INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO CEARÁ
IFCE CAMPUS ARACATI
COORDENADORIA DE CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

EMANUEL MORAES DE ALMEIDA

ALPHA RESTFUL: UM *FRAMEWORK* PARA OTIMIZAÇÕES NA IMPLEMENTAÇÃO DE FUNCIONALIDADES SOBRE DADOS NORMALIZADOS NO *MONGODB*

ARACATI-CE
21 de março de 2021

Emanuel Moraes de Almeida

ALPHA RESTFUL: UM *FRAMEWORK* PARA OTIMIZAÇÕES NA IMPLEMENTAÇÃO DE FUNCIONALIDADES SOBRE DADOS NORMALIZADOS NO *MONGODB*

Trabalho de Conclusão de Curso (TCC) apresentado ao curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia do Ceará - IFCE - Campus Aracati, como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Orientador: Me. Diego Rocha Lima
Coorientador: Esp. Thiago Felippe de Lima

Aracati-CE
21 de março de 2021

Dados Internacionais de Catalogação na Publicação
Instituto Federal do Ceará - IFCE
Sistema de Bibliotecas - SIBI
Ficha catalográfica elaborada pelo SIBI/IFCE, com os dados fornecidos pelo(a) autor(a)

A447a Almeida, Emanuel Moraes de.
Alpha Restful: Um Framework para Otimizações na Implementação de Funcionalidades Sobre Dados Normalizados no MongoDB / Emanuel Moraes de Almeida. - 2020.
74 f.

Trabalho de Conclusão de Curso (graduação) - Instituto Federal do Ceará, Bacharelado em Ciência da Computação, Campus Aracati, 2020.
Orientação: Prof. Me. Diego Rocha Lima.
Coorientação: Prof. Esp. Thiago Felippe de Lima.

1. MongoDB. 2. Normalização. 3. Framework. I. Titulo.

Emanuel Moraes de Almeida

ALPHA RESTFUL: UM *FRAMEWORK* PARA OTIMIZAÇÕES NA IMPLEMENTAÇÃO DE FUNCIONALIDADES SOBRE DADOS NORMALIZADOS NO *MONGODB*

Trabalho de Conclusão de Curso (TCC) apresentado ao curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia do Ceará - IFCE - Campus Aracati, como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Aprovada em <data>

BANCA EXAMINADORA

Prof. Me. Diego Rocha Lima (Orientador)
Instituto Federal de Educação, Ciência e Tecnologia do Ceará

Prof. Esp. Thiago Felippe de Lima Bandeira (Coorientador)
Instituto Federal de Educação, Ciência e Tecnologia do Ceará

Prof. Me. Érica de Lima Gallindo
Instituto Federal de Educação, Ciência e Tecnologia do Ceará

Prof. Me. Silas Santiago Lopes Pereira
Instituto Federal de Educação, Ciência e Tecnologia do Ceará

RESUMO

Com a crescente demanda para armazenar uma quantidade cada vez maior de dados, os modelos de bancos de dados tradicionais vêm demonstrando dificuldades no desenvolvimento de aplicações escaláveis, disponíveis e consistentes. Por isso, o modelo NoSQL tornou-se uma tecnologia emergente para suprir as deficiências do modelo relacional. O *MongoDB*, que está entre os bancos de dados NoSQL mais populares, apesar de facilitar o armazenamento de dados desnormalizados, muitas vezes não possui recursos simples e intuitivos para a manipulação de dados normalizados. Neste contexto, este trabalho apresenta um *framework*, denominado *Alpha Restful*, que propõe facilitar o desenvolvimento de funcionalidades sobre dados normalizados utilizando o *MongoDB*. Como resultado, através de exemplos de código fonte, demonstrou-se de que forma o *Alpha Restful* se sobressai em relação às alternativas apresentadas, comprovando a relevância deste estudo.

Palavras-chave: *MongoDB*; Normalização; *Framework*.

ABSTRACT

With the growing demand for storing an increasing amount of data, traditional database models have shown difficulties in developing scalable, available and consistent applications. For this reason, the NoSQL model has become an emerging technology to supply deficiencies in the relational model. The MongoDB, which is among the most popular NoSQL databases, despite facilitating the storage of unnormalized data, it often does not have simple and intuitive resources for handling normalized data. In this context, this work presents a framework, called Alpha Restful, which proposes to facilitate the development of features on normalized data using MongoDB. As a result, through examples of source code, it was demonstrated how Alpha Restful stands out in relation to the alternatives presented, proving the relevance of this study.

Key words: MongoDB; Normalization; *Framework*.

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE EXEMPLOS

LISTA DE ABREVIATURAS E SIGLAS

SQL	*Structured Query Language*
NoSQL	*Not Only SQL*
CRUD	*Create, Read, Update e Delete*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
BJSON	*Binary JSON*

SUMÁRIO

1 INTRODUÇÃO

A grande demanda pela informatização de processos, que antes eram desenvolvidos completamente por humanos e com pouco auxílio de recursos tecnológicos, tem sido bastante crescente. Isto vem estimulando a criação de novas ferramentas para facilitar o desenvolvimento de aplicações que automatizam tais processos. Dentre as características mais comuns no desenvolvimento das mais diversas aplicações, está a necessidade de armazenar, buscar, atualizar e remover dados, popularmente conhecido como CRUD (*Create*, *Read*, *Update* e *Delete*). Dependendo do propósito da aplicação desenvolvida, todas ou algumas dessas operações são utilizadas. Para facilitar e otimizar tais operações, diversos bancos de dados foram desenvolvidos.

Assim como descrito por Date (2004), um banco de dados é “um sistema de armazenamento de dados baseado em computador; isto é, um sistema cujo objetivo global é registrar e manter informação”. Assim como demonstrado por Davoudian, Chen e Liu (2018), vários tipos diferentes de Banco de Dados foram criados ao longo do tempo. Dentre eles se destaca o modelo relacional, por se tratar de um modelo tradicional. Este define uma estrutura de dados normalizada baseada em tabelas. A linguagem mais comum utilizada por tais bancos é o SQL (*Structured Query Language*). Tal modelo relacional foi desenvolvido visando a imposição de alguns limites, definidos pelas formas normais. Tais limites normalizam os dados para garantir que, por exemplo, duas ou mais linhas de uma tabela nunca possam representar o mesmo objeto no sistema, ao mesmo tempo que garante que nenhum identificador (chave primária) represente parcialmente a tabela. As formas normais auxiliam em operações como a otimização da quantidade de dados armazenados; otimização da atualização de registros; e criação de regras na própria estrutura de armazenamento, afim de dificultar a inconsistência de dados.

Arboleda, Rendón e Vásquez (2016) descrevem que existe uma enorme demanda para armazenar uma quantidade cada vez maior de dados. Neste cenário, Davoudian, Chen e Liu (2018) observam que os limites impostos pelas formas normais dificultam a criação de aplicações escaláveis, disponíveis e consistentes. Isso ocorre porque a normalização dificulta o agrupamento de dados relevantes em um único servidor, dificultando a escalabilidade horizontal em servidores distribuídos. Esta característica exige a junção de diversos dados que podem estar fisicamente distantes. Além disso, bancos com uma quantidade muito grande de dados sofrem perdas de performance ao tentar unir tais dados.

Baseado nessa problemática, começaram a surgir novos modelos de armazenamento de dados não relacionai. Estes, objetivam otimizar o armazenamento de

dados, cujo o atendimento de suas exigências fosse muito caro, complexo ou inviável para o modelo de banco de dados relacional. Tais modelos ficaram conhecidos como NoSQL.

Atualmente, um dos bancos de dados NoSQL mais populares é o *MongoDB*. Tal banco estrutura seus dados baseado em documentos. Esta abordagem quebra várias barreiras limitadas pelo modelo relacional, permitindo que os dados sejam armazenados de maneira não normalizada (desnormalização).

O *MongoDB* permite uma maior flexibilização da estrutura de armazenamento. Isso facilita o armazenamento de dados semi-estruturados e não estruturados, pois tal banco não exige a definição de uma estrutura de armazenamento pré-estabelecida (*schema*), facilitando o armazenamento dados difíceis de se estabelecer um padrão. Além disso, a desnormalização gera um ganho de performance em diversas operações para diversos tipos de aplicações, sendo o IoT (*Internet of Things*) um dos grandes beneficiários por, geralmente, gerenciar grandes volumes de dados. Assim como descrito por (DAVOUDIAN; CHEN; LIU, 2018), o armazenamento de dados relevantes em um único local otimiza a escalabilidade horizontal da aplicação, além de evitar a junção dos dados, pois os dados já serão armazenados juntos. Apesar dos benefícios da desnormalização, existem possíveis problemas que podem decorrer mediante seu uso.

Os limites impostos pela normalização tendem a garantir que um determinado valor somente precisará ser alterado em um único lugar. Por outro lado, um banco desnormalizado não garante que um determinado valor estará em um único lugar, podendo exigir buscas possivelmente lentas para a localização de todos os locais onde o dado deve ser atualizado. Por causa de tal problemática, bancos que não seguem o modelo relacional, podem sentir a necessidade de mesclar estratégias de normalização e desnormalização.

O *MongoDB* foi projetado para que os dados possam estar desnormalizados, mas ele, atualmente, não ofereceferramentas de desenvolvimento tão fáceis e intuitivas para gerenciar dados normalizados, quanto aquelas disponibilizadas para dados desnormalizados. Um exemplo disto pode ser observado na operação de junção de documentos descrito pelo trabalho de Celesti, Fazio e Villari (2019). Nele, a junção de documentos no *MongoDB*, por meio de uma operação equivalente ao *inner join* do SQL, tornou-se, não só mais complexo de ser desenvolvido, como apresentou performance inferior em comparação com um banco relacional. Mediante tal cenário, pode-se afirmar que existem operações em dados normalizados que seriam mais simples de serem realizadas usando a linguagem SQL.

O desenvolvimento de uma aplicação com *MongoDB* pode exigir um trabalho extra, em comparação a bancos relacionais, pois, as vezes, será necessário realizar

junções e buscas em dados normalizados de forma não tão intuitiva quanto o SQL. Além disso, por causa da alta flexibilidade na estruturação dos dados, as diferentes formas como os dados podem ser armazenados exigem tratamentos distintos na implementação de diversas funcionalidades. Dessa forma, se a estrutura dos dados mudar durante o desenvolvimento, as operações de busca e junção de documentos precisarão ser alteradas. Ou seja, haverão mudanças em todas as funcionalidades que estiverem referenciando os dados reestruturados.

Visando a resolução de tais problemas, este trabalho apresenta o desenvolvimento de um *framework* denominado de *Alpha Restful*, criado para desenvolver aplicações com *MongoDB* na linguagem *ECMAScript* 6 (*JavaScript*) e *NodeJS* (no mínimo na versão 8).

O objetivo geral deste trabalho é apresentar o *Alpha Restful* como uma solução capaz de abstrair a implementação de diversas funcionalidades sobre dados normalizados no *MongoDB*. Neste contexto, pretende-se comparar algumas soluções de mercado com o *framework* proposto, através da implementação de algumas funcionalidades. Além disso, objetiva-se mostrar como o *Alpha Restful* se sobressai, em alguns requisitos, em comparação às outras ferramentas utilizadas durante os testes com o *framework* proposto.

Dentre todas as funcionalidades do *framework*, as apresentadas nesse trabalho (seções 5.1, 5.2, 5.3, 5.4 e 5.5) são muito importantes para a manipulação de dados, usando o *MongoDB*. Outras funcionalidades também foram implementadas, porém seu escopo de atuação está mais intimamente relacionado à web (desenvolvimento da comunicação de aplicações dentro de uma rede local ou global de computadores). Como o escopo desse trabalho é fazer uma análise mais próxima do *MongoDB*, tais funcionalidades foram omitidas nesse documento. Apesar disso, quase todas as funcionalidades desenvolvidas estão disponíveis em um guia de uso gratuito e online[1]. Por meio do link desse guia, é possível baixar a ferramenta e utilizá-la. Um dos resultados relevantes que pode ser encontrado nesse link é de que até o dia 24/03/2020, mais de 6000 *downloads* foram feitos, desde o seu lançamento na internet (24/02/2019).

Os capítulos seguintes contém mais detalhes do trabalho. No capítulo 3 temos os trabalhos relacionados. Nele são apresentados trabalhos que contribuem e corroboram para as afirmativas e justificativas aqui apresentadas. No capítulo 2 é apresentado a fundamentação teórica do trabalho, mostrando as informações técnicas necessárias para compreender o modelo tradicional relacional e o modelo baseado em documentos proposto pelo *MongoDB*. Nele será possível compreender, com mais detalhes, as problemáticas envolvendo ambos os modelos. No capítulo 4 está

[1] Disponível pelo link `https://www.npmjs.com/package/alpha-restful`

presente a metodologia do trabalho, demonstrando como a proposta foi desenvolvida e explanada como solução para os problemas apresentados. No capítulo 5 é demonstrado, através do uso de exemplos práticos, os benefícios do *Alpha Restful*. Para isto, serão exploradas as dificuldades na implementação de 5 funcionalidades selecionadas previamente, comparando o *Alpha Restful* com outras ferramentas já existentes. Tal comparativo evidencia os benefícios adquiridos pelo uso do *framework* proposto. Por fim, no capítulo 6 encontra-se as conclusões finais deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Modelo de Banco de Dados Relacional

O modelo relacional surgiu por volta dos anos de 1970, com base no modelo proposto por Codd (1970). Através deste modelo, “os dados são armazenados com um forte grau de independência, desacoplando a lógica, da representação dos dados” (DAVOUDIAN; CHEN; LIU, 2018). Na representação relacional, os dados são normalizados e as diversas relações (tabelas) podem referenciar os dados contidos em outras tabelas.

Assim como explicado por Date (2004), cada linha de uma tabela (tupla) deve representar a abstração de um objeto do sistema, sendo que cada célula é uma característica (atributo) do objeto representado. Cada tupla deve possuir pelo menos uma célula como identificador único (chave primária) que irá representar todos os dados contido na linha a qual ela está contida. A chave primária é um valor único para a tabela, geralmente representado com um tipo numérico. Tal chave pode ser armazenada em outras tabelas por meio do uso de chaves estrangeiras. Uma chave estrangeira é uma (ou mais de uma) chave primária de outra relação. Armazenar tal chave simboliza que os dados representados por ela estão contidos dentro da tabela.

Como exemplo, pode-se abstrair um sistema que precisa armazenar várias pessoas (na qual cada uma possui um nome e uma idade) e várias casas (que possui uma rua e número). A tabela 1 apresenta como as pessoas podem estar representadas no banco de dados.

Tabela 1 – Exemplo de relação de pessoas

Chave Primária	Nome	Idade
1	Emanuel	21
2	Eduardo	40

Continuando com o exemplo, naturalmente, no desenvolvimento de um sistema, as entidades se relacionam entre si. Neste caso, uma pessoa pode possuir várias casas, mas uma casa possui apenas uma pessoa como dono. Para representar essa relação, precisa-se definir quem será o dono da relação, ou seja, quem irá receber a chave primária da outra entidade através de uma chave estrangeira. Sempre deverá existir apenas um dono da relação, ou seja, a chave estrangeira desta relação deverá estar em apenas uma tabela.

Como uma pessoa pode possuir várias casas, o dono da relação deve ser a casa, caso contrário a tupla de uma pessoa teria que ser duplicada (a menos que uma terceira tabela seja criada). Para representar as casas neste contexto, pode-se modelar os dados assim como demonstrado na tabela 2. Nela, pode-se observar que a pessoa Emanuel possui as duas casas armazenados no banco de dados, enquanto o Eduardo não possui nenhuma casa.

Tabela 2 – Exemplo de relação de casas

Chave Primária	Rua	Número	Chave Estrangeira de Pessoa
10	Rua Castelo Branco	1B	1
20	Rua Pompel	1089	1

2.1.1 Normalização

O modelo relacional possui algumas regras que o desenvolvedor observa ao modelar as estruturas das relações de seu banco. Dentre estas regras estão contidas as regras de normalização. “O objetivo da normalização é evitar os problemas de banco de dados, bem como eliminar a ‘mistura de assuntos’ e as correspondentes redundâncias desnecessárias de dados” (MACHADO, 2020). Com a redução dessas “redundâncias desnecessárias”, uma otimização da quantidade de dados é alcançada. Machado (2020) descreve 5 formas normais. Destas, apenas as 3 primeiras serão explicitadas a seguir, por terem impacto mais significante para os propósitos deste trabalho.

2.1.1.1 Primeira Forma Normal

Como visto anteriormente, uma tupla representa um objeto da entidade representada pela relação a qual a tupla pertence. Tendo esse princípio como base, como é possível representar a situação de uma casa possuir vários donos e ao mesmo tempo uma pessoa possuir várias casas?

O princípio da primeira forma normal define que nunca deve-se duplicar colunas em várias linhas, garantindo que duas ou mais tuplas nunca representem o mesmo objeto. Para manter este princípio e ainda realizar um relacionamento de muitos para muitos (uma pessoa tem muitas casas e uma casa tem muitas pessoas), torna-se necessário a criação de uma tabela auxiliar para representar o relacionamento de pessoa com casa. A tabela 3 exemplifica a criação de uma tabela auxiliar, na qual a pessoa “Emanuel” possui as duas casas e o “Eduardo” possui a casa de número 1B.

Tabela 3 – Exemplo de tabela auxiliar

Chave Primária	Chave Estrangeira de Pessoa	Chave Estrangeira de Casa
100	1	10
200	1	20
300	2	10

2.1.1.2 Segunda Forma Normal

A segunda forma normal especifica que todos os dados de uma tupla devem ser representados por toda a sua chave primária e jamais por apenas parte dela. Como exemplo disto pode-se afirmar que uma casa jamais poderia ser armazenado dentro da mesma tupla de uma pessoa, mesmo em um relacionamento um para um (Uma pessoa para uma casa e uma casa para uma pessoa). Isto ocorre pois os dados de uma casa são independentes dos dados de uma pessoa e a chave primária de uma pessoa não poderia representar também os dados de uma casa.

Uma chave primária também pode ser composta por várias células, mas todos os dados da tupla devem depender de todas as células da chave primária. Caso exista algum dado que dependa apenas de parte da chave primária, uma nova tabela deve ser criada. Uma tabela somente está na segunda forma normal se também estiver na primeira forma normal.

2.1.1.3 Terceira Forma Normal

Continuando o exemplo, pode-se imaginar que devem existir atributos no relacionamento entre pessoa e casa, indicando o valor mensal que a pessoa deve pagar como aluguel pela casa, a quantidade de meses a pagar o aluguel e o total a pagar até o fim do contrato. Para isso, poderíamos adicionar tais atributos na tabela de relacionamento de pessoa com casa, assim como descrito na tabela 4. Nesta, a pessoa de nome “Emanuel” paga 200 reais pela casa de número 1B com contrato válido por 12 meses e paga 400 reais pela casa de número 1089 com contrato válido por 24 meses. O Eduardo paga 300 reais pela casa de número 1B com contrato válido por 8 meses.

Tabela 4 – Exemplo Atributos no Relacionamento

Chave Primária	Pessoa	Casa	Valor Mensal	Quantidade de Meses	Total
100	1	10	200	12	2400
200	1	20	400	24	9600
300	2	10	300	8	2400

Para este caso, a relação 4 não está na terceira forma normal, pois o atributo *Total* é dependente do *Valor Mensal* e da *Quantidade de Meses*. O *Total* pode ser obtido através de uma busca no banco de dados, realizando uma simples multiplicação do *Valor Mensal* e da *Quantidade de Meses*. Para que esta relação fique de acordo com a Terceira Forma Normal é necessário remover a última coluna, assim como apresentado na tabela 5.

Tabela 5 – Relacionamento sem o "Total"

Chave Primária	Pessoa	Casa	Valor Mensal	Quantidade de Meses
100	1	10	200	12
200	1	20	400	24
300	2	10	300	8

A terceira forma normal especifica que não deve existir uma célula que dependa de outras células. Cada valor deverá ser o mais independente possível dos outros valores. Para que uma relação esteja na terceira forma normal, também torna-se necessário que esteja na segunda forma normal.

2.1.2 Junção de Tabelas

Assim como descrito por Machado (2020), existem situações onde as tabelas precisam ser unidas em uma única tabela, para que os dados contidos em diversas relações, que são necessários à consulta formulada, possam ser acessados. Como exemplo disto, pode-se unir as tabelas 1, 2 e 5, para a pessoa de nome *Emanuel*. No exemplo 1 encontra-se uma codificação SQL para a união de tabelas para o contexto apresentado. A seguir, na tabela 6 apresenta-se o resultado de tal codificação.

Exemplo 1 – União de Tabelas com SQL

```
01 |    SELECT p.id as Pessoa, p.nome as Nome, p.idade as Idade, rpc.
   |   valor_mensal as ValorMensal, rpc.quantidade_meses as Meses, c.id as Casa
   |   , c.rua as Rua, c.numero as Numero
02 |    FROM pessoa p JOIN
03 |        Relacionamentopessoacasa rpc ON p.id = rpc.pessoa_id JOIN
04 |        casa c ON rpc.casa_id = c.id
05 |    WHERE p.nome = "Emanuel"
```

Tabela 6 – Junção de Tabelas Para Emanuel

Pessoa	Nome	Idade	ValorMensal	Meses	Casa	Rua	Numero
1	Emanuel	21	200	12	10	Rua Castelo Branco	1B
1	Emanuel	21	400	24	20	Rua Pompel	1089

Quando há a separação dos dados, a atualização destes tendem a garantir uma maior consistência, além de tentar garantir que os valores precisarão ser atualizados em um único local. Mas em buscas no banco de dados, as junções de tabelas são, muitas vezes, inevitáveis.

Em um sistema complexo, com uma grande quantidade de dados e relações, cada tabela poderá conter milhares ou até milhões de linhas. Uma operação de junção, envolvendo muitos dados, pode ser lento demais para os requisitos da aplicação. Existem algumas técnicas e estratégias para deixar essas consultas mais rápidas, como, por exemplo, o uso de índices. Mas mesmo com essas estratégias, haverá aplicações na qual a exigência de alta performance, escalabilidade, disponibilidade e consistência tornará inviável ou altamente caro escalar uma aplicação utilizando o modelo relacional.

2.1.3 Vantagens do Modelo relacional

- O modelo relacional já está bem solidificado no mercado, possuindo várias ferramentas e *frameworks* para facilitar o desenvolvimento dos mais diversos tipos de aplicações;
- Pelo fato do modelo relacional forçar a separação e normalização dos dados, torna-se mais difícil que erros humanos ou de desenvolvimento venham a fazer os dados perderem sua consistência;
- A normalização, por causa de seu princípio de não repetir dados em várias tabelas, tende a diminuir o espaço necessário para armazenar os dados, além de a atualização de registros, no geral, ocorrer em apenas um local.

2.1.4 Desvantagens do Modelo relacional

- Os recursos de modelagem são muito limitados;
- A construção de objetos por meio de operações de junção, muitas vezes, são caras;

- O modelo relacional revelou deficiências no armazenamento e consulta de um grande volume de dados;
- Diversas aplicações com altos requisitos de escalabilidade, disponibilidade e consistência se tornam caras ou inviáveis no modelo relacional;
- As aplicações precisam se adaptar ao modelo relacional. Mesmo que uma aplicação possa ter uma parte dos dados não normalizados, sem prejudicar a consistência (por causa de alguma regra de negócio), o modelo relacional exige a normalização em vários casos, para um bom funcionamento das consultas.

2.2 Modelo de Banco de Dados NoSQL com *MongoDB*

Assim como descrito por Davoudian, Chen e Liu (2018), o uso do modelo de banco de dados relacional tornam os requisitos de alta disponibilidade, baixa tolerância a falhas para responder aos clientes, confiabilidade das transações, suporte a dados altamente consistentes e manutenção de *schemas* com baixo custo de evolução, muito difíceis ou inatingíveis. Além dos problemas apresentados, Davoudian, Chen e Liu (2018) ainda descrevem que os sistemas baseados em modelos relacionais trazem uma maior complexidade e sobrecarga para unir dados distribuídos normalizados. Por causa de tais problemas, houve uma demanda por um novo modelo de banco de dados. O modelo que tornou-se uma tendência emergente para o armazenamento de dados não relacional é o NoSQL, que foi projetado para satisfazer os requisitos de alta disponibilidade e escalabilidade de aplicações de âmbito global.

De acordo com Boaglio (2015), os bancos que estruturam seus dados usando abordagens não relacionais são denominados de NoSQL (*Not Only SQL*). Ainda de acordo com ele, tais bancos normalmente são baseados em documento, orientado a objetos, chave-valor e grafos. Para os propósitos deste trabalho, esta seção utiliza como base a descrição e utilização de modelos de banco de dados baseado em documento, sendo exemplificado pela descrição e uso do banco de dados *MongoDB*.

Ø*MongoDB* armazena os dados de forma não estruturada, ou seja, nenhum *schema* é necessário para que os dados sejam armazenados. Isto significa que cada documento no *MongoDB* podem ter uma estrutura completamente diferente uma das outras. Esta abordagem permite que dados que não possuem um padrão pré estabelecido possam simplesmente ser armazenado, sem que estes precisem, necessariamente, serem adaptados para um padrão definido anteriormente. Essa característica é bastante útil para diversas aplicações porém, tamanha liberdade pode ser prejudicial em certos casos, onde a aplicação exija regras para os tipos de dados armazenados. Neste contexto surgem diversas bibliotecas que permitem, quando necessário, a cria-

ção de *schemas*, ou seja, estruturas com regras na qual os dados precisam obedecer para serem armazenados. Dentre essas bibliotecas, o *Mongoose* destaca-se, por ser bastante citado na literatura e por ser muito popular no desenvolvimento de aplicações com *MongoDB* no *Node JS*.

Enquanto que no modelo relacional o armazenamento ocorre em uma tabela, no MongoDB o armazenamento ocorre em uma coleção de documentos, sendo cada documento a representação de um objeto do sistema. Cada banco de dados NoSQL baseado em documento terá a sua própria estrutura de documento. No caso específico do *MongoDB*, os documentos são escritos de maneira estruturada, seguindo uma variação do formato JSON (*JavaScript Object Notation*). Tal variação possui o nome de BSON (*Binary* JSON).

2.2.1 O Formato JSON

Assim como descrito por Boaglio (2015), O JSON é um formato de representação de dados na qual um objeto possui uma lista de atributos. Um atributo é o nome de uma característica pertencente ao objeto na qual o JSON representa. Cada atributo possui um valor associado a ele. Tal valor pode ser *null* (valor vazio), *true* (verdadeiro), *false* (falso), textual (entre aspas), numérico (sem aspas), pode ser um outro JSON (definido entre chaves) e pode ser uma lista de qualquer um dos tipos definidos anteriormente (entre colchetes separados por vírgula).

No exemplo 2 é apresentado uma representação de uma lista de pessoas com suas respectivas casas. Nele, observa-se que a representação de um objeto é dada entre chaves (*{}*) e, dentro dessas chaves, são definidos os nomes dos atributos do objeto (entre aspas) e, após os dois pontos (*:*), é definido o valor para aquele atributo. Quando se deseja que o valor seja uma lista de elementos, envolve-se todos os elemento da lista entre colchetes (*[]*) e cada elemento é separado por vírgula (*,*).

Exemplo 2 – JSON Representando Várias Pessoas Com Suas casas

```
1  [{
2      "nome": "Emanuel",
3      "idade": 21,
4      "casas": [
5          {
6              "rua": "Rua Castelo Branco",
7              "numero": "1B",
8              "valorMensal": 200,
9              "quantidadeMeses": 12
10         },
```

```
        {
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},{
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

No exemplo 2, está sendo representado uma lista de objetos. Para a representação desta lista, as pessoas (JSON) são envolvidas entre colchetes. Caso seja desejado representar apenas uma pessoa (ao invés de uma lista), bastaria remover os colchetes mais externos e deixar apenas um JSON na raiz do documento. Observa-se que a pessoa com o nome *Eduardo* possui apenas uma casa, e por este motivo o valor do atributo *casa* pode ser representado como um JSON, mas também seria possível representar como uma lista de apenas uma casa. Para isto, bastaria, apenas, envolver as chaves (*{}*) em colchetes (*[]*), assim como ocorre para a pessoa *Emanuel*.

2.2.2 Coleções de Documentos no MongoDB

O formato de documento utilizado pelo *MongoDB* é uma variação do JSON: o BSON (*Binary JSON*). O BSON é uma extensão do JSON, suportando novos tipos de dados. De acordo com Boaglio (2015), os novos tipos de dados suportados são:

1. *MinKey*, *MaxKey*, *Timestamp* — Tipos utilizados internamente no *MongoDB*
2. *BinData* — *Array* de *bytes* para dados binários
3. *ObjectId* — Identificador único de um registro do *MongoDB*
4. Date — Representação de data

5. Expressões Regulares

Enquanto que no modelo relacional uma tabela representa uma coleção de objetos no sistema, no *MongoDB*, esta representação é feita por uma coleção de documentos. Da mesma forma, enquanto uma tupla de uma tabela representa, no modelo relacional, um objeto, no *MongoDB*, esta representação é feita por um documento, que possui, em seu interior, um BSON (não uma lista, mas apenas um único BSON).

Seguindo esta abordagem, para, por exemplo, armazenarmos várias pessoas no banco, precisa ser criado uma coleção que irá armazenar vários documentos, sendo cada documento uma representação de uma pessoa diferente. Cada pessoa terá um identificador único para a coleção (*_id*), que irá representar o documento. Através deste identificador, pode-se fazer relações entre BJSONs diferentes. Tal identificador pode ser de qualquer tipo, porém, existe um tipo de dados chamado de *ObjectId* criado especificamente para ser usado como um identificador único. O valor do *ObjectId* pode ser explicitamente definido pelo usuário. Caso o usuário não o forneça, o *MongoDB* irá defini-lo de forma automática. De acordo com Boaglio (2015), o valor gerado não é sequencial, pois ele é criado baseado no *timestamp*, o ID da máquina, o ID do processo e um contador local.

No exemplo 3 está sendo exemplificado um BSON que poderia ser armazenado em um documento do *MongoDB*, para representar a pessoa de nome “Emanuel”. Para ilustrar a flexibilidade do valor do *_id*, o identificador das entidades serão armazenados por meio de um valor numérico, porém os identificadores dos BSON “internos” serão definidos como um *ObjectId*. Esta escolha foi tomada, apenas, para facilitar a visualização dos exemplos, mas em um sistema, o *_id* poderia assumir qualquer tipo de dado.

Exemplo 3 – BJSON da pessoa *Emanuel*

```
{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            _id: ObjectId("5e66c969d5d7cc24c0850854"),
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
```

```
12          },
13          {
14              _id: ObjectId("5e66c9740efbf528b05a7537"),
15              "rua": "Rua Pompel",
16              "numero": 1089,
17              "valorMensal": 400,
18              "quantidadeMeses": 24
19          }
20      ]
21  }
```

O identificador (*_id*) deve existir pelo menos no BSON principal (neste caso, aquele que representa uma pessoa). O BSON que representa uma casa não necessariamente precisa de um identificador, mas, para este exemplo, optou-se por gerar um. O identificador do BSON de uma casa não é o identificador de outro documento, pois, neste exemplo os dados estão desnormalizados e, por enquanto, para o exemplo apresentado, não existe outra coleção além da coleção de pessoa.

2.2.3 Desnormalização

Assim como descrito por Date (2004), a desnormalização é o agrupamento de dados (que estariam separados em diferentes coleções, pelas formas normais) em uma única coleção. É um armazenamento, gerando redundância, sem a preocupação de seguir as formas normais, para unir dados relevantes em um único local. No exemplo apresentado, a coleção de casa não existe e as casas estão armazenadas dentro do mesmo BSON na qual uma pessoa é armazenada. Tal estrutura desobedece a segunda forma normal, pois o identificador do documento (*_id*) identifica uma pessoa, não podendo, portanto, identificar os dados das casas que estão presente no próprio documento.

Uma das principais vantagens da desnormalização é que, em vários casos, não existe a necessidade de realizar junção de vários documentos, podendo ter um ganho substancial de performance. No exemplo apresentado do modelo relacional (1), foi necessário realizar buscas e uniões de tabelas para obter uma relação contendo um balanço da lista total de casas da pessoa com nome *Emanuel*. No *MongoDB*, caso desejemos realizar a mesma pesquisa para a pessoa *Emanuel*, obtendo a lista de casas que ela possui, não haveria a necessidade de juntar quaisquer documentos. Para isto, bastaria, apenas, buscar o BJSON da pessoa a qual se deseja buscar e retornar o documento, tal qual como foi armazenado. No modelo relacional, como foi abordado anteriormente, uma tabela pode ter milhares ou milhões de linhas e pode

ser necessário unir centenas de tabelas. Com a desnormalização, nenhuma junção torna-se necessária para o exemplo abordado.

2.2.4 Problemas da Desnormalização

Apesar dos benefícios da desnormalização, ela nem sempre é aconselhável ou viável. Através da desnormalização, os dados começam a ser duplicados, e dependendo das regras da aplicação, essa duplicação pode ser problemática. Como exemplo disto, pode-se imaginar uma situação onde várias pessoas possuem uma mesma casa. Neste exemplo, os dados da mesma casa precisariam ser duplicados em diversas pessoas. Se uma casa nunca precisar ser removida ou atualizada, essa duplicação possivelmente não irá gerar nenhum problema. Mas caso uma casa precise ser removida ou ter, por exemplo o seu número atualizado, seria necessário atualizar a casa em todas as pessoas que a possuem. Essa operação poderia ser extremamente lenta e dependendo da quantidade de dados armazenados, isto seria uma operação inviável. A desnormalização deve ocorrer em situações onde as regras de negócio da aplicação garantem uma segurança na realização de tal procedimento.

Quando a desnormalização não for uma opção, torna-se necessário a criação de uma nova coleção para normalizar estes dados. Para exemplificar esta normalização, uma nova coleção deve ser criada para armazenar todas as casas. No atributo *casas* no documento de pessoa, apenas será armazenado o identificador da casa a qual ela está relacionada, junto com os atributos de relacionamento *valorMensal* e *quantidadeMeses*.

Nos exemplos 4 e 5 é demonstrado como a pessoa de nome *Eduardo* e a casa que ele possui, seriam armazenados, respectivamente, de forma normalizada. Observa-se que no atributo de *casas* existem dois identificadores. O identificador *_id* identifica o BSON do relacionamento de pessoa com casa. O identificador *id* é um atributo que poderia ter recebido outro nome. Como neste caso é necessário armazenar o identificador do documento de casa, escolhe-se um nome de atributo que será padronizado na aplicação para referenciar a casa a qual esta pessoa se relaciona. Neste exemplo, escolheu-se padronizar o atributo *id* como aquele que receberá o identificador do documento da casa que se relaciona com esta pessoa. Desta forma, o identificador da casa a qual esta pessoa pertence (20) é armazenado no atributo *id*.

Exemplo 4 – Estrutura de Dados Normalizados da pessoa *Eduardo*

```
1 {
2     "_id": 2,
3     "nome": "Eduardo",
4     "idade": 40,
```

```
    "casas": {
        "_id": ObjectId("5e66cad68a1e581e3cf47dfc"),
        "id": 20,
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}
```

Exemplo 5 – Estrutura de Dados Normalizados de uma casa

```
{
    "_id": 20,
    "rua": "Rua Pompel",
    "numero": "1089"
}
```

É possível padronizar para que o *_id* represente, ao mesmo tempo, o identificador do BSON (identificador do relacionamento) e o identificador da entidade relacionada. Nesse caso, o atributo *id* não seria necessário. Para as exemplificações desse trabalho, essa padronização não será feita, pois ter esses dois identificadores pode gerar algumas vantagens em consultas. Uma possível vantagem seria a de uma entidade poder se relacionar mais de uma vez com o mesmo documento, de mesmo identificador (*id*), porém o sistema poderia diferenciar os dois relacionamentos por meio do identificador da relação (*_id*). Além dessa vantagem, pode ser que alguma biblioteca gere o *_id* automaticamente e utilize esse valor gerado de alguma maneira (a biblioteca *Mongoose*, que será explicada mais a frente, gera esse identificador automaticamente, caso não seja atribuído). Dessa forma, dependendo da situação, é uma boa ideia manter esses dois identificadores.

Nesse exemplo de normalização, os valores de uma casa (*rua* e *numero*) não são armazenados no documento de pessoa. Desta forma, caso uma casa tenha seu número atualizado, o número será atualizado no documento da casa e não precisará ser atualizado em todos os documentos das pessoas que se relacionam com esta casa.

No exemplo 5, uma das possíveis regras que poderia ser exigida é de que a rua de uma casa nunca possa ser atualizada, porém o número possa ser atualizado. Neste caso, pode-se realizar uma desnormalização, apenas para o atributo *rua*, normalizando o atributo *numero* e duplicando nas várias pessoas a *rua*. Nesta situação, não haveria necessidade de unir documentos para buscas que se interessem apenas no valor da rua das casas de uma determinada pessoa. Dependendo da natureza da aplicação, pode haver situações onde a desnormalização pode ser utilizada, mesmo

com a atualização do mesmo valor em vários documentos. Isto poderia ocorrer caso as regras de negócio da aplicação garantam que os dados não serão duplicados o suficiente para trazer uma performance inaceitável.

2.2.5 Junção de Documentos

Quando se realiza uma normalização, poderá haver situações onde os documentos precisem ser unidos. No exemplo 6 é apresentado o resultado da junção dos documentos de pessoas com casas.

Exemplo 6 – Junção de Documentos Normalizados

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": 10,
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id": 20,
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "_id": ObjectId("5e66cb581766a2056c48145f"),
```

```
30          "id": 20,
31          "rua": "Rua Pompel",
32          "numero": "1089",
33          "valorMensal": 300,
34          "quantidadeMeses": 8
35      }]
36  }]
```

2.2.6 Vantagens do Banco de Dados MongoDB

- Os modelos de armazenamento de dados são muito flexíveis, sendo que cada documento pode ser estruturado de forma diferente uns dos outros;
- Pela flexibilidade da modelagem, existe uma menor complexidade e sobrecarga para unir dados distribuídos normalizados;
- Por causa da flexibilidade da modelagem, os dados podem ser estruturados adaptando-se à aplicação, ao invés de a aplicação ter que se adaptar a modelagem;
- Com o *MongoDB* é possível agrupar dados relevantes em um único documento, além de possibilitar a duplicação dos dados, podendo alcançando, assim, uma maior performance em consulta;
- Os dados podem ser mais facilmente replicados e particionados de forma horizontal em servidores locais e remotos.

2.2.7 Desvantagens do Banco de dados MongoDB

- Por se tratar de uma tecnologia emergente, não existem tantas ferramentas e *frameworks* para a utilização de banco de dados NoSQL como existem para a utilização de banco de dados Relacionais;
- Pode levar a atualizações dispendiosas em dados duplicados;
- Não trata automaticamente os relacionamentos manuais entre documentos, podendo deixar passar (caso o desenvolvedor não se atente) a identificadores que apontam para documentos que já não existe mais;
- As buscas e uniões que precisam ser realizadas em documentos separados e relacionados podem ser mais complexas do que as consultas realizadas no modelo relacional.

3 TRABALHOS RELACIONADOS

O *MongoDB* é um dos bancos de dados NoSQL mais populares e utilizados. Isso pode ser demonstrado pela grande quantidade de citações a ele na literatura, além da pesquisa feita pela *DB-Engines*[1] citá-lo como sendo o banco de dados NoSQL mais popular em novembro de 2020. Vários são os trabalhos que o comparam a outros bancos de dados relacionais ou não relacionais. Os trabalhos de Rautmare e Bhalerao (2016); Seo, Lee e Lee (2017); Grover e Johari (2016); Fatima e Wasnik (2016); e Ansari (2018) fazem uma comparação de desempenho do *MongoDB* com o banco de dados relacional *MySql*. Neles são gerados gráficos e tabelas que medem o tempo de resposta de cada banco de dados, pela quantidade de dados teste usados em operações de leitura e escrita. Tais resultados demonstram um ganho significativo de performance do *MongoDB* em vários casos de uso. Isto demonstra a relevância da ferramenta, principalmente em aplicações com grande quantidade de dados (*Big Data*). Dentre os trabalhos apresentados, Fatima e Wasnik (2016) realizam testes com dados reais, visando simular um ambiente IoT, na qual existe a captura de uma grande quantidade de dados de sensores. Os testes realizados também fazem um comparativo com o banco de dados *VoltBD*, que utiliza um novo modelo de banco de dados chamado de *NewSQL*.

Além dos trabalhos apresentados, também pode ser citado o trabalho de Pandey (2020), que realiza um comparativo mais completo que os apresentados anteriormente, analisando diversos casos de teste diferentes. Tais testes também foram realizados sobre o *MongoDB* e o *MySql*, porém medindo o tempo de resposta por quantidade de *threads* em execução, *throughput* (operações/segundo) e latência média, em operações de leitura e escrita. Os resultados também demonstram uma vantagem do *MongoDB* em vários cenários. Dentre os trabalhos apresentados anteriormente, comparativos que envolvem cenários com várias *threads* ou a realização de operações simultâneas e distintas também podem ser encontrados em Rautmare e Bhalerao (2016); Fatima e Wasnik (2016); e Ansari (2018).

Outros trabalhos que realizam um comparativo com outros bancos de dados podem ser encontrados. Arboleda, Rendón e Vásquez (2016) fazem um comparativo com o banco de dados *Oracle*. Já Chopade e Dhavase (2017) realizam um comparativo com outro banco de dados NoSQL chamado de *CouchBase*, utilizando dados de imagens como teste.

[1] A pesquisa e a metodologia da pesquisa estão respectivamente disponíveis pelos *links* `https://db-engines.com/` e `https://db-engines.com/en/ranking_definition`. Acessos ocorridos no dia 16 de dezembro de 2020, às 16:18.

As afirmativas feitas na introdução deste trabalho sobre a dificuldade na implementação de operações sobre dados normalizados no *MongoDB* em comparação com o SQL, são confirmadas pelo trabalho desenvolvido por Celesti, Fazio e Villari (2019). Neste trabalho, implementações manual da junção de documentos, são realizadas usando o *MongoDB*. Nele, a junção de documentos no *MongoDB*, por meio de uma operação equivalente ao *inner join* do SQL, tornou-se, não só mais complexo de ser desenvolvido, como apresentou performance inferior em comparação com um banco relacional. Tal resultado demonstra a necessidade e importância de *APIs* e *frameworks* tratarem tal operação, a fim de se obter ganhos de performance e facilidade no desenvolvimento. Na seção 5.1, tal operação de junção será explicitada e alternativas à implementação manual feita por Celesti, Fazio e Villari (2019) serão demonstradas, por meio do uso de ferramentas externas ao *MongoDB*.

O *survey* de Davoudian, Chen e Liu (2018), assim como os trabalhos apresentados, confirmam as afirmativas deste trabalho da importância do NoSQL na aplicação e desenvolvimento de diversas aplicações, cujo o uso do modelo tradicional Relacional deixa a desejar. Em Davoudian, Chen e Liu (2018) podem ser encontradas mais informações e detalhes sobre o uso de bancos de *NoSQL*, principalmente no escopo de aplicações distribuídas e escaláveis de forma horizontal.

O *Mongoose*, sendo uma biblioteca que permite a criação de *schemas* no *MongoDB*, foi escolhido para ser alvo de comparação neste trabalho por causa de sua grande relevância, tando na literatura, quanto na sua popularidade no desenvolvimento de aplicações com *MongoDB* em *Node JS*. O quadro 1 referencia vários livros que citam o *Mongoose*, demonstrando sua relevância para o desenvolvimento de aplicações com *MongoDB*.

Quadro 1 – Livros que citam o *Mongoose*

Título	**Autores**
MERN Quick Start Guide	Wilson (2018)
Web development with MongoDB and NodeJs	Satheesh, D'mello e Krol (2015)
Building Node. js REST API with TDD Approach	Pandian (2018)
Full Stack JavaScript	Mardan (2018)
Mongoose for Application Development	Holmes (2013)
Node. js, MongoDB, and AngularJS web development	Dayley (2014)
JavaScript Frameworks for Modern Web Development	Uzayr, Cloud e Ambler (2019)

4 METODOLOGIA

Este trabalho apresenta uma pesquisa de abordagem qualitativa, de natureza aplicada, visando amenizar os problemas de busca, junção e remoção em documentos normalizados, no banco de dados *MongoDB*. Para isso, foi desenvolvido um *framework* denominado de *Alpha Restful*, que foi projetado para a linguagem *JavaScript*, para o ambiente de execução *Node JS*.

O *Alpha Restful* facilita e automatiza diversas funcionalidades para o desenvolvimento de aplicações com *MongoDB*. Dentre as funcionalidades disponíveis nesse *framework*, apenas cinco foram selecionadas e analisadas neste trabalho, pois tais funcionalidades melhoram o processo de desenvolvimento de aplicações, através da manipulação de dados normalizados. As 5 funcionalidades selecionadas são: Junção de Documentos; Buscas Sobre Documentos Normalizados; Remoção em Cascata de Documentos; Relação de Dependência Entre os Documentos; e Remoção de Identificadores Inválidos.

O estudo avalia, principalmente, a facilidade de desenvolver as cinco funcionalidades selecionadas, com o uso de algumas soluções de mercado, em comparação com o *framework* apresentado. Também é avaliado os recursos disponíveis nas ferramentas de mercado em comparação com o *Alpha Restful*. Além da avaliação qualitativa, foi obtido um resultado quantitativo (para a funcionalidade Buscas Sobre Documentos Normalizados), onde pode ser observada uma redução na quantidade de linhas de código, com a utilização da solução proposta.

Foram utilizados procedimentos experimentais para a análise de cada funcionalidade. Para tal, codificações em *JavaScript* foram desenvolvidas, a fim de extrair o quão fácil e intuitiva as implementações são. Além disso, pôde-se observar as particularidades de cada código, em relação aos conhecimentos técnicos necessários, sobre os comandos disponíveis na solução apresentada.

Após a apresentação dos resultados alcançados pelos procedimentos realizados, a pesquisa objetiva caracterizar o *Alpha Restful*, observando como ele apresenta melhorias para as situações apresentadas pelas outras soluções comparadas. Além disso, objetiva-se demonstrar os recursos disponíveis no *Alpha Restful*, destacando suas vantagens e utilidade.

Na prática, o *Alpha Restful* funciona como uma camada acima do *Mongoose*. Isto significa que toda a manipulação do banco é realizado pelo *Mongoose*. O *Alpha Restful* consegue abstrair diversas funcionalidades e, internamente, gerar diversos códigos fonte, graças ao objeto de sincronização (construído pelo programador) e os

metadados gerados pelo *framework*. A utilização do objeto de sincronização (*sync*) e dos metadados (*syncronized*) é apelidado de “sincronização”.

A sincronização é o estabelecimento de uma conexão lógica entre um atributo de uma entidade com outra entidade. Uma das formas de se estabelecer essa conexão é através do armazenamento de identificadores de outros documentos. Isso é equivalente ao uso de *foreign key* no modelo relacional. Uma sincronização pode ocorrer, tanto na modelagem das entidades, quanto de forma dinâmica, depois de uma pesquisa.

Uma vez que duas entidades estão sincronizadas, diversas opções podem ser definidas no objeto “*sync*”. Cada uma delas aciona eventos que esperam uma das duas entidades serem chamadas por algum método presente no *framework*. Tanto atributos reais (presentes no banco de dados) quanto atributos dinâmicos (não presentes no banco de dados, pois são criados dinamicamente em memória) podem ser alvo de uma sincronização. Um exemplo de atributo dinâmico será demonstrado na seção 5.1.5.1, quando for explicado o funcionamento do relacionamento inverso. Esses atributos dinâmicos são tratados como se eles existissem dentro do banco de dados, sendo possível utilizá-los em pesquisas. Esse tipo de atributo pode ser definido, tanto na modelagem da entidade, quanto dinamicamente na junção de documentos.

No objeto “*sync*” está presente as informações passadas pelo programador, sobre as funcionalidades a serem aplicadas pelo *framework*. Quando o método de busca é chamado, o *Alpha Restful* percorre os objetos de sincronização, à medida que cada atributo de cada entidade é analisado. A pesquisa ocorre de maneira recursiva. Para tal, é realizado uma separação da pesquisa por ponto (“.”). Cada nome de atributo, obtido por meio dessa separação, gera um código de pesquisa que é utilizado na recurção anterior. O *framework* utiliza as informações presentes no “*sync*” para saber onde se encontram os outros atributos na entidade relacionada. A lógica dessa implementação, gerada internamente, é a mesma contida no exemplo 47 do capítulo de resultados (5).

A junção de documentos também ocorre de forma recursiva. Para tal, os atributos contidos nos objetos de sincronização são percorridos. Para cada atributo, a opção “*fill*” é analisada, para saber se a entidade referente a esse atributo deve ser buscada e preenchida no atributo analisado. A recursão acaba, dependendo das opções disponíveis no objeto de sincronização, ou se não houver mais atributos a serem analisados.

Para as funcionalidades referentes à remoção de entidades, o *framework* utiliza os metadados contidos no objeto “*syncronized*”. Tal objeto é gerado automaticamente pelo *framework*, baseado nos objetos de sincronização das entidades que se relacionam com a entidade analisada. Enquanto que o “*sync*” descreve a sincroniza-

ção a partir da entidade analisada, o "*syncronized*" analisa a sincronização a partir das outras entidades. Graças a essa estratégia, as funcionalidades referentes a remoção de entidades é aplicada baseada em como as outras entidades definem o seu relacionamento com ela. As funcionalidades referentes a remoção de entidades são implementadas pelo método "*restful.deleteSync*". Tal método deve ser chamado antes de uma entidade ser removida.

A criação de *schemas* no *Alpha Restful* ocorre como uma mera interface para o *Mongoose*. Isto significa que os dados contidos em *"descriptor"* (definido na modelagem da entidade) é passado, tal qual como foi construído pelo programador, para a definição de *schemas* do *Mongoose*. Os dados contidos nesses *schemas* também são analisados pelo *framework*, apenas para a obtenção do tipo utilizado para cada atributo.

Na seção seguinte (resultados), os exemplos e as comparações entre o *Alpha Restful* e outras soluções de mercado serão apresentados.

5 RESULTADOS

O *Alpha Restful* facilita e automatiza a implementação de 5 funcionalidades sobre dados normalizados que, muitas vezes, são complexas de serem desenvolvidas no *MongoDB*, sem o uso de um *framework*. Para cada funcionalidade, será feito uma comparação, explicando sua implementação usando soluções de mercado e, posteriormente, é apresentado a implementação usando o *Alpha Restful*. O quadro 2 mostra quais soluções serão apresentadas para cada funcionalidade. Para a implementação destas funcionalidades, será utilizado como base a modelagem de dados exemplificada pelos documentos exemplo 7, 8, 9 e 10.

Quadro 2 – Comparação de Funcionalidades

Funcionalidade	**Soluções de Mercado**	***Alpha Restful***
Junção de Documentos	• *$lookup* • DBRef • *populate* • manual	• *fill* • relacionamento inverso • relacionamento transitivo
Buscas Sobre Documentos Normalizados	• *$lookup* e *$match* • manual	*query*
Remoção em Cascata de Documentos	manual	*deleteCascade*
Relação de Dependência Entre os Documentos	manual	*required*
Remoção de Identificadores Inválidos	manual	automático

Exemplo 7 – Documento da Casa de Número 1B

```
1 {
2     "_id": 10,
3     "rua": "Rua Castelo branco",
4     "numero": "1B"
5 }
```

Exemplo 8 – Documento da Casa de Número 1089

```
{
    "_id": 20,
    "rua": "Rua Pompel",
    "numero": "1089"
}
```

Exemplo 9 – Documento da Pessoa *Eduardo*

```
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "id": 20,
        "valorMensal": 300,
        "quantidadeMeses": 8
    }]
}
```

Exemplo 10 – Documento da Pessoa *Emanuel*

```
{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": 10,
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id ": 20,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
```

```
19 }
```

5.1 Funcionalidade 1: Junção de Documentos

Assim como foi visto na seção 2.2.5, as vezes, documentos que foram normalizados precisam ser unidos por meio de seus relacionamentos. A junção de documentos pode ser feita de maneira automática com a utilização do *$lookup*, *DBRef*, ou do *populate* do *Mongoose*, porém estas abordagens possuem algumas limitações, assim como será demonstrado nas subseções a seguir.

5.1.1 $lookup do MongoDB

O *$lookup* é uma operação disponibilizada de forma oficial pelo *MongoDB*. Tal operação se responsabiliza por unir dois documentos relacionados. Após tal união, pesquisas podem ser realizadas sobre esses dados e valores podem ser agrupados e ordenados. No exemplo 11 é apresentado uma codificação, utilizando o *$lookup*, para unir as coleções de pessoas e casas, ignorando os atributos de relacionamento.

Exemplo 11 – Junção de Documentos com Omissão

```
1     let UNIAO = await db.collection("pessoas").aggregate([
2         { $lookup: {
3             from: "casas",
4             localField: "casas.id",
5             foreignField: "_id",
6             as: "casas"
7         }}
8     ]).toArray()
```

A opção "*from*" contém o nome da coleção de documentos relacionada com a coleção de pessoas. O "*localField*" contém o nome do atributo que possui o identificador da entidade relacionada, contido no documento da coleção de pessoas. O "*foreignField*" contém o nome do atributo que possui o identificador da entidade relacionada, contido no documento da coleção de casas. A opção "*as*" possui o nome do atributo que conterá todos os atributos da entidade relacionada. Após essa operação, a variável "UNIAO" conterá o resultado apresentado pelo exemplo 12.

Exemplo 12 – Resultado da Junção com Omissão

```
1 [{
2     "_id": 1,
```

```
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": 10,
            "rua": "Rua Castelo Branco",
            "numero": "1B"
        },
        {
            "_id": 20,
            "rua": "Rua Pompel",
            "numero": 1089
        }
    ]
},{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "_id": 20,
        "rua": "Rua Pompel",
        "numero": "1089"
    }]
}]
```

No exemplo 12, observa-se que os atributos “valorMensal” e “quantidadeMeses” não estão contidos no resultado da junção. Isso ocorre porque o *$lookup* sobrescreve o atributo “casas” por todos os atributos presentes no documento de Casa. Pode-se observar também que o identificador (*_id*) da relação é substituído pelo identificador presente na própria entidade. Essa omissão de atributos pode ser um inconveniente, caso seja necessário obter ou realizar operações nos atributos que estão sendo omitidos. Para a utilização do *$lookup* sem a omissão de tais valores, uma codificação mais complexa e menos intuitiva tornaria-se necessária. No exemplo 13 é apresentado uma codificação que inclui os atributos anteriormente omitidos.

Exemplo 13 – Junção de Documentos sem Omissão

```
let UNIAO = await db.collection("pessoas").aggregate([
    { $unwind: "$casas" },
    { $lookup: {
        from: "casas",
```

```
5             let: { casas: "$casas" },
6             pipeline: [
7                 { $match: { $expr: {
8                     $eq: [ "$_id", "$$casas.id" ]
9                 }}},
10                { $addFields: {
11                    _id: "$$casas._id",
12                    id: "$$casas.id",
13                    valorMensal: "$$casas.valorMensal",
14                    quantidadeMeses: "$$casas.quantidadeMeses"
15                }}
16            ],
17            as: "casas"
18        }},
19        { $group: {
20            _id: {
21                _id: "$_id",
22                nome: "$nome",
23                idade: "$idade"
24            },
25            casas: { $push: "$casas" }
26        }},
27        { $project: {
28            _id: "$_id._id",
29            nome: "$_id.nome",
30            idade: "$_id.idade",
31            casas: { $reduce: {
32                input: "$casas",
33                initialValue: [],
34                in: { $concatArrays: [ "$$value", "$$this" ] }
35            }}
36        }}
37    ]).toArray()
```

Para adicionar os atributos de relacionamento dentro dos objetos de casa, tornou-se necessário utilizar-se de alguns artifícios do *MongoDB*, manipulando a união em baixo nível. Isso é necessário pois os identificadores das casas estão dentro de uma lista. Se uma pessoa pudesse, no máximo, ter uma única casa, uma codificação mais simples poderia ser realizada. Bastaria para isso usar o exemplo de código 11 e

apenas colocar na opção *"as"* do *$lookup*, um outro caminho que não sobrescreveria nenhum atributo já existente. Após a execução do exemplo de código 13, o resultado presente no exemplo 14 será obtido.

Exemplo 14 – Resultado da Junção de Documentos sem Omissão

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": 10,
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id": 20,
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "id": 20,
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

Pode-se observar que a implementação feita para obter uma simples junção de documentos pode ser complexa, sendo que tais operações são mais simples e intuitivas usando o SQL, assim como demonstrado no exemplo 1, da seção 2.1.2. A complexidade aumenta caso deseje-se fazer uniões em cascata, ou seja, unir documentos, que foram unidos com outros documentos. Quanto maior for o nível de uniões a serem feitas, mais complexo o código fica, podendo facilitar a ocorrência de erros humanos de codificação.

5.1.2 DBRef

O *DBRef* é um padrão para referenciar outros documentos de outras coleções. Essa convenção tem a finalidade de armazenar o nome da coleção relacionada (*$ref*), o identificador do documento (*$id*) e o nome do banco de dados na qual essa coleção está contida (*$db*). Se o *$db* não for informado, assume-se que a coleção está presente no banco de dados do documento que o *DBRef* reside. No exemplo a qual está sendo tratado, o *DBRef* pode ser utilizado para que as pessoas registradas no sistema possam se relacionar com suas casas, assim como demonstrado nos exemplos 15 e 16.

Exemplo 15 – Documento da Pessoa *Eduardo* com *DBRef*

```
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "$id": 20,
        "$ref": "casas",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }]
}
```

Exemplo 16 – Documento da Pessoa *Emanuel* com *DBRef*

```
{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
```

```
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "$id": 10,
            "$ref": "casas",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "$id": 20,
            "$ref": "casas",
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
}
```

Pode-se observar que o *DBRef* é utilizado quando o identificador do documento relacionado é armazenado em *$id* e quando está presente o atributo *$ref*, contendo o nome da coleção. Essa padronização é utilizada por algumas bibliotecas e *frameworks* para disponibilizar recursos de junção de documentos automáticas. Nesse caso, uniões de uniões de documentos podem ser feitas automaticamente de forma simples, dependendo da ferramenta que está sendo utilizada para o desenvolvimento. Esses recursos provenientes do *DBRef* não estão disponíveis em todas as linguagens, e cada biblioteca ou *framework* pode tratar isso de forma diferente.

Os atributos de relacionamento ("*_id*", "*valorMensal*", "*quantidadeMeses*") não necessariamente são tratados pela biblioteca ou *framework* utilizado, podendo eles serem ignorados. Nesse caso, os dados precisariam ser remodelados para extrair esses atributos para outro lugar. Uma maneira de fazer isso seria criar uma nova coleção de documentos auxiliares. Tais documentos armazenariam os atributos de relacionamento e os identificadores das entidades relacionadas. Depois seria necessário fazer um *DBRef* com o novo documento criado. Caso esses documentos auxiliares sejam aplicados na modelagem exemplo trabalhada nessa seção, os documentos se pareceriam com os exemplos 17, 18, 19, 20, 21, 22 e 23.

Exemplo 17 – Documento de *Eduardo* Usando Documento Auxiliar

```
{
    "_id": 2,
    "nome": "Eduardo",
```

```
    "idade": 40,
    "casas": [{
        "$id": ObjectId("5e66cb581766a2056c48145f"),
        "$ref": "relacionamento_pessoas_casas"
    }]
}
```

Exemplo 18 – Documento de *Emanuel* Usando Documento Auxiliar

```
{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "$id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "$ref": "relacionamento_pessoas_casas"
        },
        {
            "$id": ObjectId("5e66cb43534f8a1944cdb028"),
            "$ref": "relacionamento_pessoas_casas"
        }
    ]
}
```

Exemplo 19 – Relacionamento de *Eduardo* com sua Casa

```
{
    "_id": ObjectId("5e66cb581766a2056c48145f"),
    "valorMensal": 300,
    "quantidadeMeses": 8,
    "casa": {
        "$id": 20,
        "$ref": "casas"
    }
}
```

Exemplo 20 – Relacionamento de *Emanuel* Com Sua Primeira Casa

```
{
    "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
    "valorMensal": 200,
```

```
    "quantidadeMeses": 12,
    "casa": {
        "$id": 10,
        "$ref": "casas"
    }
}
```

Exemplo 21 – Relacionamento de *Emanuel* Com Sua Segunda Casa

```
{
    "_id": ObjectId("5e66cb43534f8a1944cdb028"),
    "valorMensal": 400,
    "quantidadeMeses": 24,
    "casa": {
        "$id": 20,
        "$ref": "casas"
    }
}
```

Exemplo 22 – Documento da Casa de Número 1B

```
{
    "_id": 10,
    "rua": "Rua Castelo branco",
    "numero": "1B"
}
```

Exemplo 23 – Documento da Casa de Número 1089

```
{
    "_id": 20,
    "rua": "Rua Pompel",
    "numero": "1089"
}
```

Esses documentos auxiliares que representam o relacionamento de pessoa com casa são necessários, caso o *DBRef* esteja sendo utilizado e as ferramentas de desenvolvimento usadas estejam ignorando os atributos de relacionamento. Dependendo das bibliotecas e *frameworks* utilizados, pode ser que esses documentos auxiliares não sejam necessários, e os dados desses documentos possam ser inseridos no documento principal. A abordagem a ser utilizada dependerá da linguagem e da plataforma.. Se o ambiente de execução utilizado suportar tal estratégia, seria

possível relacionar as pessoas com suas casas por meio de uma modelagem parecida com os exemplos 24, 25, 26 e 27.

Exemplo 24 – Documento da Pessoa *Eduardo* Sem Documento Auxiliar

```
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "valorMensal": 300,
        "quantidadeMeses": 8,
        "casa": {
            "$id": 20,
            "$ref": "casas"
        }
    }]
}
```

Exemplo 25 – Documento da Pessoa *Emanuel* Sem Documento Auxiliar

```
{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "valorMensal": 200,
            "quantidadeMeses": 12,
            "casa": {
                "$id": 10,
                "$ref": "casas"
            }
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "valorMensal": 400,
            "quantidadeMeses": 24,
            "casa": {
                "$id": 20,
```

```
                "$ref": "casas"
            }
        }
    ]
}
```

Exemplo 26 – Documento da Casa de Número 1B

```
{
    "_id": 10,
    "rua": "Rua Castelo branco",
    "numero": "1B"
}
```

Exemplo 27 – Documento da Casa de Número 1089

```
{
    "_id": 20,
    "rua": "Rua Pompel",
    "numero": "1089"
}
```

Essa última abordagem apresentada possui a vantagem de não existir um documento auxiliar intermediando o relacionamento. Isso é útil por diminuir o armazenamento e por permitir que consultas mais complexas sejam mais simples de se implementar.

Assim como explicado anteriormente, o *DBRef* pode permitir que operações de união de documentos ocorram de forma mais simples e automática, caso o ambiente de execução utilizado disponibilize tais opções para os BSONs que seguem seu padrão. Apesar dos benefícios, operações mais complexas como, por exemplo, buscas sobre valores presentes em vários documentos, podem não estar disponíveis via *DBRef*. Além disso, operações como o *$lookup* podem exigir que o *DBRef* seja convertido para um outro formato antes de tais operações serem realizadas.

Por causa disso, o uso do *DBRef* pode exigir codificações mais complexas em determinadas situações onde a biblioteca ou *framework* não daria suporte. Por essas razões que, dependendo das ferramentas disponibilizadas pelo ambiente de execução, bem como das necessidades do projeto, pode ser que o uso de uma referencia manual de outros documentos seja uma melhor escolha do que o *DBRef*.

5.1.3 Populate do Mongoose

É possível, através da biblioteca *Mongoose*, unir documentos normalizados em um único documento resultante. Para que tal operação seja possível, no *Mongoose*, torna-se necessário a implementação de *schemas* para cada coleção de documentos gerenciada pela biblioteca. Para o exemplo que está sendo apresentado, os *schemas* para as coleções de pessoas e casas podem ser definidos, assim como demonstrado no exemplo 28.

Exemplo 28 – Definição de *Schemas* no *Mongoose*

```
const PessoaSchema = new mongoose.Schema({
    _id: Number,
    nome: String,
    idade: Number,
    casas: [{
        _id: mongoose.Schema.Types.ObjectId,
        id: {
            type: Number,
            ref: "casas"
        },
        valorMensal: Number,
        quantidadeMeses: Number
    }]
})
const Pessoa = db.model("pessoas", PessoaSchema)

const CasaSchema = new mongoose.Schema({
    _id: Number,
    rua: String,
    numero: String
})
const Casa = db.model("casas", CasaSchema)
```

Um dos recursos disponibilizados pelo *Mongoose* é o *populate*. Este recurso substitui o *DBRef* e o *$lookup* para operações de busca, seguido da junção de documentos. Uma das restrições do *populate*, em comparação com o *$lookup*, é que no *$lookup* é possível realizar buscas sobre os dados dos documentos unidos, enquanto que no *populate* a busca deve ocorrer apenas sobre os dados originais de cada documento. Se desconsiderarmos essa limitação, o *populate* é mais simples de se usar do que o *$lookup*, além de disponibilizar novas opções e recursos. Atualmente, o *Mongo-*

ose está disponível apenas para o *Node JS*, que é exatamente o escopo a qual esse trabalho se propõe em atual. O exemplo 29 apresenta a junção das coleções de pessoas e casas usando o *populate*. Com apenas uma única linha, os dois documentos são unidos, mantendo os atributos de relacionamento. Após a execução do código exemplo 29, a variável "UNIAO" terá o resultado apresentado pelo exemplo 30.

Exemplo 29 – Junção de Documentos Com o *Populate*

```
let UNIAO = await Pessoa.find({}).populate("casas.id").exec()
```

Exemplo 30 – Resultado da Junção de Documentos Com o *Populate*

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": {
                "_id": 10,
                "rua": "Rua Castelo Branco",
                "numero": "1B"
            },
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id": {
                "_id": 10,
                "rua": "Rua Pompel",
                "numero": 1089
            },
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},
{
    "_id": 2,
    "nome": "Eduardo",
```

```
    "idade": 40,
    "casas": {
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "id": {
            "_id": 10,
            "rua": "Rua Pompel",
            "numero": 1089
        },
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

Pode-se observar que os valores de casa foram preenchidos no atributo onde encontra-se o identificador do documento. Por causa dessa característica do *populate*, pode ser mais intuitivo, para este caso, remodelar a entidade de pessoa, para que o atributo "*id*" seja renomeado para, por exemplo, "casa". Nesse caso, os resultados sairiam com o formato apresentado pelo exemplo 31.

Exemplo 31 – Resultado do *Populate* com "id" Renomeado para "casa"

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "casa": {
                "_id": 10,
                "rua": "Rua Castelo Branco",
                "numero": "1B"
            },
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "casa": {
                "_id": 10,
                "rua": "Rua Pompel",
```

```
                "numero": 1089
            },
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "casa": {
            "_id": 10,
            "rua": "Rua Pompel",
            "numero": 1089
        },
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

5.1.4 Junção de Documentos de Forma Manual

Caso haja a necessidade de obter um maior controle na junção dos documentos, ou se os recursos de junção automático não forem satisfatórios o suficiente para as necessidades da aplicação, é possível unir documentos de forma manual. Uma codificação manual está apresentado no exemplo 32. Ao final do código deste exemplo, a variável “pessoas” armazenará o resultado apresentado pelo exemplo 33.

Exemplo 32 – Junção Manual dos Documentos de Pessoa com Casa

```
let pessoas = await db.collection("pessoas")
    .find({}).toArray();

for (let p of pessoas) {
    for (let i = 0; i < p.casas.length; i++) {
        let c = p.casas[i];

        let casa = (await db.collection("casas").find({
```

```
                "_id": c.id
            }).toArray())[0];

            p.casas[i] = {
                ...casa,
                ...c
            };
        }
    }
```

Exemplo 33 – Junção de Documentos de Forma Manual

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": 10,
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id": 20,
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
```

```
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "id": 20,
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

5.1.5 Usando o Alpha Restful Para Unir Documentos

O *Alpha Restful* possui sua própria implementação para unir os documentos. Internamente, os documentos sempre são unidos de forma manual. A vantagem de se utilizar o *Alpha Restful* é que ele disponibiliza duas novas funcionalidades que não estão diretamente disponíveis pelos métodos descritos anteriormente:

- Relacionamento inverso
- Relacionamento inverso de relacionamento inverso

Para que as funcionalidades do *framework* sejam disponibilizadas, o *Alpha Restful* utiliza os *schemas* do *Mongoose*, em conjunto com especificações de sincronização (*sync*) entre as entidades. Essas especificações de sincronização permitem que entidades sejam relacionadas entres si, baseando-se em identificadores e em outros tipos de relacionamentos (que não serão explicados por fugir do escopo desse trabalho). O exemplo 34 apresenta uma codificação para criar os *schemas* e especificações de sincronização das entidades Pessoa e Casa, usando o *Alpha Restful*.

Exemplo 34 – Definição de *Schemas* no *Alpha Restful*

```
1   const restful = new Restful(
2       "<nome-da-aplicacao>",
3       { locale: "pt" }
4   )
5
6   const Pessoa = new Entity({
7       name: "Pessoa",
8       resource: "pessoas",
9       descriptor: {
10          nome: String,
```

```
            idade: Number,
            casas: [{
                id: Number,
                valorMensal: Number,
                quantidadeMeses: Number
            }]
        },
        sync: {
            casas: {
                name: "Casa",
                fill: true
            }
        }
    })

    const Casa = new Entity({
        name: "Casa",
        resource: "casas",
        descriptor: {
            rua: String,
            numero: String
        }
    })

    restful.add(Pessoa)
    restful.add(Casa)
```

Para a junção de documentos, o *Alpha Restful* disponibiliza uma opção denominada de *fill*, que implementa essa funcionalidade sobre qualquer objeto já pesquisado ou montado. Essa opção também possibilita que, em tempo de execução, relacionamentos temporários com outras entidades possam ser criados. O exemplo 35 realiza uma junção de documentos sobre as pessoas e casas, usando a modelagem definida no exemplo 34.

Exemplo 35 – Junção de Documentos Com o *Alpha Restful*

```
let pessoas = await Pessoa.model.find({}).exec()
pessoas = await Pessoa.fill(pessoas, restful)
```

A operação de junção de documentos ocorre após os objetos terem sido obtidos, por exemplo, por meio de uma busca. Nesse caso buscou-se por todas as

pessoas. Após essa busca, basta chamar o método *Pessoa.fill* para que os documentos sejam unidos. Como na modelagem já tinha sido definido que o atributo "casas" de pessoa fazia um relacionamento com a entidade "Casa" (através do objeto *sync*), e que por padrão os documentos devem ser unidos (através da opção *fill* igual a *true* em *sync*), apenas a chamada do método é o suficiente para realizar a junção. Na versão atual do *Alpha Restful* (0.7.38), o identificador da entidade relacionada precisa ser definido no atributo *id*. Depois de executar o código 35, a variável "pessoas" obterá o resultado apresentado pelo exemplo 36.

Exemplo 36 – Resultado da Junção de Documentos Com o *Alpha Restful*

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": 10,
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id": 20,
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "id": 20,
```

```
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

5.1.5.1 Relacionamento Inverso

Uma das funcionalidades disponibilizadas pelo *Alpha Restful* que, atualmente, não está diretamente disponíveis nos outros métodos de junção de documentos mostrados (com exceção do método manual), é o relacionamento inverso. Para ilustrar tal funcionalidade, pode-se analisar a situação onde seria necessário unir as duas coleções, mas juntando nos documentos de casa. Essa operação pode ser feita diretamente no método de *fill*, sem a necessidade de alterar a modelagem da entidade. O exemplo 37 apresenta uma codificação que realiza tal junção, definindo o relacionamento na entidade Casa, diretamente pelo método *Casa.fill*, sem alterar a modelagem apresentada no exemplo 34.

Exemplo 37 – Junção de Documentos em Relacionamento Inverso

```
    let casas = await Casa.model.find({}).exec()
    casas = await Casa.fill(casas, restful, { sync: {
        pessoas: {
            name: "Pessoa",
            syncronized: ["casas"],
            fill: true,
            jsonIgnoreProperties: "casas"
        }
    }})
```

A opção "*jsonIgnoreProperties*", nesse caso, é responsável por ignorar o atributo "casas" de Pessoa. Isso é necessário para que não ocorra uma recursão infinita. Sem essa opção, as pessoas seriam preenchidas no atributo "pessoas", as casas seriam preenchidas no atributo "casas", as pessoas seriam novamente preenchidas no atributo "pessoas" e assim por diante. Com a opção "*jsonIgnoreProperties*", as casas não serão incluídas nas pessoas. Após a execução do código exemplo 37, as coleções são unidas, mas tendo como base a entidade Casa. Os documentos unidos estarão presentes na variável "casas" e terá a estrutura definida pelo exemplo 38.

Exemplo 38 – Resultado da Junção de Documentos em "Casa"

```
[
    {
        "_id": 10,
        "rua": "Rua Castelo Branco",
        "numero": "1B",
        "pessoas": [
            {
                "id": 1,
                "nome": "Emanuel",
                "idade": 21
            }
        ]
    },
    {
        "_id": 20,
        "rua": "Rua Pompel",
        "numero": "1089",
        "pessoas": [
            {
                "id": 1,
                "nome": "Emanuel",
                "idade": 21
            },
            {
                "id": 2,
                "nome": "Eduardo",
                "idade": 40
            }
        ]
    }
]
```

Como na modelagem da entidade Casa o relacionamento com pessoa não foi definido, é possível fazer essa definição na hora de realizar a junção. Pode-se observar que não existe a necessidade de armazenar os identificadores das pessoas nas suas casas, o *framework* consegue automaticamente identificálos, bastando apenas informar na opção "*syncronized*" o caminho para se obter as casas por meio de uma pessoa. Caso fosse desejado obter esse comportamento por padrão, assim como

ocorre em “Pessoa”, bastaria atualizar o objeto “*sync*” da entidade “Casa”. Se isso for feito, a junção poderá ocorrer sem a definição do relacionamento na hora de chamar o método de junção. Neste caso, a modelagem de Casa seria definido, assim como apresentado no exemplo 39. Se a modelagem for definida desta maneira, a junção poderá ser executada, assim como demonstrado no exemplo 40.

Exemplo 39 – Relacionamento Inverso em *Schema* de Casa

```
1    const Casa = new Entity({
2        name: "Casa",
3        resource: "casas",
4        descriptor: {
5            rua: String,
6            numero: String
7        },
8        sync: {
9            pessoas: {
10               name: "Pessoa",
11               syncronized: ["casas"],
12               fill: true,
13               jsonIgnoreProperties: "casas"
14           }
15       }
16   })
```

Exemplo 40 – Junção Com o *Alpha Restful* em Relacionamento Inverso

```
1  let casas = await Casa.model.find({}).exec()
2  casas = await Casa.fill(casas, restful)
```

5.1.5.2 Relacionamento Transitivo

Outra funcionalidade disponibilizada pelo *Alpha Restful* que, atualmente, não está disponível nos outros métodos de junção de documentos mostrados (com exceção do método manual), é o relacionamento transitivo. Uma forma de ilustrar esta função é tentar obter, no documento das pessoas, a lista de moradores de uma ou mais casas que a própria pessoa também mora. Para isto, bastaria fazer um relacionamento do atributo “pessoas” em casas, assim como definido no exemplo 41.

Exemplo 41 – Relacionamento Inverso de Relacionamento Inverso

```
const restful = new Restful("<nome-da-aplicacao>", {
    locale: "pt"
})

const Pessoa = new Entity({
    name: "Pessoa",
    resource: "pessoas",
    descriptor: {
        nome: String,
        idade: Number,
        casas: [{
            id: Number,
            valorMensal: Number,
            quantidadeMeses: Number
        }]
    },
    sync: {
        casas: {
            name: "Casa",
            fill: true,
            jsonIgnoreProperties: "pessoas"
        },
        residentes: {
            name: "Pessoa",
            syncronized: ["casas.pessoas"],
            fill: true,
            jsonIgnoreProperties: ["residentes", "casas"]
        }
    }
})

const Casa = new Entity({
    name: "Casa",
    resource: "casas",
    descriptor: {
        rua: String,
        numero: String
    },
```

```
39          sync: {
40              pessoas: {
41                  name: "Pessoa",
42                  syncronized: ["casas"],
43                  fill: true,
44                  jsonIgnoreProperties: ["casas", "residentes"]
45              }
46          }
47      })
48
49      restful.add(Pessoa)
50      restful.add(Casa)
```

Os *schemas* definidos no exemplo 41 criam, nos documentos de pessoas, um novo atributo (“residentes”) que contém todas as pessoas que então contidas no atributo definido em *“syncronized”* que, neste caso, é o atributo “casas.pessoas”. Pode-se observar a presença da opção “*jsonIgnoreProperties*” no “*sync*” das entidades. Tal opção armazena o nome do atributo (poderia ser uma lista de nomes de atributos) que será ignorado nos documentos que serão unidos. Isso é necessário para impedir uma união recursiva infinita de atributos. Da forma como a modelagem está definida, ao unir os documentos de pessoas e casas, haverá um atributo nos documentos de pessoas chamado de “residentes”. Este, conterá todas as pessoas que moram em uma ou mais casas na qual a própria pessoa também mora. Essa união de documentos pode ser realizado por meio do código exemplo 42.

Exemplo 42 – Junção de Documentos Com o *Alpha Restful*

```
1   let pessoas = await Pessoa.model.find({}).exec()
2   pessoas = await Pessoa.fill(pessoas, restful)
```

Por causa do relacionamento transitivo, a junção de documentos por meio do *Alpha Restful* se torna mais simples de implementar em comparação que as outras soluções descritas anteriormente. Ao final da execução do código exemplo 42, a variável “pessoas” terá o resultado do exemplo 43.

Exemplo 43 – Resultado da Junção de Documentos Com Residentes

```
1  [{
2      "_id": 1,
3      "nome": "Emanuel",
4      "idade": 21,
5      "casas": [
6          {
```

```
            "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
            "id": 10,
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "_id": ObjectId("5e66cb43534f8a1944cdb028"),
            "id": 20,
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ],
    "residentes": [
        {
            "id": 1,
            "nome": "Emanuel",
            "idade": 21,
        },
        {
            "id": 2,
            "nome": "Eduardo",
            "idade": 40,
        }
    ]
},
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "_id": ObjectId("5e66cb581766a2056c48145f"),
        "id": 20,
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
```

```
46          "quantidadeMeses": 8
47      }],
48      "residentes": [
49          {
50              "id": 1,
51              "nome": "Emanuel",
52              "idade": 21,
53          },
54          {
55              "id": 2,
56              "nome": "Eduardo",
57              "idade": 40,
58          }
59      ]
60  }]
```

5.1.6 Junção de Documentos equivalente ao inner join do SQL

Os exemplos de código para junção de documentos apresentados anteriormente são equivalentes ao *left join* do SQL. Isto significa que a entidade sobre a qual a busca é feita, é obtida, mesmo que não haja nenhum documento a ser unido com ela. Caso seja necessário haver uma junção na qual deve ser obtido apenas as entidades que possuem documentos a serem unidos, ou seja, uma junção equivalente ao *inner join* do SQL, pode-se filtrar apenas as entidades que possuem um identificador de outro documento. No exemplo 35, fazer uma junção equivalente ao *inner join* obteria apenas as pessoas que possuem casas. Isto pode ser implementado, assim como demonstrado no exemplo 44, usando o *Alpha Restful*.

Exemplo 44 – *inner join* de Documentos Com o *Alpha Restful*

```
1  let pessoas = await Pessoa.model.find({
2      "casas.id": { $ne: null }
3  }).exec()
4  pessoas = await Pessoa.fill(pessoas, restful)
```

5.2 Funcionalidade 2: Buscas Sobre Documentos Normalizados

O *MongoDB* é nativamente capaz de realizar buscas complexas e simples dentro de um mesmo documento. Porém, a partir do momento que as buscas preci-

sam ser realizadas em documentos normalizados, as pesquisas passam a ficar mais difíceis de serem realizadas. No exemplo 45 é apresentado um código que realiza uma busca pelas pessoas cuja a idade seja igual a 40.

Exemplo 45 – Busca de Pessoas com Idade Igual a 40

```
1    let pessoas = await db.collection("pessoas").find({
2        "idade": 40
3    }).toArray()
```

Ao final desse código, a variável “pessoas” obterá todas as pessoas na qual a idade é igual a 40. O código é simples, pois a busca ocorre dentro do próprio documento. Mas e se for desejado obter todas as casas, na qual existe pelo menos uma pessoa, que essa pessoa possui pelo menos uma casa, que nessa casa possui pelo menos uma pessoa que possui a idade igual a 40 anos?

5.2.1 Pesquisa Usando o $lookup e $match

Para realizar a pesquisa proposta, uma abordagem possível é unir os documentos utilizando o *$lookup*, até o nível que todos os dados estejam no mesmo documento. Após essa união, torna-se possível fazer a pesquisa, utilizando o parâmetro “*$match*”. No exemplo 46 é apresentado uma implementação da busca proposta, utilizando *$lookup* e *$match*.

Exemplo 46 – Busca em Dados Normalizados Com o *$lookup*

```
1    let RESULTADO = await db.collection("casas").aggregate([
2        { $lookup: {
3            from: "pessoas",
4            localField: "_id",
5            foreignField: "casas.id",
6            as: "pessoas"
7        }},
8        { $unwind: "$pessoas" },
9        { $lookup: {
10           from: "casas",
11           localField: "pessoas.casas.id",
12           foreignField: "_id",
13           as: "pessoas.casas"
14       }},
15       { $unwind: "$pessoas.casas" },
16       { $lookup: {
```

```
            from: "pessoas",
            localField: "pessoas.casas._id",
            foreignField: "casas.id",
            as: "pessoas.casas.pessoas"
        }},
        { $group: {
            _id: {
                _id: "$_id",
                rua: "$rua",
                numero: "$numero",
                pessoas: {
                    _id: "$pessoas._id",
                    nome: "$pessoas.nome",
                    idade: "$pessoas.idade",
                }
            },
            casas: {
                $push: "$pessoas.casas"
            }
        }},
        { $project: {
            _id: "$_id._id",
            rua: "$_id.rua",
            numero: "$_id.numero",
            pessoas: {
                _id: "$_id.pessoas._id",
                nome: "$_id.pessoas.nome",
                idade: "$_id.pessoas.idade",
                casas: "$casas"
            }
        }},
        { $group: {
            _id: {
                _id: "$_id",
                rua: "$rua",
                numero: "$numero"
            },
            pessoas: {
                $push: "$pessoas"
```

```
56              }
57          }},
58          { $project: {
59              _id: "$_id._id",
60              rua: "$_id.rua",
61              numero: "$_id.numero",
62              pessoas: "$pessoas"
63          }},
64          { $match: {
65              "pessoas.casas.pessoas.idade": 40
66          }}
67      ]).toArray()
```

Para a realização de uma pesquisa dessa complexidade, é necessário unir os documentos várias vezes, pois é preciso acessar os dados que estão dentro de uma lista (pessoas), que estão dentro de uma lista (pessoas.casas), que estão dentro de uma lista (pessoas.casas.pessoas). Por essa razão, utilizar o *$lookup* para consultas pode ser complexo.

5.2.2 Pesquisa Manual

Também é possível realizar a pesquisa proposta de forma manual. Para isso, pode-se subdividir a pesquisa em pesquisas menores. Logo após, basta uni-las em uma pesquisa que irá obter o resultado esperado. O exemplo 47 apresenta uma codificação utilizando esta abordagem, sendo necessário 31 linhas de código.

Exemplo 47 – Busca em Dados Normalizados de Forma Manual

```
1   let pessoasIdade40 = await db.collection("pessoas").find({
2       "idade": 40
3   }).toArray();
4
5   let idsCasasPessoasIdade40 =
6   pessoasIdade40.map(p =>
7       p.casas.reduce((a,c) => [...a,c.id], [])
8   ).reduce((a,lid) => [...a, ...lid], []);
9
10  let casasPessoasIdade40 =
11  await db.collection("casas").find({
12      "_id": { $in: idsCasasPessoasIdade40 }
13  }).toArray();
```

```
14
15      let idsCasasPessoasIdade40 =
16      casasPessoasIdade40.map(c => c._id);
17
18      let pessoasCasasPessoasIdade40 = await
19      db.collection("pessoas").find({
20          "casas.id": { $in: idsCasasPessoasIdade40 }
21      }).toArray();
22
23      let idsCasasPessoasCasasPessoasIdade40 =
24      idsCasasPessoasCasasPessoasIdade40.map(p =>
25          p.casas.reduce((a,c) => [...a,c.id], [])
26      ).reduce((a,lid) => [...a, ...lid], []);
27
28      let RESULTADO_DA_PESQUISA = await
29      db.collection("casas").find({
30          "_id": { $in: idsCasasPessoasCasasPessoasIdade40 }
31      }).toArray();
```

O primeiro passo para realizar esta busca é de obter todas as pessoas que possui idade igual a 40 anos (linha 1 a 3). Depois, os identificadores das casas pertencentes a estas pessoas são extraídos (linhas 5 a 8). Após a extração destes identificadores, são buscados todas as casas que possui um identificador dentre esses (linhas 10 a 12). Após a busca de todas essas casas, são extraídos todos os identificares (linhas 14 a 15). Após a extração desses identificadores, são buscadas todas as pessoas que possuem alguma destas casas (linhas 17 a 20). Após a busca destas pessoas, são extraídos todos os identificadores das casas destas pessoas (linhas 22 a 25). Finalmente, as casas que possui seu identificador dentre os identificadores são buscadas, obtendo o resultado esperado pela consulta. Observa-se que, tanto esta consulta, quando a consulta usando o *$lookup* e *$match*, ficariam mais complexas com a adição de outros filtros, utilizando-se de outras relações com outros documentos relacionados.

5.2.3 Pesquisa Usando o Alpha Restful

Como observado nas seções 5.2.1 e 5.2.2, realizar buscas que englobam vários documentos pode ser algo complexo. Para contornar este problema, o *Alpha Restful* mapeia todos os relacionamentos normalizados dentre todos os documentos do sistema para fornecer uma sintaxe simples para se realizar pesquisas. Para ilustrar

isto, é demonstrado no exemplo 48 como a mesma pesquisa, demonstrada nas seções 5.2.1 e 5.2.2, pode ser implementada usando o *Alpha Restful*.

Exemplo 48 – Busca em Dados Normalizados com o *Alpha Restful*

```
1    let casas = await restful.query({
2        "pessoas.casas.pessoas.idade": 40
3    }, Casa);
```

Anteriormente, para a realização dessa pesquisa, foi necessário entre 31 (exemplo 47) e 67 (exemplo 46) linhas, mas com o *Alpha Restful*, apenas 3 linhas foram o suficiente. Isso ocorre porque o *Alpha Restful* consegue enxergar todos os atributos presentes em outros documentos normalizados, como se eles estivessem dentro do documento de maneira desnormalizada. A sintaxe utilizada para realizar buscas foi inspirada no *Mongoose*, com a diferença de considerar nas pesquisas os atributos contidos em outros documentos relacionados.

5.3 Funcionalidade 3: Remoção em Cascata de Documentos

No exemplo a qual está sendo abordado, pode-se supor que, por exemplo, o sistema possua uma regra de negócio que afirme que quando uma pessoa for removida, todas as casas pertencentes a ela devam ser removidas também. Para a implementação de tal regra, sem o uso de um *framework*, toda vez que uma pessoa for removida, será necessário buscar por todas as casas pertencentes a ela e, manualmente, removê-las. O problema desta abordagem manual é que uma coleção pode possuir cada vez mais relacionamentos com outros documentos, deixando o código cada vez mais complexo.

Pensando nisto, o *Alpha Restful* disponibiliza uma opção na sincronização das entidades (*sync*), que implementa exatamente esta funcionalidade. Para garantir este comportamento, é necessário apenas informar a opção *deleteCascade* no atributo da entidade a qual deseja-se que seja removida automaticamente. No exemplo 49 é apresentada tal implementação.

Exemplo 49 – Modelagem de "Casa" com *deleteCascade*

```
1    const Casa = new Entity({
2        name: "Casa",
3        // ...
4        sync: {
5            pessoas: {
6                name: "Pessoa",
7                syncronized: ["casas"]
```

```
                fill: true,
                jsonIgnoreProperties: "casas",
                deleteCascade: true
            },
            // ...
        }
    })
```

5.4 Funcionalidade 4: Relação de Dependência Entre os Documentos

No exemplo a qual está sendo abordado, se uma casa não puder ser removida, caso possua um relacionamento com alguma pessoa, seria necessário a verificação da existência de tal relacionamento, antes de uma casa ser removida. Caso todo esse procedimento seja feito manualmente e outras entidades comecem a se relacionar com a entidade Casa, o código desta verificação ficaria cada vez mais complexo, se esta regra se repetisse para outros documentos.

Para que esta regra seja aplicada de forma mais simples, o *Alpha Restful* disponibiliza no "*sync*" uma opção que define uma relação de dependência entre os documentos. Uma relação de dependência entre documentos normalizados relacionados garante que um documento não possa ser removido se estiver presente em algum relacionamento definido como dependente. Se, por exemplo, uma casa não puder ser removida, caso possua alguma pessoa, bastaria apenas informar a opção *required* no atributo da entidade a qual deseja-se criar um relacionamento de dependência (Pessoa). O exemplo 50 apresenta a codificação necessária para que tal opção seja utilizada.

Exemplo 50 – Modelagem de Pessoa com *required*

```
const Pessoa = new Entity({
    name: "Pessoa",
    // ...
    sync: {
        casas: {
            name: "Casa",
            fill: true,
            jsonIgnoreProperties: "pessoas",
            required: true
        },
```

```
11                // ...
12            }
13        })
```

5.5 Funcionalidade 5: Remoção de Identificadores Inválidos

Um dos possíveis problemas que podem ser comuns no desenvolvimento de uma aplicação com *MongoDB*, é a existência de identificadores que apontam para documentos que não existem. Isto acontece porque as entidades que possuem um identificador de outra entidade que foi removida do banco, podem continuar com esse identificador. Um exemplo que pode ser apresentado é de que se uma casa for removida, isso pode fazer com que o identificador dessa casa nos documentos da coleção de pessoas aponte para lugar algum, pois a casa a qual tais identificadores apontam, já não existe mais.

Para que este problema seja contornado, sem o uso de um *framework*, é necessário que antes que qualquer entidade seja removida, seja realizada uma análise em todas as entidades que se relacionam com a instância a qual deseja-se remover. Após esta análise, os dados que apontariam para esta entidade que está sendo removida seriam removidos também. Com o aumento da complexidade da aplicação, esse código ficaria cada vez mais e mais complexo, pois a cada novo relacionamento entre documentos, mais alterações precisariam ser feitas no código para garantir esse comportamento. O *Alpha Restful* já resolve esse problema automaticamente (bastando apenas realizar o procedimento descrito na seção 5.6). Nenhuma opção precisa ser habilitada na modelagem para que esse problema seja mitigado.

5.6 *deleteSync*

Para que as funcionalidades relacionadas à remoção de entidades no *Alpha Restful* possam acontecer (seções 5.3, 5.4 e 5.5), é necessário que o método *restful.deleteSync* seja chamado antes que qualquer entidade ser removida. O exemplo 51 apresenta o código a ser executado antes de qualquer casas ser removida.

Exemplo 51 – Antes de Remover Uma Casa

```
1  await restful.deleteSync(casa._id, "Casa", Casa.syncronized)
```

6 CONCLUSÃO

Os bancos de dados NoSQL trazem o conceito de que o desenvolvimento de uma aplicação não precisa necessariamente estar preso às regras e limites do modelo relacional. A proposta de tais bancos inclui mudar a estrutura de como os dados são armazenados, a fim de eliminar alguma barreira imposta pelo SQL. No caso do *MongoDB*, propõe-se otimizar o armazenamento e gerenciamento de dados que não precisam seguir as formas normais. Ao mesmo tempo, ele não impede a normalização, onde tal abordagem for pertinente e se demonstrar uma vantagem. Permitir que a aplicação decida sobre a normalização ou desnormalização, demonstra-se ser um bom caminho, pois possibilita que a melhor decisão possa ser tomada, dependendo das necessidades da aplicação a ser desenvolvida.

O *MongoDB* disponibiliza, de forma oficial, várias recursos para auxiliar na desnormalização dos dados. Apesar disso, este trabalho demonstrou que, até mesmo em aplicações onde a desnormalização é importante, eventualmente, os dados poderão precisar estar normalizados. Quando isso acontecer, várias operações poderão ser bastante complexas de serem desenvolvidas, em comparação com o SQL. Nesse cenário, este trabalho apresentou o *Alpha Restful* como uma solução capaz de minimizar tal problemática, simplificando operações complexas e disponibilizando novos recursos para elas.

A fim de mostrar as vantagens do *Alpha Restful*, cinco funcionalidades comuns para o desenvolvimento de aplicações com *MongoDB* foram analisadas. Tais análises demonstram a relevância da solução proposta para melhorar o processo de desenvolvimento de tais funcionalidades. Com o *Alpha Restful*, as pesquisas normalizadas demonstraram ser mais simples que os disponíveis em outras ferramentas de mercado. Além disso, nas junções de documentos, o *framework* proposto disponibiliza dois novos recursos (o relacionamento inverso e o relacionamento transitivo). Com esses novos recursos, novas funcionalidades podem ser implementadas, de maneira simples e intuitiva.

A terceira, quarta e quinta funcionalidades são tratadas pelo *framework* de maneira automática, bastando ativá-las através do uso de opções no objeto de sincronização do atributo. Isso é uma grande melhoria, pois nas outras ferramentas de mercado apresentadas, essas funcionalidades não são trabalhadas automaticamente, exigindo implementações manuais mais complexas de se fazer.

Uma das coisas que precisa ficar claras sobre o *Alpha Restful* é de que ele está em verão *beta*. A sua atual implementação é apenas uma prova de conceito. O

objetivo de tal prova é propor uma nova ferramenta funcional para suprir as necessidades que estão descritas nesse trabalho.

As funcionalidades do *Alpha Restful* aqui descritas, já estão disponíveis e foram testadas. O motivo se denominar essa ferramenta em fase *beta* e como uma prova de conceito, é de que não foram feitas as otimizações necessárias para que uma aplicação que utilize tal *framework* possa ser utilizada em produção. Além desse fato, algumas funcionalidades também relevantes não foram desenvolvidas ainda.

No futuro, pretende-se permitir a definição de sub-consultas dentro da própria modelagem das entidades. Isto é equivalente às *views* do SQL. Além disso, sub-consultas poderão ser utilizadas nos métodos de busca descritos na seção 5.2.3. Pretende-se também, disponibilizar interfaces para que outros bancos de dados NoSQL possam ser usados pelo *Alpha Restful*. Outras melhorias de performance serão realizadas a fim de que uma versão estável possa ser lançada.

REFERÊNCIAS

ANSARI, H. *Performance Comparison of Two Database Management Systems MySQL vs MongoDB*. 2018.

ARBOLEDA, F. J. M.; RENDÓN, J. E. Q.; VÁSQUEZ, R. R. A performance comparison between oracle and mongodb. *Ciencia e Ingeniería Neogranadina*, Universidad Militar Nueva Granada, v. 26, n. 1, p. 109–129, 2016.

BOAGLIO, F. *MongoDB: construa novas aplicações com novas tecnologias*. [S.l.]: Editora Casa do Código, 2015.

CELESTI, A.; FAZIO, M.; VILLARI, M. A study on join operations in mongodb preserving collections data models for future internet applications. *Future Internet*, Multidisciplinary Digital Publishing Institute, v. 11, n. 4, p. 83, 2019.

CHOPADE, M. R. M.; DHAVASE, N. S. Mongodb, couchbase: performance comparison for image dataset. In: IEEE. *2017 2nd International Conference for Convergence in Technology (I2CT)*. [S.l.], 2017. p. 255–258.

CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, ACM, v. 13, n. 6, p. 377–387, 1970.

DATE, C. J. *Introdução a sistemas de bancos de dados*. [S.l.]: Elsevier Brasil, 2004.

DAVOUDIAN, A.; CHEN, L.; LIU, M. A survey on nosql stores. *ACM Computing Surveys (CSUR)*, ACM, v. 51, n. 2, p. 40, 2018.

DAYLEY, B. *Node. js, MongoDB, and AngularJS web development*. [S.l.]: Addison-Wesley Professional, 2014.

FATIMA, H.; WASNIK, K. Comparison of sql, nosql and newsql databases for internet of things. In: IEEE. *2016 IEEE Bombay Section Symposium (IBSS)*. [S.l.], 2016. p. 1–6.

GROVER, P.; JOHARI, R. Mvm: Mysql versus mongodb. In: SPRINGER. *Proceedings of Fifth International Conference on Soft Computing for Problem Solving*. [S.l.], 2016. p. 899–909.

HOLMES, S. *Mongoose for Application Development*. [S.l.]: Packt Publishing Ltd, 2013.

MACHADO, F. N. R. *Banco de dados projeto e implementação*. [S.l.]: Saraiva Educação SA, 2020.

MARDAN, A. Intro to mongodb. In: *Full Stack JavaScript*. [S.l.]: Springer, 2018. p. 239–256.

PANDEY, R. Performance benchmarking and comparison of cloud-based databases mongodb (nosql) vs mysql (relational) using ycsb. 09 2020.

PANDIAN, P. Building node. js rest api with tdd approach: 10 steps complete guide for node. js, express. js & mongodb restful service with test-driven development. Independently published, 2018.

RAUTMARE, S.; BHALERAO, D. Mysql and nosql database comparison for iot application. In: IEEE. *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*. [S.l.], 2016. p. 235–238.

SATHEESH, M.; D'MELLO, B. J.; KROL, J. *Web development with MongoDB and NodeJs*. [S.l.]: Packt Publishing Ltd, 2015.

SEO, J. Y.; LEE, D. W.; LEE, H. M. Performance comparison of crud operations in iot based big data computing. *International Journal on Advanced Science Engineering Information Technology*, v. 7, n. 5, 2017.

UZAYR, S. bin; CLOUD, N.; AMBLER, T. Mongoose. In: *JavaScript Frameworks for Modern Web Development*. [S.l.]: Springer, 2019. p. 309–375.

WILSON, E. *MERN Quick Start Guide: Build Web Applications with MongoDB, Express. js, React, and Node*. [S.l.]: Packt Publishing Ltd, 2018.