem a condição definida na operação de seleção. O código da Listagem 5.35 indica que as junções foram realizadas para todas as ocorrências de *Product* e somente após foram removidos os registros que não satisfizeram os critérios de seleção.

```
1  db.Product.aggregate([{
2      $match: {
3          $expr: {
4              $lt: ["$Price", 5]
5          }
6      }
7  }, {
8      $lookup: {
9          from: "Store",
10         as: "data_Store_join",
11         foreignField: "_id",
12         localField: "StoreID"
13     },
14     ...
15 }]);
```

Listagem 5.36 – Fragmento de código gerado pelo algoritmo e modificado para executar a operação de seleção antes de qualquer outra operação

Para comprovar se a diferença de desempenho entre o algoritmo e a consulta gerada a mão é justificada pelo momento em que a operação de seleção é executada, foi executado um novo teste utilizando uma versão modificada do código gerado pelo algoritmo executando a operação de seleção antes de qualquer outra operação. Esse código está descrito na Listagem 5.36. Nota-se que como no código da Listagem 5.34 a operação de seleção é a primeira na ordem de execução.

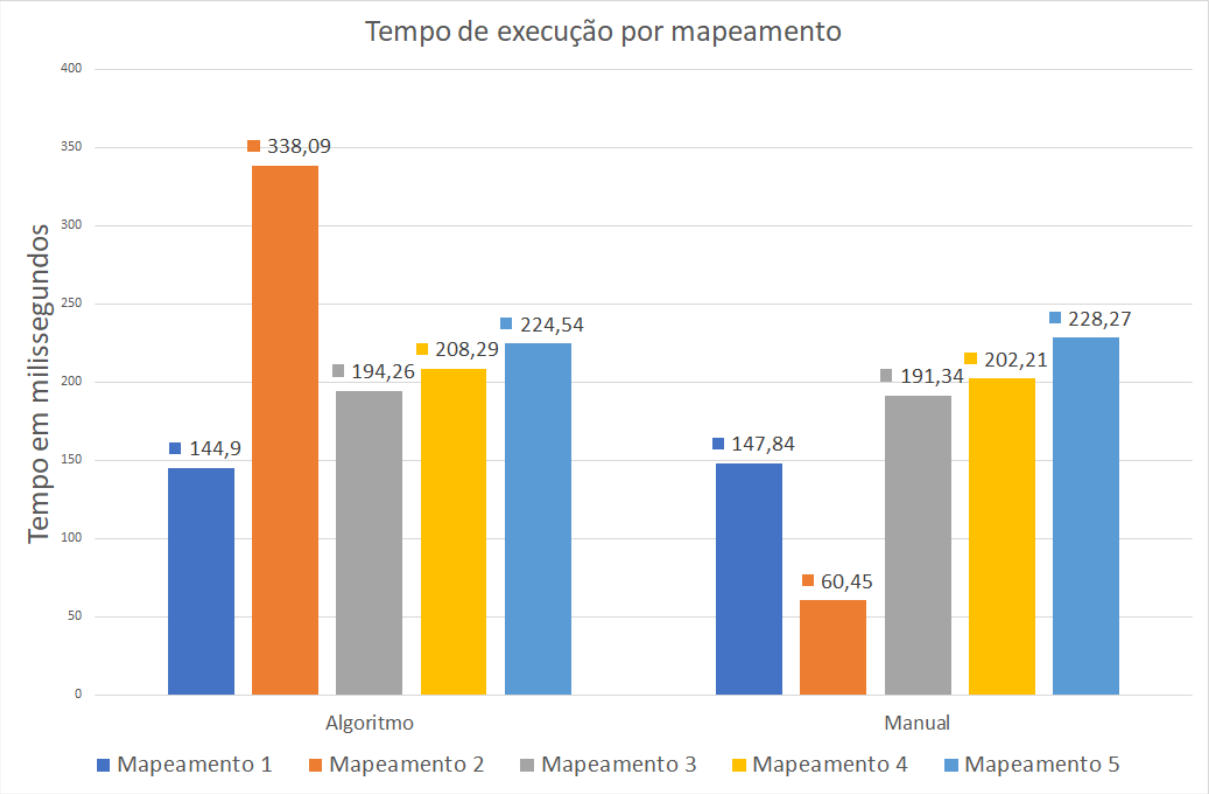

Figura 40 – Comparação entre o código do algoritmo modificado e a consulta manual

A Figura 40 contém os resultados dos códigos modificados (Listagem 5.36) e os resultados dos códigos escritos à mão. Nota-se que a diferença foi reduzida significativamente. A Tabela 7 descreve os resultados em detalhes.

Mapeamento	**Média (ms)**		**Diferença**
	Algoritmo	**Manual**	
1	144,9	147,84	2,03%
2	338,09	60,45	-82,12%
3	194,26	191,34	-1,50%
4	208,29	202,21	-2,92%
5	224,54	228,27	1,66%

Tabela 7 – Comparação entre o código do algoritmo modificado e a consulta manual

Ao analisar a Tabela 7 nota-se que a diferença entre o desempenho do algoritmo e do código manual foi reduzida a valores inferiores a 3%, e no caso dos mapeamentos 1 e 5 o desempenho do algoritmo foi 2,03% e 1,66% superior as consultas escritas a mão para os respectivos mapeamentos. Nota-se também que para o mapeamento 2 a diferença ainda é percentualmente alta apesar do tempo de execução ser 74,11% menor do que o resultado sem modificação. Essa diferença pode ser explicada pelo fato de que no mapeamento 2 todas as entidades foram incorporadas a entidade *Product* e o Engenheiro de Software ao escrever a consulta optou por usar uma operação “find”. Outra possibilidade para o

tempo de execução deste mapeamento ser maior que os demais é o tamanho físico de cada documento, por incluir todas as entidades em uma mesma coleção o tamanho médio de cada documento (em kilobytes) é superior ao dos outros mapeamentos. A Listagem 5.37 contém o código utilizado para o mapeamento 2.

```
1 db.Product.find({Price: {$lt: 5}}).explain('executionStats')
```

Listagem 5.37 – Consulta para o mapeamento 2 escrita pelo Engenheiro de Software

A Listagem 5.37 mostra que a consulta da Listagem 5.33 pode ser resumida em um simples comando para MongoDB quando todas as entidades estão incorporadas em uma única coleção, ou seja, não há operações de junção e quando não há a necessidade de alterar a estrutura do documento resultante. Neste caso, a decisão de se utilizar o "aggregate" em todos os casos, conforme discutido no final da Seção 4.2, se mostrou não ideal. Um refinamento no algoritmo, a ser implementado futuramente, poderia decidir pelo uso de "find" nos casos onde não há junção a ser realizada.

5.5 Conclusão

Neste capítulo foram analisadas as possibilidades de se construir um algoritmo para gerar consultas para o banco de dados MongoDB utilizando um Modelo ER, o mapeamento entre o Modelo ER e o Esquema MongoDB utilizando as operações propostas na álgebra de Parent e Spaccapietra (1984).

Para validar a proposta deste trabalho foram desenvolvidos testes automatizados para validar as operações de forma individual e para validar a conformidade dos resultados com a álgebra. Os testes das operações compararam os resultados do código gerado pelo algoritmo com os do código escrito à mão. Os testes de conformidade com a álgebra foram realizados utilizando os sistemas *Progradweb* e *MKCMS*, cada teste utilizava uma consulta real desses sistemas e diversos mapeamentos, então os resultados obtidos pelos códigos gerados pelo algoritmo eram comparados entre si.

Contando todos testes para validação das operações e conformidade com a álgebra, foram executados no total 61 testes e 410 asserções. Esses testes foram executados de maneira automática utilizando ferramentas de teste de unidade do software *Visual Studio*. Todos os testes foram executados com resultado positivo.

Considerando as análises realizadas pode-se concluir que é possível obter o mesmo resultado para uma consulta alterando apenas o seu mapeamento. Conclui-se também que o código gerado pelo algoritmo tem desempenho inferior ao de uma consulta escrita manualmente e sem as alterações de estruturas exigidas pela álgebra. O desempenho de uma consulta escrita manualmente é excluindo valores muito discrepantes em média 15,73% superior ao das consultas geradas pelo algoritmo, para calcular esse valor foi

excluída a consulta da Listagem 5.33 pois a diferença de desempenho obtido nessa consulta é devido a implementação do algoritmo e não a um problema conceitual. Considerando os valores obtidos na Tabela 7 a diferença média do tempo de execução é 11,29% inferior para as consultas construídas a mão.

Apesar de a comparação apontar vantagem para as consultas manuais, as mesmas foram escritas de forma a não utilizar as regras definidas pela álgebra de Parent e Spaccapietra (1984) e considerando que com as devidas otimizações é possível que as consultas geradas pelo algoritmo consigam atingir desempenho similar ou superior a uma consulta escrita mão, a Tabela 7 mostra que é possível obter melhor desempenho apesar de executar mais operações.

6 Considerações Finais

No capítulo 1 foi apresentada a motivação para este trabalho sendo parte desses motivos a velocidade com que as mudanças acontecem e novos tipos e estruturas de dados são necessárias e a falta de normalização em bancos de dados NoSQL (SADALAGE; FOWLER, 2013). O capítulo 2 contém a revisão teórica dos conceitos aplicados nesta dissertação e no capítulo 3 foi realizada breve revisão de trabalhos relacionados.

Este trabalho foi descrito com detalhes no capítulo 4. Discutiu-se a implementação das operações propostas pela álgebra de Parent e Spaccapietra (1984) e também a adaptação dos metamodelos criados por Noguera (2018), Noguera e Lucrédio (2019). No capítulo 5 foram apresentados os resultados deste trabalho.

6.1 Contribuições

Após as discussões do trabalho e apresentação dos resultados nos capítulos anteriores considera-se que os objetivos foram alcançados. Como já demonstrado por Noguera e Lucrédio (2019) é possível aplicar as operações da álgebra proposta por Parent e Spaccapietra (1984) para bancos de dados NoSQL.

O primeiro objetivo deste trabalho foi incrementar a implementação da operação de junção e implementar as outras operações propostas na álgebra de Parent e Spaccapietra (1984). Esse objetivo foi parcialmente atingido. As operações *Redução*, *Compressão*, *União*, *Intersecção* e *Diferença* não foram implementadas. A operação de *Seleção* foi implementada mas não foi testado o uso de expressões lógicas e aritméticas complexas.

Mesmo não implementando todas as operações propostas pela álgebra considera-se o primeiro objetivo bem sucedido. A implementação da operação de junção foi incrementada para permitir a execução de junções mais complexas envolvendo duas ou mais entidades e o uso do produto de uma junção como argumento para uma outra operação de junção, e também foram implementadas as operações *Produto Cartesiano*, *Projeção* e *Seleção*.

Com essas operações implementadas pode-se concluir que é possível utilizar consultas escritas a partir de um modelo ER em um banco de dados não-relacional, de forma independente de mapeamento. Isso torna o desenvolvimento uma atividade mais conceitual.

O segundo objetivo deste trabalho foi analisar o desempenho das consultas geradas pelo algoritmo e o seu potencial para uso na prática.

Demonstrou-se que é possível melhorar o desempenho de uma consulta sem modificá-la, apenas modificando o mapeamento. Considere-se um banco de dados normalizado construído a partir de um modelo ER e deseja-se migrar esse banco de dados para uma solução NoSQL (MongDB). Existe um sistema que utiliza esse banco de dados e existem 100 consultas diferentes para esse banco de dados. Com o resultado desta pesquisa é possível modificar a estrutura do banco de dados sem que as consultas precisem ser reescritas. Demonstrou-se também que o impacto de se manipular a estrutura do documento resultante é baixo. O fator de maior impacto no desempenho das consultas é o mapeamento conforme discutido no capítulo 5. Com as devidas otimizações feitas à mão no código gerado, foi possível até obter desempenho superior ao de uma consulta escrita à mão.

Considera-se então que a contribuição ao atingir o segundo objetivo foi mostrar que é factível o uso da álgebra de Parent e Spaccapietra (1984) em bancos de dados não-relacionais, que o desempenho está ligado ao mapeamento de dados e que pode-se obter resultado consistente independentemente do mapeamento utilizado.

6.2 Limitações

A proposta inicial pretendia finalizar a implementação dos operadores da álgebra proposta por Parent e Spaccapietra (1984) e analisar o desempenho das consultas geradas. Como discutido nos capítulos 1 e 2 este não é um problema com solução trivial.

Para garantir a viabilidade deste projeto foi necessário tomar decisões que resultaram em limitações, que estão descritas a seguir:

- A operação de junção não permite a herança de relacionamento, ou seja, não é possível executar junções através de relacionamentos onde a entidade origem (EO) é diferente da entidade principal (EP), após a junção de EO.
- Restrições de mapeamento devido a complexidade de operações de junção, as limitações são: (i) entidade de origem não pode estar embutida em outra entidade, (ii) não é permitido embutir apenas o identificador da entidade alvo na entidade de origem e (iii) a entidade alvo em um relacionamento N:M não pode estar embutida na coleção destinada ao relacionamento.
- Operação de seleção suporta apenas expressões lógicas simples.
- Não foram implementadas funções de agregação.
- O algoritmo não otimiza a ordem das operações a serem executadas, o que pode diminuir o desempenho da consulta.
- O algoritmo sempre usa o “aggregate” ao gerar as consultas e em casos mais simples isso pode implicar em perda de desempenho.

6.3 Trabalhos futuros

Potenciais trabalhos futuros podem ser extraídos a partir das limitações citadas anteriormente. Uma sugestão de trabalho é implementar a herança de relacionamentos como foi proposta por Parent e Spaccapietra (1984), isso permitirá novas possibilidades de junções após uma entidade ser adicionada ao conjunto resultado.

Outra sugestão importante é implementar as operações *Redução*, *Compressão*, *União*, *Intersecção* e *Diferença* propostas na álgebra de Parent e Spaccapietra (1984) e que ainda não foram implementadas. Sugere-se também incrementar a implementação da operação de *Seleção* adicionando suporte para expressões lógicas complexas e funções de agregação.

Um potencial tópico para outro projeto é focar na otimização das consultas geradas, conferir ao algoritmo capacidade de ordenar as operações de forma a manter o resultado desejado e potencialmente melhorar o desempenho. Pode-se estudar também o uso de operações "find" nativas do MongoDB ao invés do uso de agregação. Outro ponto de otimização é analisar as operações geradas pelo algoritmo e reduzir/evitar duplicação de operações e/ou operações desnecessárias.

Outra abordagem a ser considerada é simplificar e tornar mais amigável o processo de representação/manipulação dos Modelos ER e Mapeamento. Neste trabalho o modelo ER e o mapeamento são representados de forma textual e depois analisados por um parser para então serem utilizados pelo algoritmo. Talvez uma representação visual torne o processo mais amigável em determinados contextos.

Referências

Abadi, D. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, v. 45, n. 2, p. 37–42, Feb 2012. ISSN 1558-0814. Citado na página 49.

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*. New York, NY, USA: ACM, 2012. (EDBT '12), p. 582–585. ISBN 978-3-642-31094-2. Disponível em: <https://doi.org-/10.1145/2247596.2247671>. Citado 7 vezes nas páginas 11, 22, 33, 59, 64, 66 e 69.

BEERI, C.; BERNSTEIN, P. A.; GOODMAN, N. A sophisticate's introduction to database normalization theory this work was supported in part by the national science foundation under grant mcs-77-05314. In: MYLOPOLOUS, J.; BRODIE, M. (Ed.). *Readings in Artificial Intelligence and Databases*. San Francisco (CA): Morgan Kaufmann, 1989. p. 468 – 479. ISBN 978-0-934613-53-8. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780934613538500352>. Citado na página 26.

BERNSTEIN, P. A.; MELNIK, S. Model management 2.0: Manipulating richer mappings. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2007. (SIGMOD '07), p. 1–12. ISBN 9781595936868. Disponível em: <https:/-/doi.org/10.1145/1247480.1247482>. Citado 5 vezes nas páginas 22, 51, 52, 59 e 71.

BUTGEREIT, L. Four nosqls in four fun fortnights: Exploring nosqls in a corporate it environment. ACM, Johannesburg, South Africa, set. 2016. Disponível em: <https://doi.acm.org/10.1145/2987491.2987494>. Citado na página 47.

CHANG, F. et al. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 26, n. 2, jun. 2008. ISSN 0734-2071. Disponível em: <https://doi.org/10.1145/1365815.1365816>. Citado na página 25.

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 42.

CODD, E. F. Normalized data base structure: a brief tutorial. *SIGFIDET '71 Proceedings of the 1971*, ACM, New York NY, USA, p. 1–17, 1971. Citado na página 26.

COPELAND, G.; MAIER, D. Making smalltalk a database system. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 1984. (SIGMOD '84), p. 316–325. ISBN 0897911288. Disponível em: <https://doi.org/10.1145/602259.602300>. Citado na página 51.

DAVOUDIAN, A.; CHEN, L.; LIU, M. A survey on nosql stores. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 51, n. 2, abr. 2018. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3158661>. Citado na página 26.

ELMASRI, R. et al. *Sistemas de banco de dados*. 6. ed. São Paulo, SP, Brasil: Pearson Addison Wesley, 2010. 808 p. Citado 8 vezes nas páginas 11, 25, 26, 41, 42, 43, 44 e 57.

FRASER, C. et al. Testing cardinality estimation models in sql server. In: *Proceedings of the Fifth International Workshop on Testing Database Systems*. New York, NY, USA: Association for Computing Machinery, 2012. (DBTest '12). ISBN 9781450314299. Disponível em: <https://doi.org/10.1145/2304510.2304526>. Citado na página 57.

FUNCK, R.; JABLONSKI, S. Nosql evaluation: A use case oriented survey. *Proc 2011 Int Conf Cloud Serv Computing*, 12 2011. Citado na página 25.

GORLA, N.; NG, V.; LAW, D. Improving database performance with a mixed fragmentation design. *Journal of Intelligent Information Systems*, v. 39, n. 3, p. 559 – 576, 2012. ISSN 09259902. Citado na página 39.

HEUSER, C. A. *Projeto de banco de dados: Volume 4 da Série Livros didáticos informática UFRGS*. Porto Alegre - RS - Brasil: Bookman Editora, 2009. ISBN 978-85-7780-382-8. Citado 4 vezes nas páginas 11, 43, 44 e 45.

HOBERMAN, S. *Data Modeling for MongoDB: Building Well-Designed and Supportable MongoDB Databases*. Technics Publications, 2014. ISBN 9781935504702. Disponível em: <https://books.google.com.br/books?id=3tngwAEACAAJ>. Citado na página 77.

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 na página 33.

LI, Y.; GU, P.; ZHANG, C. Transforming UML class diagrams into HBase based on meta-model. In: *2014 International Conference on Information Science, Electronics and Electrical Engineering*. Sapporo, Japan: IEEE, 2014. v. 2, p. 720–724. Citado 4 vezes nas páginas 22, 59, 61 e 69.

LIANG, D.; LIN, Y.; DING, G. Mid-model Design Used in Model Transition and Data Migration between Relational Databases and NoSQL Databases. In: *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*. Chengdu, China: IEEE, 2015. p. 866–869. Citado 7 vezes nas páginas 11, 22, 59, 61, 62, 63 e 69.

MASON, R. Nosql databases and data modeling techniques for a document-oriented nosql database. In: *InSITE 2015: Informing Science + IT Education Conferences: USA*. Tampa, Florida, United States: Informing Science Institute, 2015. p. 259–268. ISSN 1535-0703. Citado na página 77.

MELNIK, S. *Generic Model Management: Concepts and Algorithms*. Berlin, Germany: Springer-Verlag Berlin Heidelberg, 2004. ISBN 3-540-21980-3. Citado na página 52.

MONGODB. *Manual MongoDB*. 2019. Acessado: 29/Jan/2019. Disponível em: <https://docs.mongodb.com/manual/>. Citado 2 vezes nas páginas 39 e 95.

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 2 vezes nas páginas 50 e 51.

NOGUERA, V.; LUCRéDIO, D. Implementing a classic er algebra to automatically generate complex queries for document-oriented databases. In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. New York, NY, USA: Association for Computing Machinery, 2019. (SBCARS '19), p. 43–52. ISBN 9781450376372. Disponível em: <https://doi.org/10.1145/3357141.3357601>. Citado 5 vezes nas páginas 7, 9, 66, 105 e 151.

NOGUERA, V. E. R. *Extensão de uma álgebra ER para execução de consultas em bancos de dados NoSQL orientados a documentos*. 2018. Citado 22 vezes nas páginas 11, 15, 22, 33, 34, 35, 36, 38, 59, 66, 67, 68, 69, 73, 74, 81, 89, 94, 95, 104, 105 e 151.

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 32 vezes nas páginas 7, 9, 11, 15, 21, 33, 34, 35, 38, 52, 53, 54, 55, 56, 57, 59, 66, 67, 69, 73, 75, 89, 95, 96, 102, 103, 105, 148, 149, 151, 152 e 153.

PRITCHETT, D. Base: An acid alternative. *Queue*, ACM, New York, NY, USA, v. 6, n. 3, p. 48–55, 2008. Citado 2 vezes nas páginas 48 e 49.

SADALAGE, P. J.; FOWLER, M. *NoSQL Essencial: Um guia conciso para o Mundo emergente da persistência poliglota*. Sâo Paulo - Brasil: Novatec Editora, 2013. ISBN 978-85-7522-338-3. Citado 7 vezes nas páginas 11, 26, 48, 49, 50, 51 e 151.

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 26.

SCHREINER, G. A.; DUARTE, D.; MELLO, R. dos S. SQLtoKeyNoSQL: A Layer for Relational to Key-based NoSQL Database Mapping. In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*. New York, NY, USA: ACM, 2015. (iiWAS '15), p. 74:1–74:9. ISBN 978-1-4503-3491-4. Disponível em: <http://doi.acm.org/10.1145/2837185.2837224>. Citado 7 vezes nas páginas 11, 22, 59, 63, 64, 65 e 69.

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 2 vezes nas páginas 33 e 38.

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 47.

VIESCAS, J.; HERNANDEZ, M. *Sql Queries for Mere Mortals®: A Hands-on Guide to Data Manipulation in Sql, Second Edition*. Second. Boston, MA: Addison-Wesley Professional, 2007. ISBN 9780321444431. Citado 5 vezes nas páginas 11, 45, 46, 47 e 48.

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 49.