
**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**

# TÍTULO DO TRABALHO: SUBTÍTULO (se houver)

**ARACATI-CE**
**ANO DE PUBLICAÇÃO**

Emanuel Moraes de Almeida

# TÍTULO DO TRABALHO: SUBTÍTULO (SE HOUVER)

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 (a): Prof. <Título Abreviado> Nome completo

Aracati-CE
ANO DE PUBLICAÇÃO

Emanuel Moraes de Almeida

TÍTULO DO TRABALHO: SUBTÍTULO (SE HOUVER)

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. <Título Abreviado> <Nome completo> (Orientador (a))
<instituição>

---

Prof. <Título Abreviado> <Nome completo> (Orientador (a))
<instituição>

---

Prof. <Título Abreviado> <Nome completo> (Orientador (a))
<instituição>

## DEDICATÓRIA

Aos meus pais.
Aos mestres.

## AGRADECIMENTOS

Agradecimentos aqui.

**RESUMO**

Resumo aqui.

**Palavras-chaves**: Primeira. Segunda. Terceira.

**ABSTRACT**

Abstract here.

**Keywords**: First. Second. Third.

# LISTA DE ILUSTRAÇÕES

# LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| IFCE | Instituto Federal de Educação, Ciência e Tecnologia do Ceará |
| TCC | Trabalho de Conclusão de Curso |

**SUMÁRIO**

# 1 INTRODUÇÃO

A grande demanda pela informatização de processos, que antes eram desenvolvidos completamente por humanos, tem sido bastante crescente. Isto vem estimulando a criação de novas ferramentas para facilitar o desenvolvimento de aplicações que automatizam tais processos.

Uma aplicação pode ser desenvolvida em diversas plataformas. Uma das plataformas mais utilizadas atualmente é a web, que permite a comunicação entre dispositivos localizados no mundo inteiro que se conectam a ela.

Dentre as características mais comuns no desenvolvimento das mais diversas aplicações web, estão 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.

Uma das abordagens mais antigas de armazenamento de dados é a utilização de arquivos de texto, mas com o passar do tempo, gerenciar tais dados dessa maneira se tornou complexo e lento. Motivados por esta problemática, os Banco de Dados e os Sistema de Gerenciamento de Banco de Dados (SGBD) foram criados.

Um Banco de Dados é um sistema que possibilita o registro de dados seguindo uma determinada estrutura de armazenamento. Um SGBD é um software capaz de manipular um determinado banco de dados, disponibilizando para o usuário ferramentas de CRUD. Um SGBD realiza todas as operações e tratamentos necessários, fornecendo um *endpoint* de inserção de comandos para o gerenciamento mais simples, consistente e performático da manipulação de dados.

Vários tipos diferentes de Banco de Dados foram criados ao longo do tempo, mas os Bancos de Dados atualmente mais utilizados pelas empresas seguem o modelo relacional. Tal modelo define uma estrutura de dados normalizada baseado em tabelas que se relacionam entre si. Tal modelo demonstrou-se consistente e performático, popularizando-se rapidamente. A linguagem padrão utilizada por tais bancos é o SQL (Structured Query Languagem).

O modelo Relacional foi desenvolvido visando a imposição de alguns limites. Tais limites são definidos pelas regras de normalização e foram motivados para:

- Otimizar a quantidade de dados armazenados;
- Otimizar a atualização de registros;

- Criar regras na própria estrutura de armazenamento, a fim de dificultar a inconsistência de dados.

Com o passar do tempo, os dispositivos começaram a aumentar sua capacidade de armazenamento drasticamente. Tal avanço tecnológico estimulou a criação de aplicações que necessitam armazenar uma quantidade cada vez maior de dados. Mediante tal cenário, pôde-se observar que os limites impostos pela normalização vêm demonstrando ser uma barreira tecnológica que dificulta a criação de aplicações altamente escaláveis, disponíveis e consistentes.

Baseado nessa problemática, começaram a surgir novos modelos de armazenamento de dados, objetivando melhorar a performance de aplicações cujo o atendimento de suas exigências fosse muito caro, complexo ou inviável para o modelo de banco de dados Relacional.

Os bancos que estruturam seus dados usando abordagens não relacionais ou parcialmente relacionais são denominados de NoSQL (Not Only SQL). Vários tipos diferentes de banco de dados NoSQL foram surgindo, como por exemplo, os bancos baseados em células de tuplas, grafos, chave-valor e documento.

Atualmente, um dos banco de dados NoSQL mais utilizados 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 desnormalizada.

A desnormalização permite uma maior flexibilização da estrutura de armazenamento, possibilitando a utilização de diversas técnicas específicas para vários tipos de aplicações. Apesar dos benefícios da desnormalização, existem possíveis problemas que podem decorrer mediante seu uso.

Os limites impostos pela normalização tentam garantir que um determinado valor somente precisará ser alterado em um único lugar. Por causa desta característica, independente da complexidade do banco, a atualização de um valor, no geral, é bastante rápida. Em contra partida, 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, aplicações que usam, por exemplo, o *MongoDB*, muitas vezes, mesclam estratégias de normalização e desnormalização. Com o *MongoDB*, os dados podem estar normalizados, parcialmente normalizados ou desnormalizados.

O *MongoDB* foi projetado para que os dados possam estar desnormalizados e ele oferece um suporte muito bom para isto, mas ele, atualmente, não oferece um

suporte tão bom quanto em bancos relacionais para dados normalizados. Por causa disto, existem operações em dados normalizados que seriam mais simples de serem realizados usando a linguagem SQL, mas que se tornam mais difíceis de serem realizados no *MongoDB*. Dentro de um ambiente de desenvolvimento web com *MongoDB*, tais problemáticas podem ser frequentemente encontradas.

O desenvolvimento de uma aplicação web com *MongoDB* pode exigir um trabalho extra, em comparação com bancos relacionais, pois, as vezes, será necessário realizar uniões e buscas em dados normalizados ou parcialmente normalizados de forma não muito intuitiva. Além disto, por causa da alta flexibilidade na estruturação dos dados, as diferentes formas como os dados podem ser armazenados exigem tratamentos muitos distintos na hora de implementar funcionalidades comuns para o desenvolvimento web.

Dessa forma, se a estrutura dos dados mudar durante o desenvolvimento, as operações de busca e união precisarão ser alteradas. Ou seja, haverão mudanças em todas as funcionalidades que estiverem referenciando os dados reestruturados.

Além dos problemas decorrentes do uso do *MongoDB*, uma aplicação web exige a implementação de um conjunto de funcionalidades que, apesar de serem comuns para este tipo de aplicação, podem exigir muito trabalho para serem desenvolvidas.

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

Um framework é um conjunto de códigos fonte que oferece camadas de abstração para facilitar o desenvolvimento de um conjunto de funcionalidades disponibilizadas por ele.

O *Alpha Restful* abstrai a implementação de diversas funcionalidades que precisariam ser desenvolvidas manualmente. O desenvolvimento feito usando tal framework torna-se mais simples, pois ele obtém informações sobre a forma como os dados estão armazenados. Essas informações são utilizadas para que diversas funcionalidades sejam abstraídas em implementações mais simples e direcionadas ao que realmente se deseja fazer.

## 1.1 Objetivos

### *1.1.1 Objetivo Geral*

### *1.1.2 Objetivos Específicos*

## 1.2 Organização do 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 (CODD, 1970). Através desse 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.

Cada linha de uma tabela (tupla) deve representar a abstração de um objeto do sistema, sendo que cada célula da tupla 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 tupla a qual ela está contida. A chave primária é um valor único para a tabela, geralmente representado com um tipo numérico.

Caso seja necessário armazenar os dados contidos em uma tupla de outra relação, basta, em alguma célula da tupla, armazenar uma chave estrangeira de outra tabela. Uma chave estrangeira é a chave primária de uma tupla de outra relação. Ao armazenar como chave estrangeira a chave primária de outra tabela, é realizado um link lógico, simbolizando que todos os dados representados por esta chave estrangeira também estão contidos dentro desta mesma tupla.

Como exemplo, pode-se abstrair um sistema que precisa armazenar várias pessoas, na qual cada uma possui um nome e uma idade. Neste exemplo pode-se representar os dados da seguinte forma:

| Pessoa | | |
|---|---|---|
| Chave Primária | Nome | Idade |
| 1 | Emanuel | 21 |
| 2 | Eduardo | 40 |

Nesse exemplo, pode-se também desejar armazenar várias casas, na qual uma casa terá a sua rua e número. Neste caso é possível criar uma nova relação para a entidade (abstração de um tipo de objeto em nosso sistema) Casa:

| Casa | | |
|---|---|---|
| Chave Primária | Rua | Número |
| 10 | Rua Castelo Branco | 1B |
| 20 | Rua Pompel | 1089 |

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. Para representar essa situação pode-se modelar os dados da seguinte forma:

| Pessoa | | |
|---|---|---|
| Chave Primária | Nome | Idade |
| 1 | Emanuel | 21 |
| 2 | Eduardo | 40 |

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

Observando os dados exemplo apresentados acima, a pessoa Emanuel possui as duas casas armazenados no banco de dados, enquanto o Eduardo não possui nenhuma casa.

### *2.1.1 Normalização*

O modelo Relacional possui algumas regras que o desenvolvedor deverá observar ao modelar as estruturas das relações de seu banco. Dentre essas regras estão contidas as regras de normalização.

A normalização pode ser definida como um conjunto de regras que visam, principalmente, organizar os dados de forma a reduzir a redundância e otimizar a quantidade de dados armazenados.

Tal normalização pode ser dividida em seis formas normais. A seguir, serão explicadas as três primeiras formas normais, por serem as mais importantes e serem a de 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 tuplas, 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:

| Pessoa | | |
|---|---|---|
| Chave Primária | Nome | Idade |
| 1 | Emanuel | 21 |
| 2 | Eduardo | 40 |

| RelacionamentoPessoaCasa | | |
|---|---|---|
| Chave Primária do Relacionamento | Chave Estrangeira de Pessoa | Chave Estrangeira de Casa |
| 1 | 1 | 10 |
| 2 | 1 | 20 |
| 3 | 2 | 10 |

| Casa | | |
|---|---|---|
| Chave Primária | Rua | Número |
| 10 | Rua Castelo Branco | 1B |
| 20 | Rua Pompel | 1089 |

No exemplo de modelagem acima, o Emanuel possui as duas casas e o Eduardo possui a casa de número 1B.

#### *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 disso 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 e 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:

| RelacionamentoPessoaCasa | | | | | |
|---|---|---|---|---|---|
| Chave Primária do Relacionamento | Chave Estrangeira de Pessoa | Chave Estrangeira de Casa | Valor Mensal | Quantidade de Meses | Total a Pagar |
| 1 | 1 | 10 | 200 | 12 | 2400 |
| 2 | 1 | 20 | 400 | 24 | 9600 |
| 3 | 2 | 10 | 300 | 8 | 2400 |

Neste exemplo, O 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.

Para este caso, esta relação não está na terceira forma normal, pois o atributo *Total a Pagar* é dependente do *Valor Mensal* e da *Quantidade de Meses*. O total a pagar 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 fica de acordo com a Terceira Forma Normal é necessário remover a coluna *Total a Pagar*:

| RelacionamentoPessoaCasa | | | | |
|---|---|---|---|---|
| Chave Primária do Relacionamento | Chave Estrangeira de Pessoa | Chave Estrangeira de Casa | Valor Mensal | Quantidade de Meses |
| 1 | 1 | 10 | 200 | 12 |
| 2 | 1 | 20 | 400 | 24 |
| 3 | 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*

Existem situações onde as tabelas precisam ser unidas em uma única tabela desnormalizada, para a realização de buscas mais complexas, ou para simplesmente agrupar os dados da maneira como eles devem ser consumidos nas aplicações que os irão utilizar.

Como exemplo disto, pode-se unir as três tabelas exemplo apresentadas anteriormente, apenas para a pessoa de nome *Emanuel*. Após esta junção a tabela resultante seria algo como:

| Junção das Três Tabelas | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Chave Primária de Pessoa | Nome | Idade | Chave Primária do Relacionamento | Chave Estrangeira de Pessoa | Chave Estrangeira de Casa | Valor Mensal | Quantidade de Meses | Chave Primária de Casa | Rua | Número |
| 1 | Emanuel | 21 | 1 | 1 | 10 | 200 | 12 | 10 | Rua Castelo Branco | 1B |
| 1 | Emanuel | 21 | 2 | 1 | 20 | 400 | 24 | 20 | Rua Pompel | 1089 |

Neste caso, algumas colunas do relacionamento de Pessoa com Casa são desnecessárias e no comando de junção das tabelas pode-se omití-las. Neste caso a tabela de junção seria algo como:

| Junção das Três Tabelas | | | | | | | |
|---|---|---|---|---|---|---|---|
| Chave Primária de Pessoa | Nome | Idade | Valor Mensal | Quantidade de Meses | Chave Primária de Casa | Rua | Número |
| 1 | Emanuel | 21 | 200 | 12 | 10 | Rua Castelo Branco | 1B |
| 1 | Emanuel | 21 | 400 | 24 | 20 | Rua Pompel | 1089 |

A ideia dos dados estarem separados é de tentar garantir que os dados fiquem consistentes na hora de atualizar os dados existentes, além de tentar garantir uma velocidade maior na hora de atualizar valores, 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, envolvendo a junção de centenas de tabelas, deixando essas operações de junção muito lentas. Existem algumas técnicas e estratégias para deixar essas consultas mais rápidas, mas mesmo com essas estratégias, haverá aplicações na qual a exigência de alta performance, escalabilidade, disponibilidade e consistência tornará inviável e 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, garante que os dados sejam armazenados utilizando pouco espaço de armazenamento;
- Por causa da normalização, a atualização de registros são, geralmente, bastante rápidas.

### *2.1.4 Desvantagens do Modelo Relacional*

- O modelo de dados relacionais possui recursos de modelagem muito limitados;
- O mapeamento de objetos com operação de junção, muitas vezes são caras;
- As demandas recentes de armazenamento e consulta de uma grande quantidade de dados revelaram várias deficiências do relacionamento relacional tradicional;
- Diversos tipos de aplicações com altos requisitos de escalabilidade, disponibilidade e consistência podem se tornam caras ou inviáveis;
- 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 para um bom funcionamento do SGBD.

## 2.2 Modelo de Banco de Dados NoSQL com *MongoDB*

Desde o início dos anos 2000, os avanços na tecnologia web, resultaram na explosão repentina de dados estruturados, semi-estruturados e não estruturados por aplicativos de escopo global. Tais aplicações geralmente exigem uma escalabilidade horizontal, adaptar-se às enormes quantidades de dados e à taxa crescente de processamento de consultas (DAVOUDIAN; CHEN; LIU, 2018).

A alta disponibilidade, a baixa tolerância a falhas para responder aos clientes, confiabilidade de transações, o suporte a dados altamente consistentes e a manutenção de schemas de banco de dados com baixo custo de evolução do sistema são requisitos que se tornam muitos difíceis ou inatingíveis nos sistemas tradicionais de banco de dados Relacional (DAVOUDIAN; CHEN; LIU, 2018).

Além desse motivo, a ampliação de sistemas exige uma movimentação de servidores autônomos com hardware aprimorados, sendo um processo caro e causa uma indisponibilidade significativa a cada movimentação. Sistemas baseados em modelos Relacionais exigem uma complexidade e sobrecarga maiores para se juntar dados distribuídos normalizados (DAVOUDIAN; CHEN; LIU, 2018).

De acordo com publicações recentes, os requisitos acima podem ser resolvidos compensando ou sacrificando outros requisitos não tão necessários para a aplicação.

Os modelos de banco de dados NoSQL tornaram-se uma tendência emergente de armazenamento de dados não relacional, que visam satisfazer os requisitos de alta disponibilidade e escalabilidade de aplicações de âmbito global (DAVOUDIAN; CHEN; LIU, 2018).

Um modelo de Banco de Dados NoSQL é caracterizado pala utilização de um modelo não relacional ou parcialmente relacional para o armazenamento de dados. Vários modelos NoSQL foram criados visando esses princípios acima descritos. Para os propósitos deste trabalho, a próxima seção utilizará 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.

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 BJSON (*Binary* JSON).

### *2.2.1 O Formato JSON*

O JSON é um formato de representação de dados chave-valor. A chave é o nome do atributo (caracterísica) pertencente ao objeto na qual o JSON representa. Uma chave é utilizada para representar o valor (valor da característica do objeto) a ela associada.

O valor associado a uma chave pode ser do tipo textual (entre aspas), tipo 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).

A seguir é apresentado um exemplo de representação de uma lista de pessoas com suas respectivas casas:

**Listing 2.1 – JSON Representando Uma Lista de Pessoas Com Suas Casas**

```
[{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
},
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

Observa-se que a representação de um objeto é dada entre chaves (*{}*) e, dentro dessas chaves, são definidos os nomes dos atributos do objeto 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 (*,*).

No exemplo apresentado acima, está sendo representado uma lista de objetos. Para a representação desta lista, as pessoas (JSON) são envolvidas entre col-

chetes. 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, bastando para isso 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 BJSON (*Binary JSON*). O BJSON é quase idêntico ao JSON, sendo qua a diferença é apenas a adição de novos tipos, como um tipo para a representação de datas e um tipo para a representação de um identificador para o documento (o *ObjectId*).

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. Cada documento possui, em seu interior, um BJSON (não uma lista, mas apenas um único BJSON).

Seguindo essa abordagem, para, por exemplo, armazenarmos várias pessoas em nosso banco, precisa ser criado uma coleção que irá armazenar vários documentos, sendo cada documento uma representação BJSON de uma pessoa diferente. Cada documento terá um identificador único para a coleção (*_id*) que irá representar todo o documento. Através deste identificador, pode-se fazer relações entre documentos. Tal identificador pode ser de qualquer tipo, porém o tipo recomentado é o *ObjectId*. A seguir é listado algumas vantagens do uso do *ObjectId*:

- O *ObjectId* é formado por 12 bytes, sendo os quatro primeiros refletidos no timestamp de quando o documento foi criado e possui uma probabilidade muito alta de ser único
- É possível obter a data de criação de um documento
- Odenar pelo *_id* com o tipo *ObjectId* é equivalente a ordenar os documentos pela data de criação (para os documentos criados no mesmo segundo não é garantido nenhuma ordem)
- O campo *_id* é automaticamente criado caso não seja informado

A seguir será exemplificado um BJSON que poderia ser armazenado em um documento do *MongoDB*. Para ilustrar a flexibilidade do valor do *_id*, o identificador das entidades serão armazenados por meio de um valor numérico, porém os identificadores dos BJSON "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.

**Listing 2.2 – Estrutura de Dados da Pessoa *Emanuel***

```
{
    "_id": 1,
    "nome": "Emanuel",
    "idade": 21,
    "casas": [
        {
            _id: ObjectId("5e66c969d5d7cc24c0850854"),
            "rua": "Rua Castelo Branco",
            "numero": "1B",
            "valorMensal": 200,
            "quantidadeMeses": 12
        },
        {
            _id: ObjectId("5e66c9740efbf528b05a7537"),
            "rua": "Rua Pompel",
            "numero": 1089,
            "valorMensal": 400,
            "quantidadeMeses": 24
        }
    ]
}
```

Pode-se observar que cada BJSON possui seu próprio identificador, até mesmo o BJSON que representa uma casa. O identificador do BJSON de uma casa não é o identificador de outro documento, pois neste exemplo os dados estão desnormalizados e, por enquanto, para o nosso exemplo, não existe outra coleção além da coleção de Pessoa.

### *2.2.3 Desnormalização*

A desnormalização é o armazenamento de dados sem a preocupação de seguir as formas normais. No exemplo apresentado, a coleção de Casa não existe e

as casas estão armazenadas dentro do mesmo BJSON 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 não existe a necessidade de realizar junção de vários documentos, tendo um ganho substancial de performance. No exemplo apresentado do modelo Relacional, foi necessário realizar buscas e uniões de tabelas para obter uma tabela 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, bastando apenas buscar o documento 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 este caso.

### *2.2.4 Limites 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 disso, 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 completa não for uma opção, torna-se necessário a criação de uma nova coleção para normalizar esses dados. Para exemplificar esta normalização, uma nova coleção deve ser criada para armazenar todas as casas, e 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*.

A seguir é apresentado exemplos de como os dados seriam estruturados me-

diante uma normalização:

**Listing 2.3 – Estrutura de Dados Normalizados de uma Casa**

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

**Listing 2.4 – Estrutura de Dados Normalizados da Pessoa *Eduardo***

```
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": {
        "_id": ObjectId("5e66cad68a1e581e3cf47dfc"),
        "id": 20,
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}
```

Observa-se que no atributo de *casas* existem dois identificadores. O identificador *_id* identifica o GJSON 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) será armazenado no atributo *id*.

É possível padronizar para que o *_id* represente, ao mesmo tempo, o identificador do BJSON (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, pode ser que seja uma boa ideia manter esses dois identificadores.

Neste 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.

### 2.2.5 Desnormalização Parcial

No exemplo apresentado na sessão anterior, 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 parcial, normalizando apenas o atributo *numero* e duplicando nas várias pessoas o valor do atributo de *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.

Pode-se observar que mesmo em situações onde os dados precisam ser atualizados, pode haver situações onde a desnormalização pode ser utilizada, caso as regras de negócio da aplicação garantam que os dados não serão duplicados o suficiente para trazer uma performance inaceitável.

### 2.2.6 Junção de Documentos

Quando se realiza uma normalização, naturalmente poderá haver situações onde os documentos precisem ser unidos. No exemplo de normalização apresentado anteriormente, caso seja necessário unir os documentos da coleção de pessoa com os documentos da coleção de casa, seria obtido algo como:

**Listing 2.5 – 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"),
        "id": 20,
        "rua": "Rua Pompel",
        "numero": "1089",
        "valorMensal": 300,
        "quantidadeMeses": 8
    }
}]
```

A junção de documentos no MongoDB tende a ser mais rápido que as junções

de tabelas no modelo Relacional, pois as buscas por identificadores de documento no MongoDB são otimizadas, além de os atributos de relacionamento em relacionamentos muito para muitos não necessitar de um terceiro documento auxiliar, diminuindo a quantidade de documentos a serem unidos.

### 2.2.7 Vantagens do Banco de Dados MongoDB

- Os modelos de armazenamento de dados são muito flexíveis, sendo que cada documento pode ser estruturado de maneira muito diferente uns dos outros
- Pela flexibilidade da modelagem, o armazenamento pode ser escalonado, podendo alcançar alta disponibilidade e baixo custo
- 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
- Possibilitando a adição de dados relevantes em um mesmo documento e a duplicação dos dados em várias partes, a junção de dados possui um melhor desempenho, alcançando uma melhor velocidade e consulta
- Há um uso inteligente de índices distribuídos, como hashing e caching para acesso a dados e armazenamento
- Os dados podem ser facilmente replicados e particionados horizontalmente em servidores locais e remotos
- Fornece uma estrutura de dados muito similar e familiar com o tratamento de dados de aplicações web

### 2.2.8 Desvantagens do Banco de dados MongoDB

- 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
- Exige uma responsabilidade maior por parte do desenvolvedor de criar uma modelagem de dados que não inviabilize a utilização de algumas operações básicas de manipulação dos dados
- Exige da parte da aplicação mais verificações na hora de alterar e remover dados

- Não trata automaticamente os relacionamentos manuais entre documentos descritos anteriormente, podendo deixar passar (caso o desenvolvedor não se atente) identificadores que apontam para documentos que já não existe mais

  catastróficos, possivelmente irreversíveis a consistência dos dados

- As buscas que precisam ser realizadas em documentos separados e relacionados podem ser mais complexas do que as consultas realizadas no modelo Relacional

# 3 *Alpha Restful*

Assim como descrito anteriormente, o foco do *MongoDB* é o armazenamento de dados desnormalizados, disponibilizando diversas funcionalidades para a sua manipulação. Por essa razão, operações de busca e união de documentos normalizados precisam, as vezes, serem implementadas de forma menos intuitiva e mais trabalhosa, em comparação com o SQL. Visando amenizar os problemas apresentados, foi desenvolvido um framework denominado de *Alpha Restful*. Tal ferramenta foi projetada para a linguagem *JavaScript*, usando o ambiente de execução *Node JS*, utilizando internamente o *MongoDB* e o *Mongoose* (uma biblioteca que implementa algumas funcionalidades extras para o *MongoDB*).

O *Alpha Restful* facilita e automatiza a implementação de 5 funcionalidades que, por causa dos motivos descritos anteriormente, podem ser complexas de serem desenvolvidas no *MongoDB* sem o uso de um framework. Para cada funcionalidade, será exemplicado sua implementação sem o uso do *Alpha Restful* e, posteriormente, com o uso desse framework. Para o detalhamento dessas funcionalidades, será utilizado como base a modelagem de dados exemplificada pelos seguintes documentos normalizados:

**Listing 3.1 – Documento da Pessoa *Eduardo***

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

**Listing 3.2 – 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
        }
    ]
}
```

**Listing 3.3 – Documento da Casa de Número 1B**

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

**Listing 3.4 – Documento da Casa de Número 1089**

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

## 3.1 Junção de Documentos

Assim como foi visto na seção *2.2.6*, as vezes, documentos que foram completamente ou parcialmente 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.

### *3.1.1 $lookup*

O *$lookup* é uma operação disponibilizada de forma oficial pelo *MongoDB*. Tal operação se responsabiliza por unir dois ou mais documentos relacionados. Após tal união, pesquisas podem ser realizadas sobre esses dados e valores podem ser agrupados e ordenados. Para unirmos as coleções de Pessoas e Casas, ignorando os atributos de relacionamento, a seguite operação poderia ser feita:

**Listing 3.5 – Junção de Documentos com *$lookup* 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. Opós essa operação, a variável “UNIAO” conterá o seguinte resultado:

**Listing 3.6 – Junção de Documentos com Omissão**

```
[{
    "_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"
    }]
}]
```

Observa-se que os atributos “valorMensal” e “quantidadeMeses” não encontram-se no resultado da operação feita anteriormente. 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 substituída pelo identificador presente na própria entidade. Essa omisão 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. Uma possível codificação para isso seria:

**Listing 3.7 – Junção de Documentos sem Omissão**

```
let UNIAO = await db.collection("pessoas").aggregate([
    { $unwind: "$casas" },
    { $lookup: {
        from: "casas",
        let: { casas: "$casas" },
        pipeline: [
            { $match: { $expr: {
                $eq: [ "$_id", "$$casas.id" ]
            }}},
            { $addFields: {
                _id: "$$casas._id",
                id: "$$casas.id",
                valorMensal: "$$casas.valorMensal",
                quantidadeMeses: "$$casas.quantidadeMeses"
            }}
        ],
        as: "casas"
    }},
    { $group: {
        _id: {
            _id: "$_id",
            nome: "$nome",
            idade: "$idade"
        },
        casas: { $push: "$casas" }
    }},
    { $project: {
        _id: "$_id._id",
        nome: "$_id.nome",
        idade: "$_id.idade",
        casas: { $reduce: {
            input: "$casas",
            initialValue: [],
            in: { $concatArrays: [ "$$value", "$$this" ] }
        }}
    }}
]).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 tornou-se necessário pois os identificadores das casas estão dentro de uma lista. Se uma pessoa podesse, no máximo, ter uma única casa, uma codificação mais simples poderia ser realizada. Bastaria para isso apenas colocar na opção *"as"* do *$lookup* um outro caminho que não sobrescreveria nenhum atributo já existente. Após a execução do código anterior, a variável "UNIAO" obterá o seguinte resultado:

**Listing 3.8 – Junção de Documentos com *$lookup***

```
[{
    "_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. 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 gerar erros humanos de codificação.

### *3.1.2 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 que o documento *DBRef* reside. No exemplo a qual está sendo tratado, as pessoas registradas no sistema poderiam se relacionar com suas casas da seguinte forma:

**Listing 3.9 – Documento da Pessoa *Eduardo***

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

**Listing 3.10 – Documento da Pessoa *Emanuel***

```
{
    "_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 obervar 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 união de documentos automáticas. Nesse caso, uniões de uniões de documentos poderiam ser feitas automaticamente de forma simples, dependendo da ferramenta que está sendo utilizada para o desenvolvimento. Esses recursos provenientes do *DBRef* não está disponível 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 o identificador da entidade relacionada. 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:

**Listing 3.11 – Documento da Pessoa *Eduardo***

```
{
    "_id": 2,
    "nome": "Eduardo",
    "idade": 40,
    "casas": [{
        "$id": ObjectId("5e66cb581766a2056c48145f"),
        "$ref": "relacionamento_pessoas_casas"
    }]
}
```

**Listing 3.12 – Documento da Pessoa *Emanuel***

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

**Listing 3.13 – Documento do Relacionamento da Pessoa *Eduardo* Com Sua Casa**

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

**Listing 3.14 – Documento do Relacionamento da Pessoa *Emanuel* Com Sua Primeira Casa**

```
{
    "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
    "valorMensal": 200,
    "quantidadeMeses": 12,
    "casa": {
        "$id": 10,
        "$ref": "casas"
    }
}
```

**Listing 3.15 – Documento do Relacionamento da Pessoa *Emanuel* Com Sua Segunda Casa**

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

**Listing 3.16 – Documento da Casa de Número 1B**

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

**Listing 3.17 – Documento da Casa de Número 1089**

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

Esses documentos auxiliares para representar 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 que está sendo utilizada. Se o ambiente de execução utilizado suportar tal estratégia, seria possível relacionar as pessoas com suas casas por meio de uma modelagem parecida com isso:

**Listing 3.18 – Documento da Pessoa *Eduardo***

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

**Listing 3.19 – Documento da Pessoa *Emanuel***

```
{
    "_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"
            }
        }
    ]
}
```

**Listing 3.20 – Documento da Casa de Número 1B**

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

**Listing 3.21 – 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 BJSONs 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*.

### *3.1.3 Populate do Mongoose*

O *Mongoose* é uma biblioteca feita para *Node JS*, que disponibiliza algumas funcionalidades a mais para o *MongoDB*, principalmente relacionadas à modelagem dos dados. Através dessa ferramenta, torna-se possível criar *schemas* para os dados a serem armazenados. Esses *schemas* permitem que a estrutura do BJSON de cada documento seja padronizada e obeceça as regras de estrutura definidos pelo programador. Para o exemplo que está sendo apresentado, os seguintes *schemas* podem ser utilizados:

**Listing 3.22 – 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*. Esse recurso substitui o *DBRef* e o *$lookup* para operações de busca, seguido de união de documentos. A restrição do *populate* em comparação com o *$lookup* é que no *$lookup* é possível realizar buscas sobre os dados dos documentos unidos, enquanto que no *populate* a busca deve ocorrer apenas sobre os dados originais do documento. Se desconsiderarmos essa "limitação", o *populate* é mais simples de se usar do que o *$lookup*, além de disponibilizar novas opções e recursos. Atualmente, o *Mongoose* está disponível apenas para o *Node JS*, que é exatamanete o escopo a qual esse trabalho se propõe em atual. Para que os documentos da coleção de Pessoa e Casa sejam unidos com o *populate*, seria necessário fazer uma codificação parecida com isso:

**Listing 3.23 – União de Documentos Com o *Populate***

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

Com apenas uma única linha, os dois documentos foram unidos, mantendo os

atributos de relacionamento. Após a execução do código anterior, a variável “UNIAO” terá o seguinte resultado:

**Listing 3.24 – 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 esse caso, remodelar a entidade de pessoa, para que o atributo "*id*" seja renomeado para, por exemplo, "casa". Nesse caso, os resultados sairiam com o seguinte formato:

**Listing 3.25 – Junção de Documentos Com o *Populate***

```
[{
    "_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
    }
}]
```

### 3.1.4 União de Documentos de Forma Manual

Caso haja a necessidade de obter um maior controle na união dos documentos, ou se os recursos de união automático não forem satisfatório o suficiente para as necessidades da aplicação, é possível realizar uma união de forma manual. Um exemplo de uma codificação manual de junção de documentos encontra-se a seguir:

**Listing 3.26 – Junção dos Documentos de Pessoa com Casa**

```
1     let pessoas = await db.collection("pessoas")
2         .find({}).toArray();
3
4     for (let p of pessoas) {
5         for (let i = 0; i < p.casas.length; i++) {
6             let c = p.casas[i];
7
8             let casa = (await db.collection("casas").find({
9                 "_id": c.id
10            }).toArray())[0];
11
12            p.casas[i] = {
13                ...casa,
14                ...c
15            };
16        }
17    }
```

Ao final desse código, a variável “pessoas” terá a junção dos documentos da coleção de Pessoa e Casa. Tal variável teria o seguinte resultado:

**Listing 3.27 – 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
    }
}]
```

### *3.1.5 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 de um documento
- Relacionamento inverso de um 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, não só baseado em identificadores, mas também baseado em relacionamentos virtuais ou em outros relacionamentos. No exemplo a qual está sendo trabalhado nessa seção, os schemas e especificações de sincronização das entidades podem ser definidos no *Alpha Restful* com uma codificação parecida com isso:

**Listing 3.28 – Definição de *Schemas* no *Alpha Restful***

```
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
        }
    }
})

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

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

Para a união de documento, o *Alpha Restful* disponibiliza uma opção denominada de *fill*. Tal opção é poderosa, pois ela pode ser utilizada sobre qualquer objeto

já pesquisado ou montado, sendo possível definir, na chamada da função, relacionamentos temporários com outras entidades. Para que os documentos da coleção de Pessoa e Casa sejam unidos com o *Alpha Restful*, uma codificação parecida com isso teria que ser feita:

**Listing 3.29 – União de Documentos Com o *Alpha Restful***

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

A operação de união 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 união. Na versão atual do *Alpha Restful* (0.7.37), o identificador da entidade relacionada precisa ser definido no atributo *id*. Depois de executar o código anterior, a variável “pessoas” obterá o seguintes resultado:

**Listing 3.30 – 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
    }
}]
```

#### *3.1.5.1 Relacionamento Inverso*

Uma das funcionalidades disponibilizadas pelo *Alpha Restful* que, atualmente, não estão diretamente disponíveis nos outros métodos de união 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 unindo nos documentos de casa. Essa operação pode ser feita diretamente no método de *fill*, sem a necessidade de alterar a modelagem da entidade:

**Listing 3.31 – União de Documentos em Relacionamento Inverso**

```
1     let casas = await Casa.model.find({}).exec()
2     casas = await Casa.fill(casas, restful, { sync: {
3         pessoas: {
4             name: "Pessoa",
5             syncronized: "casas",
6             fill: true,
7             jsonIgnoreProperties: "casas"
8         }
9     }})
```

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 atribututo "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 anterior, os documentos de Casa e Pessoa são unidos, mas tendo como base a entidade "Casa". Os documentos unidos estarão presentes na variável "casas" e teria a seguinte estrutura:

**Listing 3.32 – Junção de Documentos em "Casa"**

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

Como na modelagem da entidade de "Casa" o relacionamento com pessoa não foi definido, é possível fazer essa definição na hora de realizar a união. 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 união poderia ocorrer sem a definição do relacionamento na hora da união. Nesse caso, a modelagem de Casa ficaria da seguinte forma:

**Listing 3.33 – Definição do *Schema* de Casa**

```
const Casa = new Entity({
    name: "Casa",
    resource: "casas",
    descriptor: {
        rua: String,
        numero: String
    },
    sync: {
        pessoas: {
            name: "Casa",
            fill: true,
            jsonIgnoreProperties: "casas"
        }
    }
})
```

### *3.1.5.2 Relacionamento Inverso do Relacionamento Inverso*

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

**Listing 3.34 – Definição de *Schemas* no *Alpha Restful***

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

const Pessoa = new Entity({
    name: "Pessoa",
    resource: "pessoas",
    descriptor: {
```

```
9               nome: String,
10              idade: Number,
11              casas: [{
12                  id: Number,
13                  valorMensal: Number,
14                  quantidadeMeses: Number
15              }]
16          },
17          sync: {
18              casas: {
19                  name: "Casa",
20                  fill: true,
21                  jsonIgnoreProperties: "pessoas"
22              },
23              residentes: {
24                  name: "Pessoa",
25                  syncronized: ["casas.pessoas"],
26                  fill: true,
27                  jsonIgnoreProperties: "residentes"
28              }
29          }
30      })
31
32      const Casa = new Entity({
33          name: "Casa",
34          resource: "casas",
35          descriptor: {
36              rua: String,
37              numero: String
38          },
39          sync: {
40              pessoas: {
41                  name: "Pessoa",
42                  syncronized: ["casas"],
43                  fill: true,
44                  jsonIgnoreProperties: "casas"
45              }
46          }
47      })
```

```
48
49      restful.add(Pessoa)
50      restful.add(Casa)
```

Os *schemas* definidos no código anterior criam em Pessoa 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 se unir os documentos de pessoa com casa, havera um atributo no documento de pessoa chamado de “residentes”, que 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 seguinte código:

**Listing 3.35 – União 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 inverso, que também pode se relacionar com um outro relacionamento inverso, a união dedocumentos por meio do *Alpha Restful* é mais poderosa que as outras opções descritas anteriormente. Ao final da execução do código anterior, a variável “pessoas” terá o seguinte resultado:

**Listing 3.36 – Junção de Documentos Com Residentes**

```
1  [{
2      "_id": 1,
3      "nome": "Emanuel",
4      "idade": 21,
5      "casas": [
6          {
7              "_id": ObjectId("5e66cb3a7216361ff05b3b8f"),
8              "id": 10,
9              "rua": "Rua Castelo Branco",
10             "numero": "1B",
11             "valorMensal": 200,
12             "quantidadeMeses": 12
13         },
14         {
15             "_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,
        "quantidadeMeses": 8
    }],
    "residentes'': [
        {
            "id": 1,
            "nome": "Emanuel",
            "idade": 21,
        },
        {
```

```
55              "id": 2,
56              "nome": "Eduardo",
57              "idade": 40,
58          }
59      ]
60  }]
```

## 3.2 Buscas Filtradas em Documentos Relacionados

O *MongoDB* é nativamente capaz de realizar buscas complexas e simples dentro de um mesmo documento, mas a partir do momento que as buscas precisam ser realizadas em documentos separados que estão relacionados entre si, as pesquisas passam a ficar mais difíceis de serem realizadas.

Para, por exemplo, ser feito uma busca pelas pessoas cuja a idade seja igual a 40, poderia ser realizado uma codificação parecido com isso:

**Listing 3.37 – 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?

### *3.2.1 Usando o $lookup*

Para realizar a pesquisa proposta, uma abordagem possível é unir os documentos utilizando o *$lookup*, até o nível que todos os dados estariam no mesmo documento. Após essa união, seria possível fazer a pesquisa, utilizando o parâmetro “textit$match”. A pesquisa exemplo seguinte funciona utilizando essa abordagem.

**Listing 3.38 – Busca em Dados Normalizados Com o *$lookup***

```
1      let RESULTADO = await db.collection("casas").aggregate([
2          { $lookup: {
3              from: "pessoas",
4              localField: "_id",
```

```
            foreignField: "casas.id",
            as: "pessoas"
        }},
        { $unwind: "$pessoas" },
        { $lookup: {
            from: "casas",
            localField: "pessoas.casas.id",
            foreignField: "_id",
            as: "pessoas.casas"
        }},
        { $unwind: "$pessoas.casas" },
        { $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",
```

```
44                  idade: "$_id.pessoas.idade",
45                  casas: "$casas"
46              }
47          }},
48          { $group: {
49              _id: {
50                  _id: "$_id",
51                  rua: "$rua",
52                  numero: "$numero"
53              },
54              pessoas: {
55                  $push: "$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": 21
66          }}
67      ]).toArray()
```

Para a realização de uma pesquisa dessa complexidade, é necessário unir os documentos várias vezes, pois é necessário 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.

### *3.2.2 Pesquisa Manual*

Também é possível realizar a pesquisa proposta de forma manual, subdividindo a pesquisa em pesquisas menores e unindo-as em uma pesquisa que irá obter o resultado esperado. A utilização de tal abordagem para a pesquisa proposta resultaria em uma codificação parecida com isso:

**Listing 3.39 – 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 = await db.collection("casas").find({
11        "_id": { $in: idsCasasPessoasIdade40 }
12    }).toArray();
13
14    let idsCasasPessoasIdade40 =
15    casasPessoasIdade40.map(c => c._id);
16
17    let pessoasCasasPessoasIdade40 = await
18    db.collection("pessoas").find({
19        "casas.id": { $in: idsCasasPessoasIdade40 }
20    }).toArray();
21
22    let idsCasasPessoasCasasPessoasIdade40 =
23    idsCasasPessoasCasasPessoasIdade40.map(p =>
24        p.casas.reduce((a,c) => [...a,c.id], [])
25    ).reduce((a,lid) => [...a, ...lid], []);
26
27    let RESULTADO_DA_PESQUISA = await
28    db.collection("casas").find({
29        "_id": { $in: idsCasasPessoasCasasPessoasIdade40 }
30    }).toArray();
```

Para buscar as casas, na qual existe pelo menos uma pessoa, que esta pessoa possui pelo menos uma casa, na qual esta casa possui pelo menos uma pessoa com idade igual a 40 anos, foi necessário 30 linhas com códigos.

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 essa consulta, quando a consulta usando o *$lookup* e *$match*, possui apenas uma ramificação de filtros encadeados. Os códigos ficariam mais complexo com a adição de outras ramificações de filtros, utilizando-se de outras relações com outros documentos relacionados.

### 3.2.3 Usando o Alpha Restful

Como visto anteriormente, realizar buscas que englobam vários documentos pode ser algo complexo. Para contornar esse 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 esse comportamento, será considerado o exemplo de busca apresentado anteriormente. Para realizar esta pesquisa bastaria executar o seguinte código:

**Listing 3.40 – 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 30 (de forma manual) e 67 (usando *$lookup* com *$match*) linhas, mas com o *Alpha Restful*, apenas 3 linhas foi 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 é uma extensão da sintaxe utilizada pelo *Mongoose*, com a diferença de considerar nas pesquisas os atributos contidos em outros documentos relacionados.

## 3.3 Remoção em Cascata de Documentos Relacionados

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 essa pessoa 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 manualmente buscar por todas as casas pertencentes a esta pessoa e removê-las. O problema dessa abordagem manual é que uma coleção pode possuir vários relacionamentos com outros documentos, deixando o código cada vez mais complexo.

Pensando nisso, o *Alpha Restful* disponibiliza uma opção na sincronização das entidades (*sync*), que faz exatamente essa funcionalidade. Para garantir esse comportamento, é necessário apenas informar a opção *deleteCascade* no atributo da entidade a qual deseja-se que seja removida automaticamente:

**Listing 3.41 – Modelagem de "Casa" com *deleteCascade***

```
1    const Casa = new Entity({
2        name: "Casa",
3        // ...
4        sync: {
5            pessoas: {
6                name: "Pessoa",
7                syncronized: ["casas"]
8                fill: true,
9                jsonIgnoreProperties: "casas",
10               deleteCascade: true
11           },
12           // ...
13       }
14   })
```

## 3.4 Relação de Dependência Entre os Documentos

No exemplo a qual está sendo abordado, se uma casa não poder 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 essa regra se repetisse para outros documentos.

Para que essa 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 pessoa não poder ser removida, caso possua alguma casa, bastaria apenas informar a opção *required* no atributo da entidade a qual deseja-se criar um relacionamento de dependência (Pessoa):

**Listing 3.42 – Modelagem de Pessoa com *required***

```
1     const Pessoa = new Entity({
2         name: "Pessoa",
3         // ...
4         sync: {
5             casas: {
6                 name: "Casa",
7                 fill: true,
8                 jsonIgnoreProperties: "pessoas",
9                 required: true
10            },
11            // ...
12        }
13    })
```

## 3.5 Identificadores de Documentos Relacionados Apontando para Lixo

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. Isso 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 “Pessoa” apontará para lugar algum, pois a casa a qual tais identificadores em Pessoa apontam, já não existe mais.

Para que esse problema seja contornado, sem o uso de um *framework*, é necessário que antes que qualquer entidade seja removida, seja realizado 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 com-

portamento. O *Alpha Restful* já resolve esse problema automaticamente (bastando apenas realizar o procedimento descrito na seção 3.6). Nenhuma opção precisa ser habilitada na modelagem para que esse problema seja mitigado.

## 3.6 *deleteSync*

Para que as funcionalidades relacionadas à remoção de entidades no *Alpha Restful* possam acontecer (seções 3.3, 3.4 e 3.5), é necessário que o método *restful.deleteSync* seja chamado antes que qualquer entidade ser removida. Se, por exemplo, uma casa tiver que ser removida, o seguinte código deve ser executado antes da remoção:

**Listing 3.43 – Antes de Remover Uma Casa**

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

## 3.7 Considerações Finais Sobre o *Framework*

O *Alpha Restful* é um framework que tem uma proposta ousada. Um de seus principais objetivos é de fazer com que a escolha de se normalizar ou desnormalizar os dados no *MongoDB*, seja mais como uma questão de regra de negócio, do que como uma questão estrutural profunda na forma como entidades são modeladas e buscadas. Se essa proposta fosse alcançada, a facilidade que já existe para se desnormalizar os dados no *MongoDB*, também existiria para normalizá-los. Isso eliminaria um dos obstáculos que atualmente encontram-se na utilização do banco de dados: a normalização de dados do *MongoDB* que, como foi demonstrado anteriormente, as vezes precisa ser feita em, pelo menos, parte dos dados armazenados, pode ser mais complexo que na desnormalização.

Como foi demonstrado, esse objetivo pôde ser atingido parcialmente na versão atualmente feita do *framework* (0.7.37). As pesquisas normalizadas usando o *Alpha Restful*, são tão simples de serem realizadas quanto aquelas que acessam dados desnormalizados. A união de documentos pode ser feita de forma simples, com um recurso não disponíveis nas outras ferramentas utilizadas: o relacionamento inverso.

Uma das coisas que precisam ficar claras sobre esse *framework* é de que ele não está completo. A versão atualmente feita está em versão *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 funcionando e foram muito bem testadas. O motivo se denomiar 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 importantes não poderam ser desenvolvidas ainda.

Depois que a versão atual do *Alpha Restful* foi desenvolvida, foi observado que o *framework* pode ser melhor otimizado e que pode propor algo mais ousado. Por causa disso, o *Alpha Restful* será completamente refeito. Isso será necessário para que uma nova proposta de desenvolvimento possa ser feita.

Dentre todas as funcionalidades do *framework*, as 5 funcionalidades apresentadas nesse trabalho (seções 3.1, 3.2, 3.3, 3.4 e 3.5) são as mais relevantes para o uso do *MongoDB*. Outras funcionalidades também foram implementadas, porém seu escopo de atuação está mais intimamente relacionado à web. Como o escopo desse trabalho é fazer uma análise mais próxima do *MongoDB*, tais funcionalidades foram omitidas desse documento. Apesar disso, quase todas as funcionalidades desenvolvidas estão disponíveis em um guia de uso gratuito e online, disponiblizado pelo *link* https://www.npmjs.com/package/alpha-restful. Por meio desse mesmo *link*, é possível baixar a ferramenta e utilizá-la. Um dos resultados relevantes que pode ser encontrado nesse link é de que até agora (24/03/2020), mais de 6.062 *downloads* foram feitos, desde o seu lençamento na internet (24/02/2019).

# 4 RESULTADOS

# 5 CONCLUSÃO

# REFERÊNCIAS

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. Citado na página 17.

DAVOUDIAN, A.; CHEN, L.; LIU, M. A survey on nosql stores. *ACM Computing Surveys (CSUR)*, ACM, v. 51, n. 2, p. 40, 2018. Citado 2 vezes nas páginas 17 e 23.

# Apêndice

# APÊNDICE A - TÍTULO

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

# APÊNDICE B - TÍTULO

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

# Anexos

# ANEXO A - TÍTULO

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

# ANEXO B - TÍTULO

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.