# Capítulo 3

# Projeto e ajuste de desempenho de bancos de dados NoSQL

Arlino Magalhães, Francisco Imperes, Manoel Melo

***Abstract***

*Many contemporary applications need to store and process large amounts of data, often in real-time, to make information retrieval feasible. However, traditional storage systems have yet to keep pace with this demand. An alternative to this problem has been using NoSQL databases due to their high availability, scalability, and flexibility for managing large amounts of data. This mini-course aims to present the main concepts of NoSQL technology. Additionally, it shows how to evaluate whether this technology is appropriate for the database design of a particular system. Furthermore, the mini-course also discusses some strategies for tuning the performance of NoSQL databases.*

***Resumo***

*Muitas sistemas contemporâneas tem tido a necessidade de armazenar e processar grandes quantidades de dados, muitas vezes em tempo real, a fim de tornar a recuperação da informação viável. Entretanto, os sistemas de armazenamento tradicionais não tem acompanhado essa demanda. Uma alternativa para esse problema tem sido a utilização de bancos de dados NoSQL devido às suas características como alta disponibilidade, escalabilidade e flexibilidade para gerenciar grandes quantidades de dados. Esse minicurso visa apresentar os principais conceitos da tecnologia NoSQL. Além disso, ele mostra como avaliar se essa tecnologia é apropriada para o projeto de banco de dados de um determinado sistema. Adicionalmente, o minicurso também discute algumas estratégias para o ajuste de desempenho de bancos de dados NoSQL.*

## 3.1. Introdução

Os Sistemas de Gerenciamento de Banco de Dados (SGBDs) relacionais têm sido a principal opção de armazenamento de dados em sistemas de informação, tanto

grandes como pequenos. Contudo, muitas sistemas contemporâneos têm necessitado de um fluxo massivo de dados tão grande que os sistemas relacionais não conseguem gerenciar. Os SGBDs NoSQL têm se mostrado como uma alternativa para sistemas que gerenciam grande volume de dados e que necessitam de desempenho escalável, alta disponibilidade e resiliência [Davoudian et al. 2018, Magalhães et al. 2021].

NoSQL é uma abreviação do termo em inglês *"não apenas SQL"*. Contudo, esse termo não define esses SGBDs de forma clara. O termo NoSQL refere-se a alguns bancos de dados não relacionais que possuem algumas características, como: dados sem esquema, execução em *clusters* e tolerância à inconsistência. Alguns exemplos desses sistemas são o Redis [Redis Database 2024], MongoDB [MongoDB Database 2024], Cassandra [Cassandra Database 2024] e Neo4J [Neo4J Database 2024].

É importante ressaltar que os SGBDs relacionais são uma ferramenta poderosa e uma opção confiável que ainda continuará sendo utilizada no desenvolvimento de sistemas por muito tempo, talvez décadas. Entretanto, NoSQL tem se mostrado como uma solução mais eficiente para o gerenciamento de grandes quantidades de dados. Nesse sentido, muitos sistemas contemporâneos têm se utilizado de uma persistência poligloto que faz o uso de várias tecnologias de gerenciamento de dados. Consequentemente, os arquitetos devem se familiarizar com essas tecnologias para avaliar quais delas podem ser utilizadas para diferentes necessidades [Davoudian et al. 2018, Magalhães et al. 2023].

Cada projeto tem suas peculiaridades e não há um guia preciso de como escolher o armazenamento de dados ótimo. Embora a tecnologia NoSQL exista há bastante tempo, o seu uso adequado em projetos de bancos de dados não é amplamente conhecido. Além disso, a maioria dos livros e cursos de bancos de dados foca no projeto relacional, que é significativamente diferente do projeto NoSQL. Enquanto o projeto relacional se baseie em tabelas, relacionamentos e normalização, o projeto NoSQL leva em consideração agregados, distribuição dos dados e até desnormalização dos dados. Além disso, em muitos casos, sistemas NoSQL permitem uma consistência mais relaxada, ou seja, dados inconsistentes podem ser tolerados em algum nível [Davoudian et al. 2018, Magalhães et al. 2018].

Esse minicurso discute ferramentas relacionados a tecnologia NoSQL, como produtos, linguagens de acesso, manipulação e processamento dos dados. Porém, o principal foco do curso é fornecer uma base teórica e prática suficiente para julgar se a tecnologia NoSQL é adequada a um projeto. O minicurso utiliza uma amostra representativa dos SGBDs NoSQL, um representante de cada uma das quatro categorias existentes: chave-valor, documentos, colunar e grafos. Embora sejam utilizados exemplos específicos, a maioria da discussão pode ser aplicada e reproduzida facilmente em outros SGBDs. Além disso, todos os códigos das atividades práticas abordadas no curso e uma imagem no VirtualBox do sistema operacional Linux com todos os *softwares* necessário instalados para o curso estão disponíveis para *download*[1].

[1] https://github.com/ArlinoMagalhaes/curso-nosql

Após a implantação de um sistema, o gerenciamento de seu desempenho é necessário a fim de fornecer a continuidade do serviço e, se possível, aprimorar o seu desempenho [Mohammad 2016]. Assim, esse minicurso também provê noções básicas de ajuste de desempenho de bancos de dados NoSQL, tais como: análise de fatores que influenciam o desempenho, análise de tempo de resposta, avaliação de operações e ajuste de consultas.

O restante minicurso está organizado como segue. A Seção 3.2 apresenta brevemente uma revisão de bancos de dados, focando na diferença entre SGBDs relacionais e NoSQL. A Seção 3.3 cobre aspectos relacionados a tecnologia NoSQL, incluindo modelos de dados chave-valor, orientados a documentos, colunar e orientados a grafos. A Seção 3.4 descreve como projetar um sistema de banco de dados NoSQL. O ajuste de desempenho em bancos de dados NoSQL é tratado na Seção 3.5. Por fim, as conclusões são apresentadas na Seção 3.6.

## 3.2. Revisão de bancos de dados

Desde o surgimento dos primeiros computadores, houve a necessidade de armazenar e manipular dados. As primeiras estruturas de bancos de dados surgiram na década de 60, tais como o banco de dados hierárquico e o em rede. Porém, esses primeiros sistemas eram caros e difíceis de utilizar. Assim, houve bastante investimento na área de pesquisa de banco de dados. Na década de 80, o barateamento de *hardware/-software* fez o SGBD (Sistema de Gerenciamento de Banco de Dados) relacional se tornar a opção padrão para bancos de dados. A popularidade do banco de dados relacional se deve também a sua facilidade em projetar, armazenar e recuperar dados. Atualmente o SGBD relacional é um dos principais componentes nos sistemas de informação [Ramakrishnan and Gehrke 2003, Elmasri and Navathe 2000, Magalhães et al. 2021].

Essa seção faz uma breve revisão dos principais conceitos dos SGBDs relacionais, focando nas características que fizeram esses sistemas se tornarem a escolha padrão para armazenamento de grandes volumes de dados, especialmente no mundo de aplicativos corporativos. Além disso, essa seção discute as limitações desses sistemas que levaram a procura por alternativas de armazenamento de dados, como os bancos de dados NoSQL. Leitores conhecedores desses conceitos podem pular essa seção.

### 3.2.1. Bancos de dados relacionais

Os SGBDs relacionais oferecem aos usuários/sistemas processos de validação, verificação, segurança e garantias de integridade dos dados. Para isso, esses sistemas são projetados em componentes, como armazenamento, controle de concorrência, otimização de consultas, indexação e recuperação após falhas. Esses componentes facilitam a construção de sistemas, visto que os desenvolvedores não precisam se preocupar com o processo de armazenamento de dados, possibilitando que eles possam focar exclusivamente no sistema [Ramakrishnan and Gehrke 2003, Elmasri and Navathe 2000].

Os SGBDs relacionais utilizam o modelo relacional. Esse modelo organiza

os dados em estruturas de relações (tabelas) e tuplas (linhas). Uma tabela é um conjunto de linhas e uma linha é um conjunto de valores. Cada linha (registro) representa um objeto da entidade representada pela tabela. Cada registro possui um ID (chave) exclusivo que o diferencia entre todos os outros registros da entidade. A tabela possui colunas que são os atributos dos valores de cada linha. A Figura 3.1 ilustra uma tabela que representa uma entidade de empregados de uma empresa [Heuser 2009].

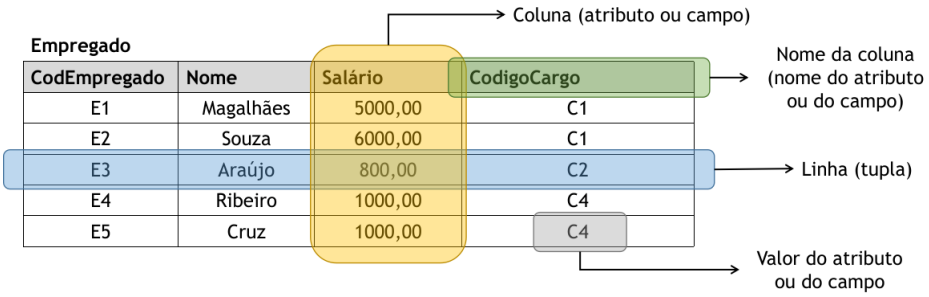



**Figura 3.1. Tabela de banco de dados relacional [Heuser 2009].**

O modelo relacional utiliza comumente o processo de Normalização cujo objetivo é, principalmente, organizar o projeto de banco de dados para reduzir a redundância de dados e aumentar a integridade de dados e o desempenho do sistema. Para isso, a normalização aplica algumas regras sobre as tabelas do banco de dados. Uma abordagem básica da normalização consiste na separação dos dados referentes a elementos distintos em tabelas distintas associadas através da utilização das chaves [Heuser 2009].

O modelo relacional utiliza uma linguagem padrão para definição, manipulação e consulta de dados, a SQL (*Structured Query Language*). SQL é uma linguagem declarativa, ou seja, ela descreve o que uma instrução deve fazer e não a maneira como a instrução deve ser executada. O SGBD fica responsável por escolher a estratégia mais eficiente para execução das instruções SQL. Assim, o desenvolvedor fica livre do trabalho complexo de acesso ao armazenamento, podendo se concentrar mais nas aplicações. A SQL é inspirada em álgebra relacional que é uma linguagem de consulta formal que possui uma coleção de operações de alto nível sobre relações ou conjuntos. Sua simplicidade e alto poder de expressão fizeram de SQL a linguagem de consulta mais utilizada em bancos de dados [Imielinski and Jr. 1984].

Os SGBDs relacionais utilizam o modelo ACID para garantir consistência dos dados durante o processamento das transações. Uma transação é um conjunto de operações submetidas ao banco de dados, por exemplo, inserir, modificar e excluir linhas de tabelas. ACID é um acrônimo derivado da primeira letra das suas quatro principais propriedades: atomicidade, consistência, isolamento e durabilidade. A atomicidade faz uma transação indivisível, ou seja, todas as operações de uma transação devem ser executadas apropriadamente ou nenhuma delas. A consistência exige que a execução isolada de uma transação não viole a consistência do sistema. O isolamento permite que cada transação não perceba a execução concorrente de outras transações. Por fim, a durabilidade exige que transações finalizadas tenham

seus dados persistidos no sistema [Härder and Reuter 1983, Magalhães et al. 2021].

Os SGBDs comumente utilizam algum método de controle de concorrência para garantir a consistência dos dados em face a múltiplos acessos aos dados. O controle de concorrência em um banco de dados assegura que transações distintas acessem os dados concorrentemente sem levar o sistema a uma inconsistência. O gerenciamento do controle de concorrência é um tópico de desempenho estudado desde as primeiras pesquisas em bancos de dados relacionais (por exemplo, [Gray et al. 1975], [Bernstein et al. 1987], [Gray and Reuter 1993], [Bernstein and Goodman 1981]).

A maioria dos bancos de dados relacionais utiliza algum protocolo de controle de concorrência baseado em bloqueio, como o Bloqueio em Duas Fazes (*Two-Phase Locking* - 2PL). Nesse protocolo, por exemplo, uma transação $T_1$ deve adquirir bloqueios nos dados que necessita antes de começar a modificá-los. Uma transação $T_2$ que necessite alterar dados bloqueados por $T_1$ deve esperar até que $T_1$ não precise mais desses dados e desbloqueie-os. Esse tipo de consistência, que assegura que diferentes tipos de dados façam sentido juntos, é chamado de consistência lógica [Magalhães et al. 2021, Magalhães et al. 2023].

### 3.2.2. Por que utilizar bancos de dados NoSQL?

#### 3.2.2.1. Escalabilidade

Nas últimas décadas, houve um crescimento exponencial na quantidade de dados a serem armazenados. Assim, muitos sistemas tem tido a necessidade de armazenar e processar uma quantidade massiva de dados, muitas vezes em tempo real, a fim de tornar a recuperação da informação viável. Entretanto, os sistemas de armazenamento tradicionais não têm acompanhado essa demanda. Esse cenário foi inicialmente percebido por empresas de *internet*, como Amazon, Google, Facebook e Twitter. Contudo, atualmente outras empresas/organizações têm encontrado esse mesmo obstáculo para fornecer serviços com tempos de resposta aceitáveis [Magalhães et al. 2018].

Juntamente com o aumento da quantidade de dados, também houve o aumento de usuários. Lidar com aumento de dados e tráfego exige a adoção de mais recursos computacionais. Para suportar esse crescimento, há duas opções de escalabilidade: vertical e horizontal. Escalonar verticalmente (*scale up/down*) significa adicionar recursos ao servidor, como processador, memória e disco. Porém, máquinas mais robustas tornam-se cada vez mais caras. Além disso, há limites físicos quanto ao aumento de recursos em uma única máquina. Escalonar horizontalmente (*scale out/in*) significa utilizar um ou mais *clusters*. Um *cluster* é um conjunto de servidores (nós ou nodos) interconectados, que atuam como se fossem um único sistema trabalhando juntos. Um *cluster* pode utilizar máquinas de *hardware* mais acessível e barato. Essa estratégia também pode ser mais tolerante a falhas, visto que, embora nodos possam falhar, o *cluster* pode continuar funcionando na totalidade [Hill 1990, Bondi 2000, Michael et al. 2007]. A Seção 3.4.4 detalha bancos de dados distribuídos.

Os SGBDs relacionais foram concebidos para funcionar em um servidor centralizado. Assim, embora existam bancos de dados relacionais distribuídos (por exemplo: MySQL NDB Cluster, Oracle RAC e Microsoft SQL Server), a escalabilidade horizontal ainda pode ser um desafio. Existem vários problemas a serem resolvidos na implementação de um banco de dados distribuído. Manter a consistência dos dados é um dos desafios mais importantes e complexos. Por exemplo, uma única transação pode manipular dados em máquinas diferentes e, por esse motivo, sincronizar atualizações torna-se um processo complexo. Geralmente, esses sistemas utilizam um sistema de arquivos para gerenciar os dados no *cluster*, o que representa um ponto único sujeito a falhas [Sadalage and Fowler 2019, Davoudian et al. 2018].

Os SGBDs NoSQL têm sido uma alternativa para cenários em que os sistemas tradicionais não têm se mostrado tão eficientes. Esses SGBDs possuem arquitetura diferenciada e seguem conceitos diferentes para facilitar o tratamento com alta escalabilidade de dados. Por exemplo, alguns SGBDs NoSQL comumente utilizam uma consistência relaxada (ver Seção 3.4.5 para mais detalhes). Eles também podem utilizar o conceito de agregado para facilitar a distribuição de seus dados em *clusters*. A Seção 3.4.1 explica agregados em detalhes e a Seção 3.4.4 discute a distribuição de dados [Sadalage and Fowler 2019, Davoudian et al. 2018].

É importante ressaltar que os SGBDs relacionais ainda são uma opção confiável e bem consolidada para desenvolvimento de muitos sistemas contemporâneos. Eles são muito utilizados em vários setores, como finanças, saúde, varejo e muitos outros, devido à sua capacidade de garantir a integridade dos dados e fornecer recursos flexíveis a consultas nos dados. Portanto, ao considerar a implementação de um banco de dados, é essencial avaliar as necessidades específicas do seu projeto e considerar as vantagens e desvantagens dos bancos relacionais em relação a outras opções, como bancos de dados NoSQL [Sadalage and Fowler 2019, Davoudian et al. 2018]. A Seção 3.4 discute tópicos que podem ajudar na escolha do SGBD a ser implementado no projeto de banco de dados.

#### 3.2.2.2. Incompatibilidade de impedância

As operações em um banco de dados relacional consomem e retornam relações. Essas operações são simples e práticas, mas também possuem limitações. Os valores de uma tupla devem ser simples. Contudo, as estruturas de dados em memória utilizadas nas linguagens de programação podem ser muito mais complexas. Elas podem conter estruturas como registros ou listas, muitas vezes aninhadas. Como consequência, antes de armazenar um dado, é necessário traduzi-lo para o modelo relacional. Esse processo traz um trabalho adicional para os desenvolvedores. Essa diferença de representação do dado entre as estruturas em memória e o modelo relacional é chamada de incompatibilidade de impedância [Sadalage and Fowler 2019, Davoudian et al. 2018].

A Figura 3.2 ilustra a incompatibilidade de impedância de dados entre uma estrutura de dados em memória (a) e tabelas de um banco de dados relacional (b). Os dados representam informações de um pedido de compras de um cliente cujos

objetos estão aninhados. Antes de salvar esses objetos no banco de dados, eles devem ser desaninhados e distribuídos em tabelas diferentes [Sadalage and Fowler 2019].

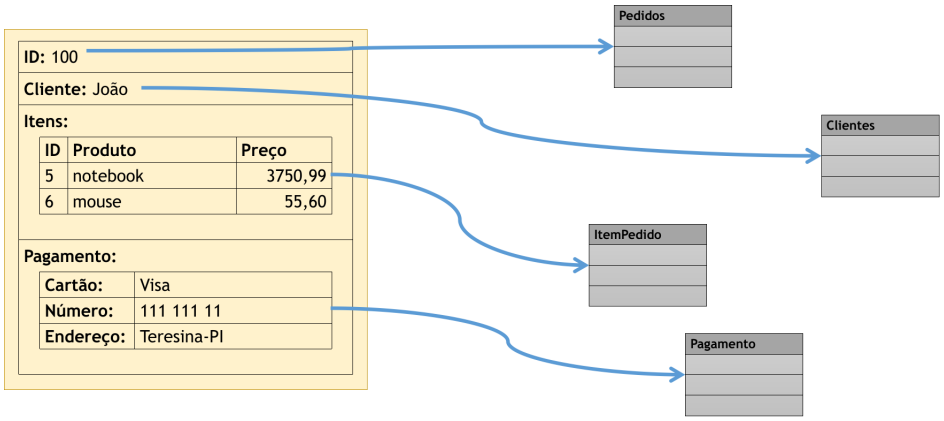



**Figura 3.2. Incompatibilidade de impedância entre uma estrutura de dados em memória (a) e tabelas de banco de dados (b) [Sadalage and Fowler 2019].**

Alguns *frameworks* de Mapeamento Objeto-Relacional (ORM), como Hibernate, JPA, Sequelize e Django, são amplamente utilizados para lidar com a incompatibilidade de impedância. Esses *frameworks* implementam padrões para tornar essa incompatibilidade mais transparente, poupando muito trabalho dos desenvolvedores. Porém, esses sistemas podem comprometer o desempenho do SGBD em alguns casos. O ORM adiciona uma camada extra de abstração e processamento que pode ser mais lenta do que usar instruções SQL diretas ao SGBD em sistemas que lidam com grandes volumes de dados ou que precisam de tempos de resposta muito rápidos. Além disso, o ORM pode não conseguir mapear tão bem objetos em tabelas do banco de dados, o que pode resultar em inconsistências ou comportamentos inesperados. Adicionalmente, o ORM pode não permitir que os desenvolvedores otimizem consultas para um melhor desempenho. O ORM ainda pode adicionar complexidade ao sistema. Por exemplo, Hibernate possui mais de mil opções de configuração [Ambler 2000].

Os bancos de dados orientados a objetos (BDOOs) surgiram na década de 80 como uma promessa de solução para a incompatibilidade de impedância. Esse SGBD armazenam as informações na forma de objetos, ou seja, utiliza a estrutura de dados denominada orientação a objetos, a qual permeia as linguagens mais modernas. Contudo, os BDOOs não conseguiram se consolidar no mercado devido a algumas desvantagens, como falta de padronização, dificuldade em expressar consultas mais complexas, os produtos existentes não estão tão maduros como os SGBDs relacionais, dentre outros [Boscarioli et al. 2006].

## 3.3. Bancos de dados NoSQL

Google e Amazon estiveram à frente no desenvolvimento de sistemas de bancos de dados alternativos para gerenciar quantidades massivas de dados em *clusters*.

Na primeira década de 2000, elas apresentaram os bancos de dados Big Table e Dynamo, respectivamente. Os bancos de dados NoSQL podem ser vistos como alternativa para trabalhar com esses casos, por possuírem arquitetura diferenciada e seguirem conceitos diferentes, para facilitar o tratamento com alta escalabilidade de dados. Grandes empresas, por exemplo, Google, oFacebook e Twitter, adotaram a utilização de bancos de dados NoSQL, mais especificamente de modelo colunar, devido à grande demanda por consultas, vinculado ao elevado número de informações que manipulam [Paniz 2016, Vieira et al. 2012].

Modelos de dados referem-se às formas como os dados são organizados e estruturados em um banco de dados NoSQL. Ao contrário dos bancos de dados relacionais, que seguem um modelo de dados tabular com linhas e colunas, os bancos de dados NoSQL permitem uma variedade de modelos de dados para armazenar informações [Marquesone 2016, Vieira et al. 2012].

Cada modelo de dados tem suas particularidades, vantagens e deve ser projetado conforme os tipos de aplicativos, problema e casos de uso específicos. A escolha do modelo de dados adequado depende dos requisitos específicos do aplicativo, incluindo escalabilidade, flexibilidade e complexidade das consultas. Nessa seção são discutidos os principais tipos de bancos de dados NoSQL: chave-valor, orientado a documentos, colunar e orientado a grafos. O principal objetivo dessa seção é tratar das ferramentas relacionados a tecnologia NoSQL, como produtos, linguagens de acesso, manipulação e processamento dos dados [Marquesone 2016, Vieira et al. 2012].

### 3.3.1. Modelo chave-valor

O modelo chave-valor armazena dados em pares de chave e valor simples. Exemplos de bancos de dados chave-valor incluem Redis, Amazon DynamoDB, Microsoft Azure Cosmos DB, Memcached, etcd, Hazelcast, Aerospik, Ehcache, Riak KV e InterSystems IRIS. De acordo com [Silva et al. 2021], essa estrutura de armazenamento se baseia em uma função *hash* (Figura 3.3), que contém uma chave única e um apontador para um item. A organização baseada em *hashing* fornece um acesso muito rápido às informações.

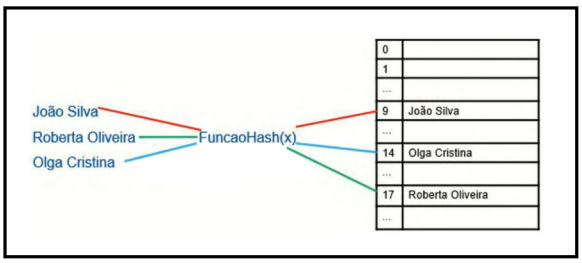



**Figura 3.3. Representação da tabela *Hash*. [Silva et al. 2021].**

Redis possui vários comandos simples para manipular valores em chaves. Para executá-los, algum programa de *interface* pode ser utilizado, como o Redis-Cli (*Command Line Interface*). O Redis-Cli é uma *interface* de linha de comandos

instalada juntamente com o Redis. Para utilizá-lo, basta apenas digitar o nome do programa (redis-cli) em algum terminal de comandos do sistema operacional [Redis Documentation 2024]. Exite também a opção de utilizar uma *interface* gráfica através do programa *Redis Insight* [Redis Insight 2024]. Seguem alguns dos comandos mais comuns e exemplos de uso deles.

- **SET**: O comando SET é usado para definir um valor para uma chave. Caso a chave já exista, o seu valor é atualizado. O comando SET também permite definir opções, como o tempo de expiração da chave [Redis Documentation 2024].

```
1       SET minha_chave "valor da chave"
2
```

**Listagem 3.1. Comando SET.**

- **APPEND**: O comando APPEND é usado para anexar um valor a uma chave existente. Se a chave não existir, o comando criará uma nova chave com o valor especificado [Redis Documentation 2024].

```
1       APPEND minha_chave " valor adicional"
2
```

**Listagem 3.2. Comando APPEND.**

- **GET**: O comando GET retorna o valor associado uma chave [Redis Documentation 2024].

```
1       GET minha_chave
2
```

**Listagem 3.3. Comando GET.**

- **HSET**: O comando HSET define campos especificados com seus respectivos valores em uma *hash* armazenada em uma chave [Redis Documentation 2024].

```
1       HSET minha_hash campo1 "valor1" campo2 "valor2"
2
```

**Listagem 3.4. Comando HGET.**

- **HGET**: O comando HGET retorna o valor de um campo associado a uma *hash* armazenada em uma chave [Redis Documentation 2024].

```
1       HGET minha_hash campo1
2
```

**Listagem 3.5. Comando HSET.**

- **INCR**: O comando INCR incrementa o valor associado a uma chave em 1 [Redis Documentation 2024].

```
1       INCR meu_numero
2
```

**Listagem 3.6. Comando INCR.**

- **DECR**: O comando DECR decrementa o valor associado a uma chave em 1 [Redis Documentation 2024].

```
DECR meu_numero

```

**Listagem 3.7. Comando DECR.**

- **DEL**: O comando DEL exclui uma ou mais chaves especificadas [Redis Documentation 2024].

```
DEL meu_numero

```

**Listagem 3.8. Comando DEL.**

Redis armazena valores simples. Caso seja necessário manipular valores complexos, eles devem ser gerenciados pela aplicação. A Listagem 3.9 exemplifica o armazenamento de uma estrutura de dados complexa ilustrada na Figura 3.4. Essa figura representa um exemplo de um sistema de carrinho de compras *online* em que são armazenadas informações sobre itens de produtos selecionados por um cliente [Paniz 2016, Lazoti 2016, Redis Documentation 2024].

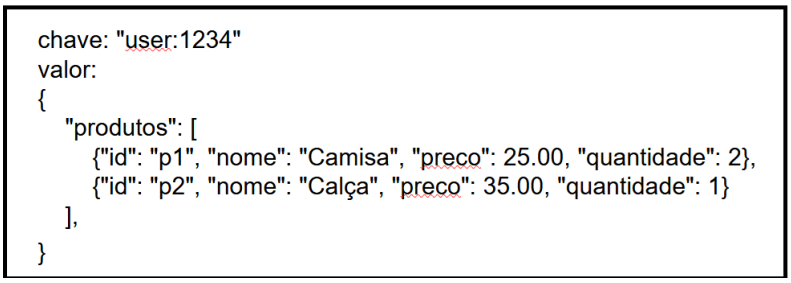



**Figura 3.4. Ilustração do modelo Chave-valor para exemplo de carrinho de compras *online*.**

```
# Adiciona os produtos ao carrinho do utilizador com ID 1234
HSET user:1234 produto:p1 '{"id": "p1",
                            "nome": "Camisa",
                            "preço": "25.00",
                            "quantidade": "2"}'
HSET user:1234 produto:p2 '{"id": "p2",
                            "nome": "Calça",
                            "preço": "35.00",
                            "quantidade": "1"}'
# Retorna os produtos do carrinho do utilizador com ID 1234
HGETALL user:1234

```

**Listagem 3.9. Comandos de exemplo em Redis CLI.**

O código da Figura 3.4 utiliza uma *hash* para armazenar os itens de produtos no carrinho de um cliente. A chave da *hash* (user:1234) possui um identificador único do usuário, por exemplo, o ID do usuário. Um item de produto é um campo da *hash*

que possui um identificador do produto, como o ID do produto. No exemplo, a *hash* possui os campos *produto:p1* e *produto:p2*. O valores dos campos são estruturas de dados complexas, em um formato semelhante a JSON, serializados em *strings*. Para a informação contida nos valores fazerem sentido, a aplicação deve gerenciá-los. Algumas linguagens de programação possuem bibliotecas para serializar JSONs [Paniz 2016, Lazoti 2016, Redis Documentation 2024].

### 3.3.2. Modelo orientado a documentos

Modelos de dados orientados a documentos trabalham com dados não relacionais e os armazena como documentos estruturados, geralmente nos formatos *XML* (*eXtensible Markup Language*; ou Linguagem de Marcação Extensível, em português) ou *JSON* (*JavaScript Object Notation*; ou Notação de Objetos JavaScript, em português) [Harrison 2015].

Os dados são estruturados de forma encadeada, podendo conter atributos das coleções, *tags* e metadados, e seguem uma hierarquia de informações. Além disso, é permitido que tenham redundância e inconsistência, conforme o ambiente NoSQL em que estão inseridos [Silva et al. 2021]. Exemplos de bancos orientados a documentos são: MongoDB, CouchDB, BigCouch, RavenDB, Clusterpoint Server, ThruDB, TerraStore, RaptorDB, JasDB, SisoDB, SDB, SchemaFreeDB e djondb.

Este trabalho dará ênfase ao MongoDB, um projeto *open source* com distribuição gratuita para Linux, Mac e Windows. Embora seja escrito em C++, seu ambiente iterativo e suas buscas são escritas em *JavaScript* [MongoDB Documentation 2024]. MongoDB possui uma *interface* de linha de comandos chamada de Mongo. Para utiliza-lo, basta apenas digitar o nome do programa (mongo) em algum terminal de comandos do sistema operacional. Exite também a opção de *interface* gráfica através do programa Studio 3T [Studio 3T 2024].

Os dados de um documento em uma coleção (equivalente à linha de uma tabela) são armazenados no formato BSON (*Binary* JSON). O BSON é um formato de serialização de documentos em binário, projetado para armazenar e transmitir dados de forma eficiente e estruturada. BSON é um superconjunto de JSON com mais alguns tipos de dados, principalmente a matriz de *bytes* binários. Um BSON pode ser definido de forma semelhante a um JSON entre os caracteres de chaves "{}". Dentro das chaves, são definidos grupos de chaves e seus respectivos valores. Os valores também podem ser listas de chave-valor. A Listagem 3.10 ilustra um registro no formato BSON [MongoDB Documentation 2024, Paniz 2016].

```
1    {
2      firstname: 'Arlino'
3      lastename: 'Magalhães'
4      email: ['home': 'arlinoh@gmail.com',
5              'work': 'arlino@ufpi.edu.br'],
6      phone: ['home': '+810001000'],
7      adress: []
8    }
9
```

**Listagem 3.10. Um documento no formato BSON de uma base de dados em MongoDB.**

A Listagem 3.11 mostra como inserir documentos utilizando o comando *insertOne* do MongoDB. Esse documento representa um álbum de músicas inserido na coleção *albuns* cujos campos são *nome_album* e *duracao*. Adicionalmente, podem ser inseridos outros álbuns com diferentes informações, tais como: data de lançamento, estúdio de gravação, nome do artista que produziu a capa, número de *singles*, quantidade de semanas que ficou em primeiro lugar na *billboard*, etc. Os álbuns não precisam ter todas essas informações, ou seja, eles não precisam seguir o mesmo esquema de dados. Essa abordagem é bastante prática, pois, por exemplo, a grande maioria dos álbuns não possui nenhum *single* e, consequentemente, esse dado seria inexistente em muitos documentos. Em contraste, em um banco de dados relacional, todos os registros de uma tabela devem seguir um mesmo esquema. Os bancos NoSQL baseados em documentos permitem que os documentos, de uma mesma coleção, tenham esquemas diferentes [MongoDB Documentation 2024, Paniz 2016].

```
1    db.albuns.insertOne ({
2      "nome_album": "Legião Urbana",
3      "duracao": 3286
4    )}
5
```

**Listagem 3.11. Inserção de documento no MongoDB (coleção albuns).**

O comando *find* pode ser utilizado para buscar documentos no MongoDB. A Listagem 3.12 mostra como fazer uma busca na coleção *albuns* utilizando um filtro no formato BSON que busca pelo álbum de nome Legião Urbana. Sem o filtro (BSON vazio) o comando *find* busca por todos os documentos da coleção. Para fins ilustrativos, a instrução SQL da Listagem 3.13 em um SGBD relacional é equivalente ao comando MongoDB da Listagem 3.12. Assim como nos SGBDs relacionais, nos quais uma tabela deve possuir uma chave primária, no MongoDB todas as coleções possuem um campo chamado *_id*, que também funciona como chave primária, mas é gerenciado pelo própria banco. Assim, o resultado obtido pelo comando *find* da Listagem 3.12 deve retornar o _id de cada documento, além dos demais campos do documento [MongoDB Documentation 2024, Paniz 2016].

```
1    db.albuns.find ({'nome_album': 'Legião Urbana'})
2
```

**Listagem 3.12. Busca de um documento no MongoDB.**

```
1    SELECT * FROM albuns WHERE nome_album = 'Legião Urbana'
2
```

**Listagem 3.13. Busca em um SGBD relacional.**

Os relacionamentos no MongoDB são realizados referenciando o ID de um objeto de outra coleção (semelhante a uma chave estrangeira em SGBDs relacionais). Para exemplificar relacionamentos, considere o gênero musical *MPB* inserido através do código da Listagem 3.14. O documento desse gênero musical está representado na Listagem 3.15 cujo ID é igual a *ObjectId(54d1562cf7bb967d7b976d07)*. O ID do

gênero musical é utilizado para referenciá-lo em um álbum. O comando da Listagem 3.16 insere o álbum *Tom Jobim* com referência (campo *id_genero*) ao ID do gênero musical *MPB*. O documento gerado por esse comando é ilustrado na Listagem 3.17.

```
db.generos.insertOne({
  nome_genero: "MPB"
});
```

**Listagem 3.14. Inserção de documento MongoDB (coleção gêneros).**

```
{
  "_id": ObjectId("54d1562cf7bb967d7b976d07"),
  "nome_genero": "MPB"
}
```

**Listagem 3.15. Documento da coleção de gêneros musicais.**

```
db.generos.insertOne({
  "nome_album": "Tom Jobim"
  "duracao": 3200
  "id_genero": ObjectId("54d1562cf7bb967d7b976d07")
});
```

**Listagem 3.16. Inserção de documento com referência a outro documento.**

```
{
  "_id": ObjectId("63c828b5342dda43e93bee1e"),
  "nome_album": "Tom Jobim"
  "duracao": 3200
  "id_genero": ObjectId("54d1562cf7bb967d7b976d07")
}
```

**Listagem 3.17. Documento de uma coleção com referência a outro documento.**

Para exibir todos os álbuns e seus respectivos gêneros musicais, basta apenas listar todos os documentos da coleção de álbuns e buscar o documento da coleção de gêneros referenciado através do campo *id_genero*. O código da Listagem 3.18 mostra como fazer essa busca. Nesse código, a primeira linha busca todos os álbuns armazenando-os na variável *albuns*. Na segunda linha há a função *forEach* que recebe como argumento outra função (*album*), na qual é feita uma nova busca na coleção *generos* pelo documento correspondente ao _id *genero* de cada álbum [MongoDB Documentation 2024, Paniz 2016]

```
var albuns = db.albuns.find({});
albuns.forEach(function(album) {
  var genero = db.generos.findOne({"_id": album["id_genero"]});
  print(album["nome_album"], genero["nome_genero"]);
})
```

**Listagem 3.18. Código para buscar todos os álbuns e seus respectivos gêneros musicais.**

### 3.3.3. Modelo Colunar

O modelo colunar, ou famílias de colunas, armazenam dados em famílias de colunas como linhas. Essas linhas possuem inúmeras colunas relacionadas com a chave dessa linha. São coleções de dados que estão relacionados e são acessados juntos. Cada família de colunas pode ser comparada a um contêiner de linhas em uma tabela de um SGBD relacional. Nos SGBDs relacionais, a chave é usada para reconhecer cada linha. No modelo colunar, cada linha é composta por inúmeras colunas [Benymol and Sajimon 2017].

O modelo colunar utiliza-se de tabelas para representação de entidades e os dados são gravados em disco agrupados por coluna, o que reduz o tempo de leitura e escrita em disco. Os bancos colunares são os que mais se assemelham aos bancos relacionais por terem uma tabela, mesmo que, na verdade, eles sejam muito diferentes [Benymol and Sajimon 2017].

Cassandra, foco desta seção, é um dos bancos de dados de famílias de colunas mais conhecidos [Paniz 2016, Cassandra Documentation 2024]. Outros bancos de dados importantes de famílias de colunas são HBase, Microsoft Azure Cosmos DB, Datastax Enterprise, ScyllaDB, Microsoft Azure Table Storage, Accumulo, Google Cloud Bigtable e Amazon Keyspaces.

Cassandra utiliza o conceito de *keyspace*, que é similar a um *database*, onde tabelas são agrupadas para uma finalidade específica, geralmente para separar dados de aplicações diferentes. Outro conceito importante é o de família de colunas (*column family*) que é semelhante a uma entidade. Nas versões mais recentes de Cassandra o termo família de colunas foi substituído por tabela [Paniz 2016, Cassandra Documentation 2024]. A Figura 3.5 ilustra como os dados são organizados em Cassandra.

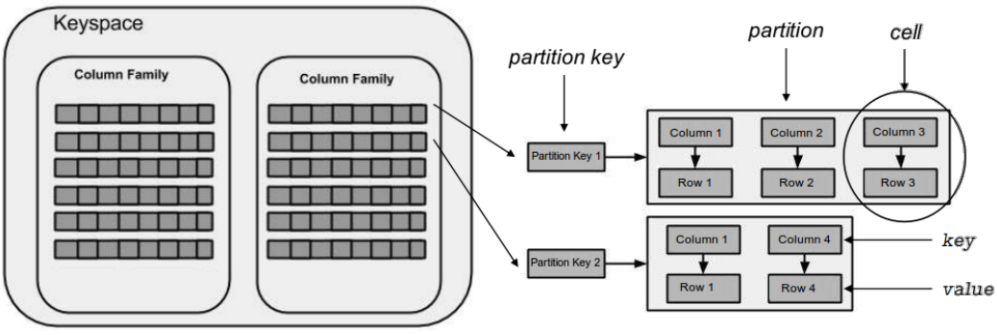



**Figura 3.5. Colunas dinâmicas no banco de dados Cassandra.**

Cassandra utiliza a linguagem CQL (*Cassandra Query Language*) que é semelhante a SQL utilizada por SGBDs relacionais. A *interface* de linhas de comandos cqlsh pode ser usada para interagir com Cassandra. Para usá-la, basta apenas digitar o seu nome (cqlsh) em um terminal de comandos do sistema operacional [Paniz 2016, Cassandra Documentation 2024]. A Listagem 3.19 mostra como criar uma tabela em Cassandra através do comando *CREAT TABLE*, no qual é criada uma tabela de filmes. Os comandos *SELECT, INSERT, UPDATE e DELETE* também são utilizados de forma idêntica a SQL [Paniz 2016,Cassandra Documentation 2024].

```
CREATE TABLE filmes (
  videoId uuid,
  nome_filme varchar,
  descricao varchar,
  localizacao_imagem_capa text,
  data timestamp,
  PRIMARY KEY (videoId)
)

```

**Listagem 3.19. Craindo tabela no Cassandra.**

Apesar da sintaxe da busca em Cassandra ser idêntica a da busca nos SGBD relacionais, diferentemente deles, Cassandra não faz buscas em campos que não possuem índices. Nos SGBDs relacionais, apesar de não ser performático, esse tipo de busca funciona. Porém, devido à maneira como o banco de dados Cassandra organiza os dados, uma busca por campos sem índices seria tão ineficiente que ele prefere não suportar. Isso ocorre porque Cassandra é otimizado para operações de leitura rápidas e eficientes, que são possíveis quando os dados são acessados diretamente por meio de chaves primárias ou índices secundários. Buscas em colunas sem índices podem exigir varreduras completas de tabelas, o que é ineficiente em ambientes distribuídos e escaláveis. Para ilustrar melhor essa situação, na Listagem 3.20 há uma consulta em CQL que não será executada se não for criado um índice na coluna filtrada, conforme ilustrado na Listagem 3.21 [Paniz 2016, Cassandra Documentation 2024].

```
SELECT * FROM filmes WHERE nome_filme = 'Matrix Revolutions'

```

**Listagem 3.20. Busca utilizando um filtro em Cassandra.**

```
CREATE INDEX ON filmes (nome_filme)

```

**Listagem 3.21. Criação de índice em Cassandra.**

A tabela criada com a sintaxe definida na Listagem 3.19 é muito semelhante a uma tabela no modelo relacional. Entretanto, o modelo de dados utilizado comumente em bancos colunares é o baseado em colunas dinâmicas (*dynamic columns*). A Figura 3.5 ilustra o modelo de colunas dinâmicas implementado em Cassandra. Nesse modelo, as tabelas são *schameless*, nas quais uma linha é chamada de partição (*partition*). Cada partição possui sua chave única (*partition key*) que, a partir desta chave, podem ser adicionados vários pares chave/valor chamados de pares coluna/-linha (*colunm/row*). E cada par coluna/linha em uma partição é chamado de célula (*cell*). As células de uma mesma partição são gravadas agrupadas para reduzir o tempo de leitura e escrita em disco [Paniz 2016, Cassandra Documentation 2024].

Para criar uma tabela com colunas dinâmicas é necessário utilizar o parâmetro *WITH COMPACT STORAGE* na criação da tabela. Além disso, a tabela deve possuir, pelo menos, três colunas: uma que será a chave da partição; uma que será o nome da coluna dinâmica e a última será o valor da linha. Adicionalmente,

a chave primária deve ser uma chave composta entre a chave da partição e o nome da coluna [Paniz 2016, Cassandra Documentation 2024].

A Listagem 3.22 exemplifica como criar uma tabela com colunas dinâmicas em Cassandra usando o parâmetro *WITH COMPACT STORAGE*. Nesse exemplo, a tabela *filmes_assistidos* terá o campo *userId* como chave de partição, os campos *watch_date* e *videoId* como nome da coluna dinâmica e, finalmente, os campos *nome_filme* e *localizacao_imagem_capa* como valores da linha. Nessa tabela, os campos *nome_filme* e *localizacao_imagem_capa* são campos provenientes da tabela filmes (Listagem 3.19) [Paniz 2016, Cassandra Documentation 2024].

```
1    CREATE TABLE filmes_assistidos (
2      userId uuid,
3      watch_date timestamp,
4      videoId uuid,
5      nome_filme varchar,
6      localizacao_imagem_capa text,
7      PRIMARY KEY (userId, watch_date, videoId)
8    ) WITH COMPACT STORAGE
9
```

**Listagem 3.22. Criação de tabela com colunas dinâmicas em Cassandra por meio do parâmetro WITH COMPACT STORAGE.**

Cassandra não tem nenhum conceito de integridade referencial e, portanto, não possui a operação *join*. Por esse motivo, os dados foram desnormalizados, evitando acessos à tabela *filmes*, para que uma busca na tabela *filmes_assistidos* fosse mais rápida. Como já dito anteriormente, a linguagem CQL de Cassandra é muito semelhante a SQL dos bancos relacionais, mesmo que a organização dos dados nos dois tipos de bancos sejam bem diferentes, como mostrado na Figura 3.5. Isso foi feito para que os programadores acostumados com SQL utilizem CQL mais facilmente. Em versões mais antigas do Cassandra, era permitido acessar as colunas dinâmicas de forma semelhante aos dados nos bancos chave-valor [Paniz 2016, Cassandra Documentation 2024].

Em versões mais recentes de Cassandra (a partir da versão 3.0) o uso de *WITH COMPACT STORAGE* foi desaconselhado e pode ser removido em versões futuras. A criação de tabelas com colunas dinâmicas pode ser feita sem esse parâmetro usando outras abordagens, como tabelas estáticas e coleções. A Listagem 3.23 apresenta uma código sem o uso do parâmetro *WITH COMPACT STORAGE* [Cassandra Documentation 2024].

```
1    CREATE TABLE filmes_assistidos (
2      userId uuid,
3      watch_date timestamp,
4      videoId uuid,
5      nome_filme varchar,
6      localizacao_imagem_capa text,
7      PRIMARY KEY (userId, watch_date, videoId)
8    )
9
```

**Listagem 3.23. Criação de tabela com colunas dinâmicas em Cassandra sem o parâmetro WITH COMPACT STORAGE.**

### 3.3.4. Modelo orientado a grafos

A matemática e a computação caminham juntas, e suas teorias foram aplicadas nos sistemas computacionais. Uma delas é a teoria dos grafos, que pode ser aplicada na representação de diversos problemas e implementada em *software* [Silva et al. 2021]. A teoria dos grafos oferece inúmeras ferramentas que podem ser aplicadas para quantificar e analisar os padrões de conectividade de sistemas complexos [Phillips et al. 2015].

A teoria dos grafos pode ser aplicada no contexto NoSQL, em especial ao modelo orientado a grafos. Esse modelo possui estruturas de dados modeladas na forma de grafos. A manipulação de dados é expressa mediante operações orientadas a grafos. Com isso, eles abstraem a complexidade dessas estruturas provendo ferramentas para explorar seu potencial.

Os dados nos bancos de grafos são representados por nodos, arestas e propriedades. Eles não implementam tabelas. Os nodos representam as entidades, as arestas expressam as relações entre os nodos e as propriedades apresentam características das entidades e relacionamentos [Phillips et al. 2015]. As bases de dados orientadas a grafos processam com eficiência densos conjuntos de dados e o seu *design* permite a construção de modelos preditivos e análise de correlações e de padrões de dados. Seguem as principais vantagens de se utilizar o modelo orientado a grafos:

- alto desempenho independentemente do tamanho total do conjunto de dados (garantido pelas travessias nos grafos);
- o modelo de grafos aproxima os domínios técnicos e de negócios facilitando a modelagem dos dados e;
- facilidade de se alterar o esquema de dados incluindo novas entidades e relacionamentos sem a necessidade de reestruturar o esquema de dados.

Exemplos de bancos orientados a grafos são: Neo4J, Microsoft Azure Cosmos DB, Aerospike, Virtuoso, ArangoDB, GraphDB, OrientDB, Memgraph, Amazon Neptune, NebulaGraph, Stardog, JanusGraph, TigerGraph, Fauna, Dgraph, Giraph e AllegroGraph. Nessa seção serão mostrados exemplos práticos de como modelar, consultar e recuperar dados no banco de dados Neo4j. Ele pode ser executado na máquina virtual Java e possui distribuições para Windows, Mac e Linux. Semelhantemente a outros bancos orientados a grafos, Neo4j pode ser aplicado a problemas que envolvem gerenciamento de rede, *softwares* analíticos, pesquisa cientifica, roteamento, gestão organizacional e de projeto, recomendações, redes sociais, dentre outros [Paniz 2016, Neo4J Documentation 2024].

Neo4J utiliza a linguagem de consulta *cypher* como uma ferramenta intuitiva para interagir com bancos de dados orientados a grafos. Cada nodo criado no Neo4j pode ter um tipo e propriedades associadas a ele. Na Listagem 3.24 são criados dois nodos que representam pessoas, utilizando o comando CREATE. Os dois nodos são do tipo Pessoa e possuem as propriedades *nome* e *idade*. Eles estão associados às variáveis *neo* e *smith* [Paniz 2016, Neo4J Documentation 2024].

```
CREATE (neo:Pessoa {nome:'Neo', idade: 29})
CREATE (smith:Pessoa {nome:'Agente Smith', idade: 36})
```

**Listagem 3.24. Comando para criar nodos no Neo4J.**

Para exibir os nodos inseridos, deve ser utilizado o comando *MATCH*. Além do *MATCH*, a outra parte obrigatória em um comando de busca é o *RETURN*, usado para especificar o que será retornado. O comando da Listagem 3.25 busca por todos os nodos do tipo Pessoa retornando seus nomes. Para fins ilustrativos, comando *cypher* da Listagem 3.25 é equivalente ao comando SQL da Listagem 3.26 [Paniz 2016, Neo4J Documentation 2024].

```
MATCH (m:Pessoa)
RETURN m.nome
```

**Listagem 3.25. Comando para buscar todos os nodos de um determinado tipo no Neo4J.**

```
SELECT m.nome FROM Pessoa m
```

**Listagem 3.26. Comando para buscar todos os registros de uma tabela em SGBD relacional.**

Uma alternativa ao método expresso na Listagem 3.24 é a utilização do comando *MERGE*. Porém, este comando verifica se o nodo já existe no banco de dados antes se o inserir. Se ele existir, ele não cria um novo nodo, evitando duplicação. A Listagem 3.27 representa a sintaxe para uso do comando *MERGE* no Neo4J [Paniz 2016, Neo4J Documentation 2024].

```
MERGE (neo:Pessoa {nome: 'Neo', idade: 29});
MERGE (smith:Pessoa {nome: 'Agente Smith', idade: 36});
```

**Listagem 3.27. Comando para criar nodos no Neo4J sem duplicação.**

Após nodos terem sido criados, arestas (relacionamentos) são necessárias para dar forma ao grafo. A Listagem 3.28 mostra como criar um relacionamento no Neo4j. Primeiro, dois nodos já existentes são buscados e armazenados nas variáveis *p1* e *p2* através do comando MATCH. Em seguida, um relacionamento é criado entre *p1* e *p2* através do comando CREATE, no qual é atribuído a propriedade *CONHECE* [Paniz 2016, Neo4J Documentation 2024].

```
MATCH (p1:Pessoa {nome: 'Neo'}),
  (p2:Pessoa {nome: 'Agente Smith'})
CREATE (p1)-[:CONHECE]->(p2)
```

**Listagem 3.28. Comando para criar relacionamento no Neo4J.**

A Listagem 3.29 apresenta uma alternativa para criar arestas em relação ao expresso na Figura 3.28. O diferencial para esta abordagem é que ela verifica a existência do relacionamento antes de criá-lo. A cláusula *WHERE NOT* garante que o relacionamento *CONHECE* será criado somente se ainda não existir entre *p1* e *p2* [Paniz 2016, Neo4J Documentation 2024].

```
MATCH (p1:Pessoa {nome: 'Neo'}),
  (p2:Pessoa {nome: 'Agente Smith'})
  WHERE NOT (p1)-[:CONHECE]->(p2)
CREATE (p1)-[:CONHECE]->(p2)

```

**Listagem 3.29. Comando para criar de relacionamento no Neo4J sem duplicação.**

Enfim, existem várias alternativas para criar relacionamentos no Neo4j, dependendo do contexto e das necessidades específicas da aplicação. A escolha da alternativa mais apropriada depende dos requisitos, como evitar duplicações, melhorar a flexibilidade, ou verificar condições adicionais antes da criação do relacionamento. Todas essas alternativas são válidas e podem ser usadas conforme a situação específica do banco de dados e da aplicação em questão [Paniz 2016, Neo4J Documentation 2024].

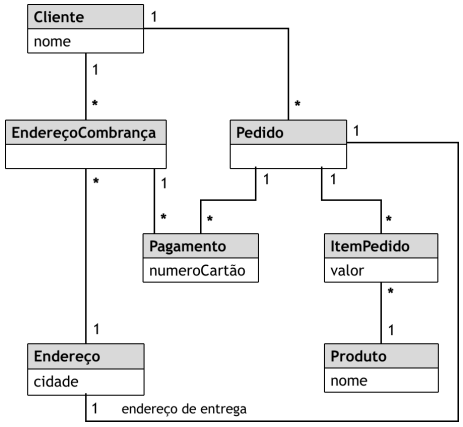
## 3.4. Projeto de bancos de dados NoSQL

### 3.4.1. Agregados

Os bancos de dados NoSQL chave-valor (Seção 3.3.1), documento (Seção 3.3.2) e colunar (Seção 3.3.3) podem fazer uso de uma estrutura de dados complexa chamada de agregado. Essa estrutura é capaz de conter listas e outras estruturas de dados aninhadas dentro dela. O agregado é a representação de um conjunto de objetos relacionados tratados como uma unidade que pode ser atualizada por meio de operações atômicas. Ele facilita a execução de bancos de dados em um *cluster*, visto que um agregado representa uma unidade natural para replicação e fragmentação (ver Seção 3.4.4). Além disso, os agregados são mais simples de ser manipulados pelos desenvolvedores, pois eles se assemelham às estruturas de dados utilizadas nas linguagens de programação. Os bancos de dados NoSQL orientados a grafos (Seção 3.3.4) não implementam agregados [Strauch et al. 2011].

Embora os SGBD orientados a agregados sejam projetados para trabalhar com unidades de objetos agreados, esses sistemas não implementam essas unidades de forma automatizada. A fim de exemplificar a implementação de agregados, considere a modelagem relacional de um sistema de pedidos de produtos representada na Figura 3.6(a). A Figura 3.6(b) exemplifica dados típicos para o banco de dados da modelagem da Figura 3.6(a). A modelagem está normalizada, de modo a não existirem valores repetidos em tabelas diferentes, igual como se espera no modelo relacional. Contudo, essa modelagem não está detalhada devido a limitações de espaço e por questões didáticas [Sadalage and Fowler 2019].

O mesmo sistema representado na Figura 3.6 está representado na Figura 3.7. Porém, enquanto a Figura 3.6 implementa uma modelagem relacional, a modelagem da Figura 3.7 considera a orientação agregada, que é bastante diferente. Na modelagem da Figura 3.7(a), os dados agregados estão representados através do marcador de composição UML, o losango preto. Assim, existem os agregados Cliente, Pedido e Pagamento. A Figura 3.7(b) representa os objetos modelados na Figura 3.7(a) [Sadalage and Fowler 2019].




(a) Modelagem relacional.

**Cliente**

| id | nome |
|---|---|
| 1 | João |

**Pedidos**

| id | idCliente | idEndereçoEntrega |
|---|---|---|
| 31 | 1 | 21 |

**Endereço**

| id | cidade |
|---|---|
| 21 | Teresina |

**EndereçoCobrança**

| id | idCliente | idEndereço |
|---|---|---|
| 11 | 1 | 21 |

**ItemPedido**

| id | idPedido | idProduto | preço |
|---|---|---|---|
| 41 | 31 | 5 | 3750.99 |
| 43 | 31 | 6 | 55.60 |

**Produto**

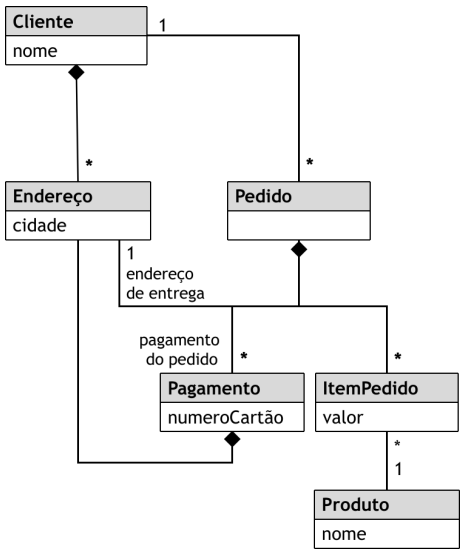
| id | nome |
|---|---|
| 5 | notebook |
| 6 | mouse |

**Pagamento**

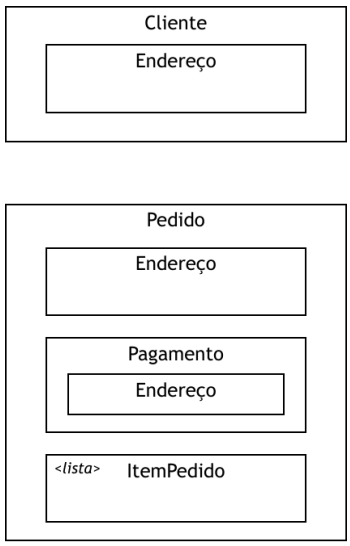
| id | numeroCartão | idPedido | idEndereçoCobrança |
|---|---|---|---|
| 11 | 111 111 11 | 31 | 21 |

(b) Dados típicos de tabelas de um banco de dados.

**Figura 3.6. Modelagem de sistema de pedidos de produtos em SGBD relacional [Sadalage and Fowler 2019].**




(a) Agregados de clientes e pedidos separados.

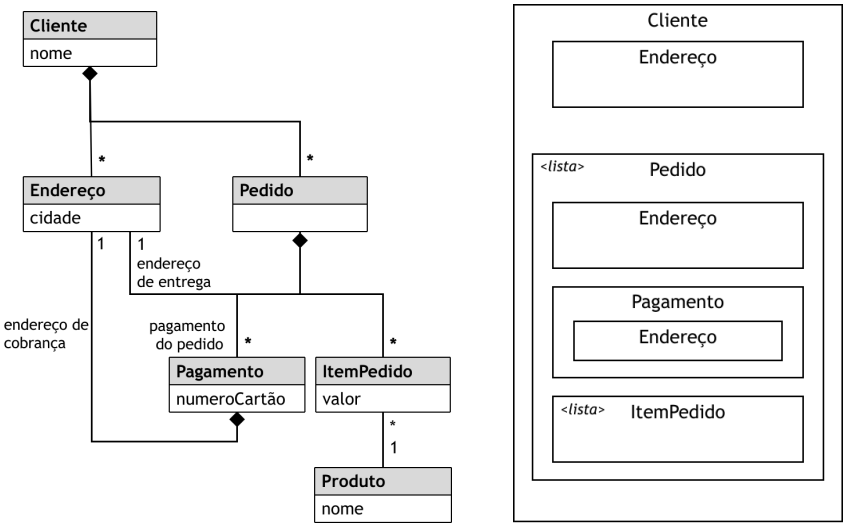



(b) Objetos de clientes e pedidos separados.

**Figura 3.7. Esquema de pedidos de produtos em modelagem orientada a agregados [Sadalage and Fowler 2019].**

O agregado Cliente possui o objeto Endereço embutido dentro dele. O agregado Pedido, por sua vez, possui os objetos Endereço e Pagamento e, ainda, uma lista de objetos ItemPedido aninhada. Por fim, o objeto Pagamento é um agregado que possui um objeto Endereço embutido. Observe que um mesmo registro de endereço pode se repedir três vezes para um pedido de um cliente na forma de endereço do cliente, endereço de cobrança e endereço de envio. A conexão entre o cliente e seus pedidos é feita por relacionamento entre agregados através do ID do cliente. A modelagem aplicou uma desnormalização, onde os produtos passaram a fazer parte de itens de pedido, ou seja, cada objeto da lista ItemPedido também possui informações dos produtos [Sadalage and Fowler 2019, Frozza et al. 2022].

Não existe uma maneira específica e bem definida de como modelar agregados. Esse processo deve ser feito levando em consideração como os dados devem ser acessados pelo aplicativo. A orientação agregada espera que um agregado seja um conjunto de objetos relacionados e acessados (recuperados/gravados) juntos. A modelagem da Figura 3.7(a) permite que todas as informações de um pedido e de todos os seus produtos sejam acessadas como uma unidade. Contudo, os limites dos agregados poderiam ser definidos de forma diferente, como no exemplo da Figura 3.8. Nesse exemplo, todos os pedidos (e produtos) de um cliente estão no agregado Cliente. Dessa maneira, é possível acessar todos os pedidos de um determinado cliente por vez. A Figura 3.8(b) ilustra os objetos da modelagem da Figura 3.8(a).




(a) Agregado de cliente com pedidos embutidos.

(b) Agregado de cliente com pedidos embutidos.

**Figura 3.8. Dois esquemas de pedidos de produtos em modelagem orientada a agregados [Sadalage and Fowler 2019].**

A Listagem 3.30 exemplifica dados típicos considerando a modelagem da Figuras 3.7. Nessa listagem os objetos *cliente* e *pedido* estão separados. Em contraste, a Listagem 3.31 exemplifica dados da modelagem 3.8 cujos objetos estão aninhados [Sadalage and Fowler 2019].

```
// Cliente
{
  "id": 1,
  "nome": "João",
  "enderecoCobrança": [ {"cidade": "Teresina"} ]
}
// Pedido
{
  "id": 31,
  "idCliente": 1,
  "enderecoEntrega": {"cidade": "Teresina"},
  "itemPedido": [
                  {
                    "idProduto": 6,
                    "nome": "mouse",
                    "preço": 55.60
                  },
                  {
                    "idProduto": 5,
                    "nome": "notebook",
                    "preço": 3750.99
                  }
                ],
  "pagamento": [
                  {
                    "numeroCartão": "111 111 11",
                    "EndereçoCobrança":
                    {
                      "cidade": "Teresina"
                    }
                  }
                ]
}

```

**Listagem 3.30. Dados de clientes e pedidos separados.**

```
// Cliente e pedidos embutidos
{
  "id": 1,
  "nome": "João",
  "enderecoCobrança": [ {"cidade": "Teresina"}],
  "pedidos":[
              {
                "id": 31,
                "enderecoEntrega": {"cidade": "Teresina"},
                "itemPedido": [
                                {
                                  "idProduto": 6,
                                  "nome": "mouse",
                                  "preço": 55.60
                                },
                                {
                                  "idProduto": 5,
                                  "nome": "notebook",
                                  "preço": 3750.99
```

```
                                        }
                                    ],
                        "pagamento": [
                        {
                          "numeroCartão": "111 111 11",
                          "EndereçoCobrança": {"cidade": "Teresina"}
                        }]
                      }
    ]
}

```

**Listagem 3.31. Dados de clientes com pedidos embutidos.**

Os exemplos mostrados nessa seção evidenciam que não há uma maneira universal para modelar agregados. Esse processo depende de como os dados serão manipulados, ou seja, depende da forma pela qual os dados são usados pelo aplicativo. Cada modelagem possui seus *trade-offs*. O segundo agregado (Figura 3.8) permite um acesso rápido a todos os pedidos de um cliente. Porém, ele torna custoso o acesso a determinados pedidos como, por exemplo, o acesso aos pedidos do último trimestre. Em contraste, o primeiro agregado (Figura 3.7) facilita a busca por um determinado pedido, mas é potencialmente mais lento para encontrar todos os pedidos de um cliente [Sadalage and Fowler 2019, Rodrigues et al. 2017].

Como desvantagem, os bancos de dados orientados a agregados não suportam transações ACID. Esses sistemas suportam apenas manipulações atômicas em um único agregado por vez. Assim, a manipulação atômica de múltiplos agregados, como em uma transação ACID, deve ser gerenciada pelo código do aplicativo. Caso contrário, pode haver inconsistências nos dados em caso de uma falha. Na modelagem orientada a agregados, é esperado que as necessidades de atomicidade dos dados sejam agrupadas em um único agregado. Em contraste, os bancos de dados NoSQL orientados a grafos implementam ACID ao invés de agregados [Sadalage and Fowler 2019, Rodrigues et al. 2017].

### 3.4.2. Relacionamentos

Os relacionamentos entre entidades representam as associações e interações entre diferentes entidades no banco de dados. Esses relacionamentos são estabelecidos para refletir as conexões lógicas entre as entidades do mundo real. Os relacionamentos em SGBDs NoSQL são semelhantes aos relacionamentos em SBDS relacionais. Porém, existem algumas observações importantes a serem ressaltadas.

#### 3.4.2.1. Relacionamentos em bancos de dados orientados a agregados

Os agregados são a parte central na modelagem de bancos de dados NoSQL orientados a agregados. Eles são modelados para capturar a maioria das necessidades de acesso aos dados de um sistema. Porém, às vezes as aplicações precisam acessar dados relacionados em agregados diferentes [Sadalage and Fowler 2019, Frozza et al. 2022].

No exemplo da Figura 3.7, os agregados de clientes e pedidos estão separados. Contudo, eles devem estar relacionados de alguma maneira para garantir que os clientes encontrem seus pedidos (e vice-versa). Uma maneira simples de conectar agregados separados é o uso de IDs. O agregado Pedido da Figura 3.7 possui o campo idCliente que contem o ID do cliente do pedido. Nesse caso, por exemplo, para obter os dados do cliente através de um pedido, é necessário ler o pedido, descobrir o ID do cliente e acessar o banco de dados novamente para ler os dados do cliente através de seu ID [Sadalage and Fowler 2019, Rodrigues et al. 2017].

Usar IDs para relacionar agregados é uma abordagem simples e eficiente, mas o SGBD não possui ciência dos relacionamentos. Esses SGBDs não tratam restrições de integridade de relacionamentos. Nesse caso, a consistência dos dados deve ser gerenciada pelas aplicações. Por exemplo, a aplicação que precisar atualizar múltiplos agregados de uma vez deve lidar com uma eventual falha durante o processo. Sistemas com muitos relacionamentos são mais adequados em SGBDs relacionais do que em SGBDs NoSQL. SGBDs relacionais permitem que múltiplos registros possam ser modificados em uma transação com as garantias ACID [Sadalage and Fowler 2019, Rodrigues et al. 2017].

### 3.4.3. Relacionamentos em bancos de dados orientados a grafos

A maioria dos bancos de dados NoSQL (de agregados) foi concebida pela necessidade de escalar grandes quantidades de dados em *clusters* (ver Seção 3.2.2.1 para mais detalhes). Por outro lado, os bancos de dados de grafos foram criados com uma intenção diferente: relacionar muitos registros pequenos como muitas conexões complexas. Embora os SGBDs relacionais possam implementar relacionamentos, as junções utilizadas em consultas com relacionamentos podem ser muito custosas, diminuindo o desempenho de consultas. Em contrate, os SGBDs NoSQL de grafos são mais simples. Eles utilizam os conceitos de grafos, ou seja, nodos ligados por arestas. Além disso, eles possuem a implementação dos algoritmos clássicos de grafos, como caminhos de custo mínimo, travessia em largura, travessia em profundidade, etc [Gueidi et al. 2021, Boudaoud et al. 2022, Passos et al. 2023].

Em muitos casos, os SGBDs orientados a grafos pode ser utilizados para resolver problemas em relacionamentos que são muito custosos em SGBDs relacionais. Nesse caso, uma alternativa para o problema é migrar os dados de um SGBD para o outro. Em muito casos, não é necessário migrar o banco inteiro, apenas a parte dos dados indispensáveis para resolver o problema. Diversas propostas na literatura lidam com a problemática de mapeamento entre SGBDs relacionais e SGBDs de grafos [Gueidi et al. 2021, Boudaoud et al. 2022, Passos et al. 2023]. No entanto, existe um consenso quanto às regras de mapeamento do modelo relacional para o modelo de grafos na grande maioria dos trabalhos, como ilustrado na Tabela 3.1. Cada uma das colunas da tabela relaciona os objetos que podem ser equivalentes nos dois tipos de SGBD.

O exemplo da Figura 3.9 considera as regras clássicas de mapeamento relacional-grafo mostradas na Tabela 3.1. Nesse exemplo, as tabelas de um SGBD relacional (Figura 3.9(a)) são convertidas em nodos e arestas (Figura 3.9(b)) de um SGBD

em grafos. Cada uma das tuplas das tabelas concebeu um nodo. Cada um dos nodos possui uma propriedade cujo valor é único (não mostrado na figura) gerado a partir das chaves primárias. Este controle é realizado por uma restrição de integridade implementada no SGBD. Cada uma das arestas foi gerada a partir das chaves estrangeiras. Por exemplo, o nodo do personagem *Homer Jay Simpson* possui um relacionamento *Familiar*, por meio da aresta *idFamilia*, com o nodo *Simpsons* [Passos et al. 2023].

**Tabela 3.1. Regras de mapeamento relacional-grafo [Passos et al. 2023].**

| **Relacional** | tabela | tupla | atributo | chave primária | chave estrangeira |
|---|---|---|---|---|---|
| **Grafo** | rótulo | nodo | propriedade | propriedade "ID" | aresta |

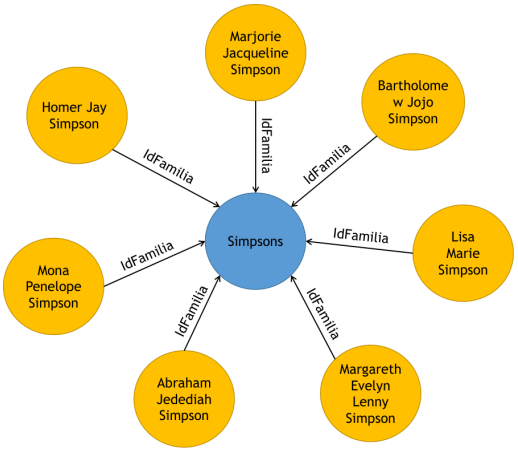
**Personagens**

| id | Nome | IdFamilia |
|---|---|---|
| 1 | Homer Jay Simpson | 1 |
| 2 | Marjorie Jacqueline Simpson | 1 |
| 3 | Bartholomew Jojo Simpson | 1 |
| 4 | Lisa Marie Simpson | 1 |
| 5 | Margareth Evelyn Lenny Simpson | 1 |
| 6 | Abraham Jedediah Simpson | 1 |
| 7 | Mona Penelope Simpson | 1 |

**Familias**

| id | Nome |
|---|---|
| 1 | Simpsons |

(a) Tabelas de banco de dados relacional.




(b) Nodos e arestas de banco de dados em grafo.

**Figura 3.9. Exemplo de mapeamento relacional-grafo [Passos et al. 2023].**

### 3.4.4. Distribuição de dados

Uma das principais motivações para o uso da tecnologia NoSQL é sua facilidade de escalar em Bancos de Dados Distribuídos (BDD). Em um BDD, o banco de dados é dividido em vários servidores conectados por meio de uma rede de computadores de forma transparente aos usuários. Assim, um sistema de BDD armazena fisicamente os dados em várias máquinas (*sites*) e os gerencia de forma independente. Esses sistemas computacionais são também chamados de *clusters* e possuem alta disponibilidade, balanceamento de carga e processamento paralelo. Existem duas maneiras de implementar um *cluster*: fragmentação e replicação [VolDB Documentation 2020, Faerber et al. 2017, Özsu and Valduriez 1996].

#### 3.4.4.1. Fragmentação

Na fragmentação, uma tabela é particionada em fragmentos e cada fragmento pode ser armazenado em diferentes *sites* do sistema. Dessa maneira, um banco de dados

pode possuir muito mais recursos (armazenamento, memória e processamento) do que teria em apenas um *site*. As tabelas podem ser dividas verticalmente (em colunas) ou horizontalmente (em linhas). A Figura 3.10 ilustra um particionamento de tabelas de banco de dados. Por exemplo, a tabela A é divida nas partes A', A" e A"' armazenadas nas partições X, Y e Z, respectivamente. Cada uma das partições é armazenada em um máquina diferente. Uma transação que precise acessar A' deve acessar apenas o *Site* 1. Para uma transação percorrer a tabela A inteira, ela deve acessar os três *sites* [VolDB Documentation 2020, Taft et al. 2014, Özsu and Valduriez 1996].

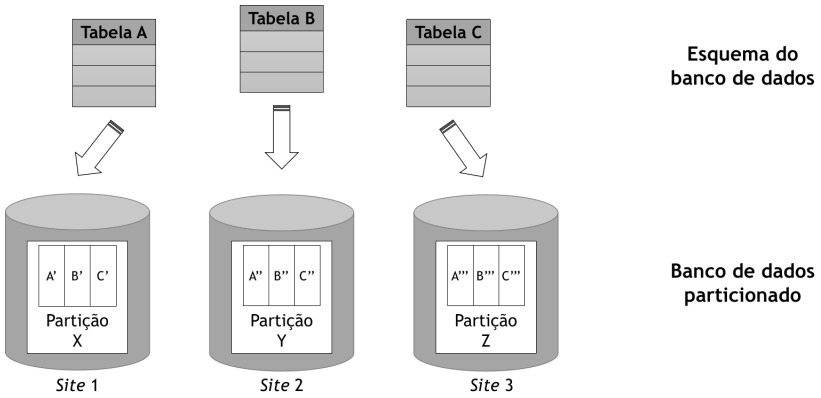



**Figura 3.10. Tabelas de banco de dados particionadas [VolDB Documentation 2020].**

Em um projeto ótimo de BDD, cada transação deve acessar apenas um *site* para obter uma resposta mais rápida. Além disso, a carga de trabalho deve ser balanceada entre os *sites* para não sobrecarregar uma determinada máquina. Por motivos óbvios, esse cenário é raro. Para se aproximar dele, é necessário que os dados acessados em conjunto estejam em apenas uma partição. Nesse contexto, o agregado pode funcionar como uma unidade óbvia de distribuição, uma vez que ele é projetado para combinar dados que são, normalmente, acessados em conjunto [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

Existem diversos fatores que podem influenciar no desempenho do sistema quanto a maneira como os dados devem ser organizados em cada nodo. Se o acesso a um conjunto de agregados é baseado em uma localização física, os dados podem ser colocados próximo ao lugar onde são acessados. Por exemplo, o pedido de um cliente da cidade de São Paulo pode ser armazenado no centro de dados da região sudeste do país. Uma alternativa é organizar os agregados de modo que sejam distribuídos em paralelo entre os nodos para que eles recebam cargas de trabalho iguais. Também pode ser "performático"juntar dados que comumente são acessados em sequência [Faerber et al. 2017, Özsu and Valduriez 1996].

Em muitos casos, a fragmentação segue a lógica do aplicativo. Por exemplo, a primeira letra do sobrenome do usuário pode ser usada como critério de organização dos agregados entre os nodos diferentes. Contudo, isso dificulta a programação dos aplicativos, visto que o código deve gerenciar o acesso aos diversos fragmentos. Além disso, um rebalanceamento no sistema implica em alterar o código do aplicativo e

migrar os dados. Alguns bancos de dados possuem auto fragmentação, deixando o banco de dados responsável pelo gerenciamento do acesso ao fragmento de cada dado [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

#### 3.4.4.2. Replicação

A replicação provê a manutenção de várias cópias idênticas do banco de dados em vários *sites* diferentes. Entre os principais benefícios da replicação de dados estão: (i) a redundância, o que torna o sistema tolerante a falhas; (ii) a possibilidade de um balanceamento de carga do sistema, uma vez que o acesso pode ser distribuído entre as réplicas, e finalmente, (iii) o *backup online* dos dados, já que todas as réplicas estariam sincronizadas [Özsu and Valduriez 1996]. As técnicas de replicação mais conhecidas são a Mestre-Escravo (*Master-Slave*) e a Ponto a Ponto (*Peer-to-Peer* – P2P) .

A replicação Mestre-Escravo (Figura 3.11) possui um nodo primário, denominado mestre, responsável por processar e gerenciar todas as operações de gravação. Ela ainda possui um ou mais nodos secundários, chamados de escravos, que replicam passivamente os dados do mestre e atendem consultas de leitura. Caso aconteça um *crash* no mestre, uma das réplicas pode assumir o seu lugar. Alguns sistemas fazem esse processo de forma automática. Por exemplo, Redis possui um serviço, chamado de Sentinel, que monitora a integridade de seus nodos, detecta falhas e promove um novo nodo mestre se o mestre atual falhar [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

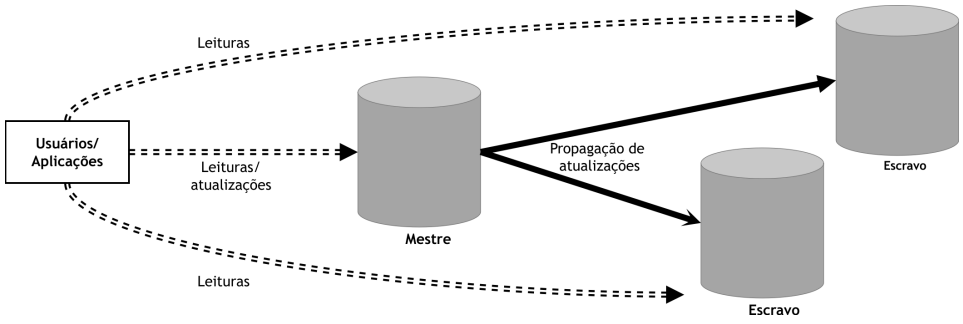



**Figura 3.11. Replicação mestre/escravo.**

A replicação Mestre-Escravo possui as vantagens de distribuir efetivamente a carga de trabalho entre vários nodos, permitindo desempenho otimizado de consulta, maior confiabilidade dos dados e minimização do tempo de inatividade do sistema. Existem ainda variantes do paradigma Mestre-Escravo que oferecem diferentes níveis de desempenho, tolerância a falhas e garantias de consistência [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

Uma das grandes vantagens da replicação é a resiliência de leitura. Nesse caso, se o mestre falhar, as réplicas ainda podem continuar atendendo requisições de leitura. Contudo, as requisições de gravação ficam comprometidas até o mestre ser restaurado ou um novo mestre seja eleito. Para garantir a resiliência de leitura,

é necessário que os caminhos de leitura e gravação sejam diferentes. Por exemplo, as conexões de leitura e gravação devem ser diferentes. Assim, caso a conexão de gravação falhe, a conexão de leitura pode continuar operando. Porém, muitas bibliotecas de interação com bancos de dados não suportam essa funcionalidade [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

Uma desvantagem da replicação é a possibilidade de inconsistência. Clientes diferentes podem ler valores diferentes de escravos diferentes, caso as atualizações não sejam propagadas de forma uniforme em cada escravo. Assim, um cliente pode ler um valor desatualizado. Contudo, essa inconsistência é temporária. Em algum momento as atualizações serão propagados e os clientes poderão ler dados consistentes. Esse intervalo de tempo em que as réplicas estão desatualizadas é chamado de janela de inconsistência. Uma alternativa seria ter apenas uma réplica para evitar leituras inconsistentes nos escravos. Porém, mesmo assim, caso o mestre falhe antes de propagar suas atualizações, o escravo ainda continuará inconsistente. Uma outra hipótese é um cliente não conseguir ver sua própria atualização. Por exemplo, um cliente atualiza seus dados no nodo mestre e os lê em um nodo escravo. Caso a sua atualização ainda não tenha sido propagada para o escravo, ele lerá dados desatualizados [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

A replicação Ponto a Ponto (Figura 3.12) não possui um mestre. Todos os nodos podem receber gravações ou leituras e a perda de um nodo não impede o acesso ao armazenamento de dados. Além disso, novos nodos podem ser inseridos mais facilmente para melhorar o desempenho. Uma rede Ponto a Ponto é mais adequada em armazenamento de dados imutáveis. Inconsistências de leitura podem acontecer, mas elas são temporárias [Sadalage and Fowler 2019, Özsu and Valduriez 1996].

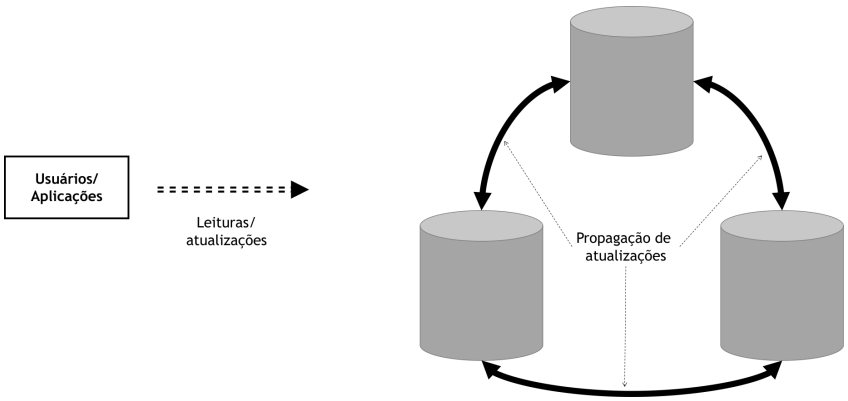



**Figura 3.12. Replicação ponto a ponto.**

A utilização de replicação Ponto a Ponto em sistemas muito atualizados é mais complexa. Caso duas réplicas atualizem o mesmo dado ao mesmo tempo, pode acontecer um conflito de gravação. Por exemplo, dois nodos podem ter o mesmo dado alterado simultaneamente por clientes diferentes. Nesse caso, haverá um impasse de qual atualização deve ser a correta. Para resolver esse problema, pode ser utilizado um servidor (ou servidores) para gerenciar versões de atualização de

dados e identificar a versão corrente. Contudo, o servidor pode se tornar um ponto de falha. Uma alternativa são os protocolos de propagação de atualização, tais como Phase King [Berman and Garay 1993], Chubby [Burrows 2006] e Paxos [Chandra et al. 2007]. Contudo, esse protocolos implicam em aumento do tráfego na rede para confirmar as atualizações. A Seção 3.4.5 discute outras maneiras de como lidar com conflitos em sistemas NoSQL.

### 3.4.5. Consistência e durabilidade

A consistência é o aspecto mais diferente entre os projetos de bancos de dados relacional e NoSQL. Os sistemas relacionais tentam implementar uma consistência forte que evita inconsistência a todo momento. Por exemplo, o SGBD deve garantir que o comprimento máximo do tipo de dados, restrições de chaves primária e estrangeira, etc., sejam absolutamente mantidos. Em contraste, os sistemas NoSQL podem tolerar algum nível de inconsistência, como uma consistência eventual [Davoudian et al. 2018, Rodrigues et al. 2017].

No contexto de sistemas de banco de dados, a consistência garante que a execução das transações não violem nenhuma restrição de integridade do sistema. Para isso, comumente, os SGBDs utilizam algum método de controle de concorrência (ver Seção 3.2.1 para mais detalhes). Por outro lado, a consistência em sistemas distribuídos garante que cada réplica tenha a mesma visão de dados em um determinado momento, independentemente de qualquer atualização de dados realizada pelos clientes. Por exemplo, uma solicitação para qualquer nodo deve retornar exatamente a mesma resposta, dando a impressão de que o sistema possui apenas um nodo [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Os bancos de dados NoSQL orientados a grafos implementam uma consistência baseada no modelo ACID, semelhante aos bancos de dados relacionais. Por outro lado, os SGBDs orientados a agregados (ver Seção 3.4.1) não suportam transações ACID. Ao invés disso, eles suportam atualizações atômicas dentro de um único agregado. Assim, atualizações em um agregado não geram inconsistência lógica. Porém, se o banco de dados for distribuído, pode haver janelas de inconsistência. Sistemas NoSQL podem ter uma janela de inconsistência bem curta, com menos de um segundo [Rodrigues et al. 2017, Rodrigues et al. 2017].

A modelagem da Figura 3.7 permite atualizações consistentes nos agregados cliente e pedido. Assim, atualizações em um cliente ou em itens de um pedido são feitas de forma atômica. Porém, não é possível uma atualização atômica em um cliente e em seus pedidos, visto que cliente e pedido são agregados diferentes. Uma atualização desse tipo deve ser gerenciada pela aplicação, sob pena de gerar alguma inconsistência. A modelagem da Figura 3.8 evita esse problema fazendo do cliente e seus pedidos um único agregado [Rodrigues et al. 2017, Sadalage and Fowler 2019].

#### 3.4.5.1. Consistência de replicação

A replicação de dados pode introduzir a consistência de replicação no sistema, que é significativamente diferente da consistência lógica. A inconsistência de replicação

pode acontecer devido a janela de inconsistência. Caso o sistema falhe durante a propagação de dados, alguma réplica pode não ser atualizada devidamente. Assim, as réplicas vão retornar resultados diferentes a uma mesma solicitação [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Para ilustrar essa consistência de replicação, considere o exemplo de sistema de reservas de quartos de hotel ilustrado na Figura 3.13. Esse sistema possui um banco de dados distribuído entre os nodos 1, 2 e 3, que são réplicas idênticas. Suponha que os usuários 1, 2 e 3 estejam acessando o sistema no mesmo momento e visualizando os dados através dos nodos 1, 2 e 3, respectivamente. Suponha também que existe apenas um quarto disponível no hotel. Os usuários 2 e 3 se conhecem e estão conversando por telefone sobre o último quarto. Enquanto isso, o usuário 1 faz a reserva do quarto. Consequentemente, o sistema atualiza a disponibilidade replicada do quarto. Porém, a propagação dos dados chega no nodo 2 antes do nodo 3. Isso pode ocorrer devido a atrasos na rede, por exemplo. Assim, o usuário 2 vê o quarto reservado enquanto o usuário 3 vê o quarto disponível, ou seja, eles veem respostas diferentes para a mesma solicitação. Contudo, essa leitura inconsistente é temporária. Em algum momento a atualização será totalmente propagada e todos os usuários verão a mesma informação. Essa situação é chamada de consistência eventual, ou seja, em algum momento os nodos podem estar inconsistentes, mas eles acabarão sendo atualizados [Rodrigues et al. 2017, Sadalage and Fowler 2019].

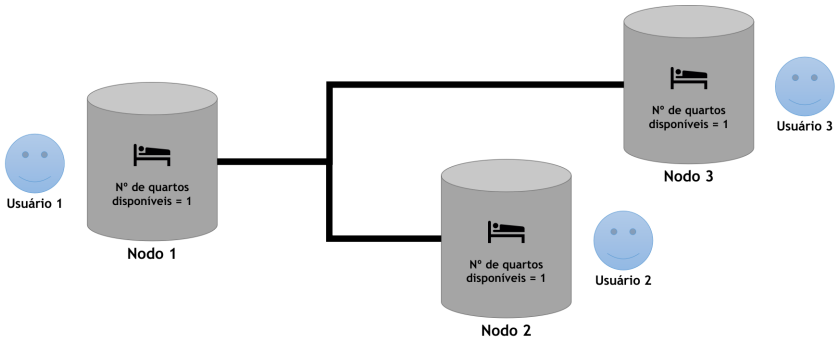



**Figura 3.13. Sistema distribuído de reservas de quartos de hotel.**

Janelas de inconsistência também podem ser um problema para usuários quando a inconsistência é consigo mesmo. Considere um sistema de *blog* implantado em um *cluster* cujos usuários podem fazer postagens e comentários instantâneos. Suponha que esse sistema usa o modelo mestre-escravo. Além disso, o balanceamento de carga do sistema permite escritas apenas no mestre e leitura apenas nos escravos. Nesse cenário, pode acontecer uma situação em que um usuário faz uma postagem, mas não consegue lê-la, dando a impressão de que ela foi pedida. Isso pode acontecer se a propagação de dados do mestre para o escravo possui um atraso. Ou seja, o usuário escreve no mestre, mas não consegue ver sua postagem no escravo devido à janela de inconsistência. Obviamente, a inconsistência é temporária e o usuário verá sua postagem em um futuro próximo, mesmo que talvez demore apenas um segundo. A Seção 3.4.5.2 discute alguns exemplos práticos de inconsistência de replicação e manerias de lidar com ela [Rodrigues et al. 2017, Sadalage and Fowler 2019].

#### 3.4.5.2. Teorema CAP

O balanceamento da inconsistência é algo inevitável em projetos de sistemas distribuídos NoSQL. Além disso, diferentes domínios de projeto possuem diferentes tolerâncias à inconsistência. Para considerar a tolerância a inconsistência, os projetos NoSQL usam frequentemente o teorema CAP com o objetivo relaxar a consistência. CAP é um acrônimo derivado de suas três propriedades principais: *Consistency, Availability, and Partition Tolerance* (Consistência, Disponibilidade e Tolerância a Partições) [Rodrigues et al. 2017, Sadalage and Fowler 2019].

A definição básica do teorema CAP afirma que um sistema distribuído pode garantir apenas duas das suas três características desejadas. Consistência significa que todos os clientes veem os mesmos dados ao mesmo tempo, não importa em qual nodo eles se conectem. Disponibilidade significa que qualquer cliente que fizer uma solicitação de dados obterá uma resposta, mesmo que um ou mais nodos estejam desativados. Tolerância de partição significa que o *cluster* deve continuar a funcionar mesmo se ocorrer uma ou mais falhas de comunicação entre os nodos no sistema [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Os SGBDs relacionais centralizados utilizam um sistema CA (*Consistency and Availability*), ou seja, garantem consistência e disponibilidade, mas não tolerância a partição. Uma vez que existe apenas um nodo no sistema, não é necessário se preocupar com particionamentos. Ainda é possível implementar o *cluster* CA. Porém, se houver um particionamento, todos os nodos ficam incomunicáveis. A implementação de um *cluster* CA é possível, mas muito custosa, visto que deve ser assegurado que ele raramente se particionará [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Considere o cenário do exemplo de reservas de quartos de hotel ilustrado na Figura 3.13. Suponha que o sistema está replicado em um modelo ponto-a-ponto. Para assegurar a consistência, antes do sistema confirmar a reserva do usuário 1 no nodo1, ele deve se comunicar com os outros nodos para que eles concordem com a solicitação do usuário. A atualização é confirmada para o usuário apenas após todos os nodos confirmarem a atualização. Esse protocolo assegura a consistência. Porém, caso a conexão de rede falhe em algum dos nodos, nenhum deles poderá fazer reservas, ou seja, o sistema não proverá disponibilidade [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Manter um *cluster* CA é proibitivamente caro. Assim, os *clusters* devem ser tolerantes a partições. O teorema CAP afirma que só se pode ter duas de suas três propriedades. Porém, um sistema não pode descartar totalmente a consistência ou a disponibilidade. Dessa maneira, um sistema distribuído deve balancear a consistência com a disponibilidade, ou seja, o sistema não deve ser totalmente consistente e nem totalmente disponível [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Para melhorar a disponibilidade, o *cluster* descrito nessa seção poderia utilizar o protocolo mestre-escravo. Suponha que o nodo 1 foi o escolhido como mestre e o usuário 1 fez a reserva do último quarto de hotel. Adicionalmente, suponha que a propagação de atualização do mestre pra os escravos teve algum atraso. Como

consequência, os usuários 2 e 3 verão informações inconsistentes, mas não conseguirão fazer reservas. Porém, ainda haverá um problema caso a conexão entre um dos escravos e o mestre falhe. Por exemplo, caso o nodo 2 (escravo) não consiga se conectar ao nodo 1 (mestre), o usuário 2 não conseguirá fazer reservas mesmo que ainda tenham quartos disponíveis, ou seja, haverá uma falha de disponibilidade [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Uma alternativa para aumentar a disponibilidade é permitir que todos os nodos possam continuar fazendo reservas mesmo que algum deles perca a conexão com os outros nodos. Dessa maneira, usuários em nodos diferentes desconectados poderão fazer a reserva do último quarto de hotel, podendo gerar alguma inconsistência de gravação. Contudo, dependendo da maneira como o hotel trabalha, esse cenário não é um problema. Muitos hotéis trabalham com *overbooking* para lidar com cancelamentos de reservas. Em contraste, alguns hotéis possuem quartos extras para lidar com imprevistos. Existe também a possibilidade de hotéis possuírem parcerias para transferência de clientes entre si. Alguns hotéis podem simplesmente cancelar a reserva caso achem que os custos de cancelamentos é menor do que perder reservas por falha de rede [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Outro cenário em que a janela de inconsistência não é um problema é o sistema de *blog*. Um usuário pode suportar não ver o comentário que acabou de fazer por alguns segundos ou minutos devido a uma falha de rede, por exemplo. Ou esse mesmo usuário não vai se aborrecer se o seu comentário aparecer mais de uma vez devido a uma inconsistência de gravação. Nesse caso a solução é simples: apagar o comentário repetido [Rodrigues et al. 2017, Sadalage and Fowler 2019].

O carrinho de compras de *sites* de *e-comerce* é um exemplo em que se exige alta disponibilidade. Os usuários esperam poder comprar sempre e rapidamente, ou seja, eles podem não tolerar indisponibilidade em seus carrinhos. Caso contrário, eles podem procurar outro *site*. Porém, a implementação de disponibilidade pode resultar em inconsistências de gravação. Por exemplo, a inconsistência pode gerar múltiplos carrinhos de compras. Nesse caso, para resolver o problema, o sistema pode identificar os vários carrinhos existentes e fazer uma operação de união entre seus itens, resultando em um único carrinho. Uma alternativa é permitir que o usuário analise os itens do carrinho antes de finalizar a compra [Rodrigues et al. 2017, Sadalage and Fowler 2019].

O balanceamento da inconsistência está ligada diretamente ao domínio do sistema. As decisões de como lidar com inconsistência pode ser mais associada ao conhecimento dos especialistas na área do sistema do que ao conhecimento dos desenvolvedores do sistema. Os exemplos dos sistemas de reserva de quartos de hotel, comentários em *blog* e carrinho de compras apresentados nessa seção podem permitir alguma inconsistência. Por outro lado, um sistema de transações financeiras pode não tolerar informações inconsistentes ou desatualizadas [Rodrigues et al. 2017, Sadalage and Fowler 2019].

#### 3.4.5.3. Durabilidade

Os bancos de dados em disco utilizam um mecanismo de *buffer* para acessar registros na memória secundária. Quando é necessário acessar um registro no banco de dados, o gerenciador de *buffer* verifica se a página do banco de dados que contém esse registro está na memória. Caso contrário, ele deve copiar o bloco (ou blocos) do disco para a memória principal. Caso seja necessário, atualizações são copiadas da memória para o disco. Para evitar que dados sejam perdidos em caso de uma de falha de sistema (por exemplo, falta de energia), os SGBDs comumente utilizam um mecanismo de recuperação após falhas. Por limitações de espaço, a recuperação de bancos de dados após falhas de sistema não pode ser descrita nesse trabalho. Porém, esse tópico pode ser visto com detalhes no *survey* [Magalhães et al. 2021], no minicurso [Magalhães et al. 2023] e nos tutoriais [Magalhães et al. 2023b] e [Magalhães et al. 2023a].

Parece ser algo impensável desativar o mecanismo de recuperação após falhas de um SGBD, visto que ele garante a durabilidade das alterações no sistema. Contudo, esse mecanismo pode diminuir o desempenho do sistema, uma vez que ele deve fazer acessos ao disco, que é muito mais lento do que a memória. Assim, em alguns casos, relaxar a durabilidade (ou seja, desativar o mecanismo de recuperação) pode ser aceitável em troca de um melhor desempenho. Além disso, para melhorar ainda mais o desempenho, atualizações no banco de dados podem ser mantidas em memória e descarregadas apenas periodicamente para o disco. Obviamente, o preço dessa abordagem é a perda de atualizações em caso de falhas de sistema [Rodrigues et al. 2017, Sadalage and Fowler 2019].

Considere como exemplo um *website* que monitora e armazena informações de sessão de seus usuários. Suponha que o *website* possui um tráfego muito grande. Como consequência, o armazenamento de muitas informações de usuário pode prejudicar a responsividade do *site*. Para evitar esse problema, as informações do que os usuários estão fazendo no *site* são armazenadas temporariamente na memória e descarregadas periodicamente para um banco de dados. Essa abordagem melhora o desempenho do sistema, uma vez que diminui a quantidade de dados enviados para o disco. Porém, ela pode fazer o *site* perder alguns dados de sessão de usuário em caso de falha de sistema. Contudo, isso não seria um problema tão grande quanto uma lentidão no sistema [Rodrigues et al. 2017, Sadalage and Fowler 2019].

#### 3.4.5.4. Map-reduce

Além da mudança na forma de armazenamento, os sistemas distribuídos também mudam a maneira de como processar os dados. Um vez que os dados estão espalhados em máquinas diferentes, pode ser necessário acessar várias máquinas para atender a uma requisição. Além disso, se existirem muitos dados armazenados em um *cluster*, é necessário processá-los de uma maneira eficiente para recuperá-los em um tempo viável. Um cenário ideal em um *cluster* é dividir o processamento entre as várias maquinas de forma paralela. Além disso, deve-se reduzir a quantidade de dados transferidos pela rede, processando tudo que for possível em cada

nodo [Davoudian et al. 2018, Magalhães et al. 2018].

MapReduce é um modelo de execução distribuída projetado para processar grandes volumes de dados em paralelo. A grande vantagem desse modelo é remover a complexidade da programação distribuída. Para isso, ele utiliza duas etapas de processamento: (i) *Map* e (ii) *Reduce*. Na etapa de *Mapping* (mapeamento), os dados são divididos em blocos e processados em tarefas menores em paralelo. Uma lógica de transformação pode ser aplicada a cada bloco de dados. Após a execução das tarefas, a fase *Reduce* (redução) agrega os dados [Davoudian et al. 2018, Magalhães et al. 2018].

Para exemplificar a ideia básica de MapReduce, considere o exemplo de agregado da Figura 3.7. Esse exemplo possui o agregado *Pedido*, em que cada pedido contem itens de produto. Esse agregado atende a requisições que precisem ler todos as informações de um pedido, incluindo seus itens. Porém, o agregado pode não ser "performático"em requisições que necessitem acessar mais de um pedido. Por exemplo, uma consulta que busque a receita total da venda de um produto no último trimestre. Essa busca irá precisar acessar vários agregados em vários nodos, possivelmente tornado a recuperação da informação muito lenta. Esse tipo de cenário demanda a utilização de MapReduce [Sadalage and Fowler 2019].

A etapa *map* é uma função que recebe como entrada um único agregado e retorna alguns pares chave-valor. O exemplo da Figura 3.14 atende a uma requisição que deseja saber a receita total de *notebooks* vendidos. O *mapping* recebe um agregado de pedido e retorna os itens correspondentes ao produto *notebook*. Essa função de *mapping* é aplicada a todos os agregados de pedido. Cada uma das aplicações da função *map* pode ser realizada de forma paralela, uma vez que elas são independentes [Sadalage and Fowler 2019].

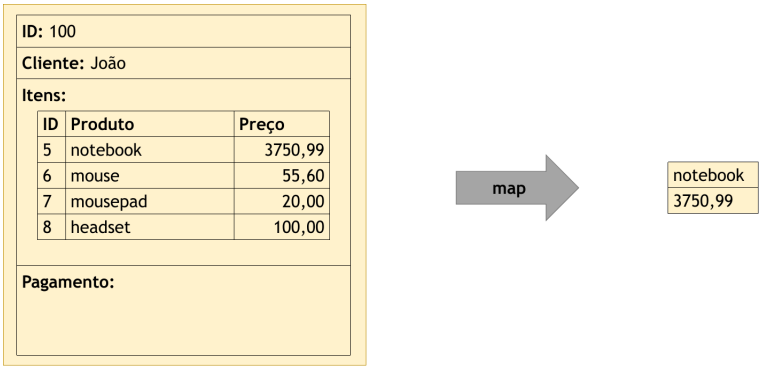



**Figura 3.14. A função *map* lê agregados de pedido do banco de dados e produz pares chave-valor [Sadalage and Fowler 2019].**

Após a etapa *map*, a função *reduce* recebe múltiplos resultado do mapeamento e combina seus valores. A Figura 3.15 representa a etapa *reduce* após a etapa *map* da Figura 3.14. A redução faz o somatório dos valores dos *notebooks* mapeados para calcular o valor da receita total [Sadalage and Fowler 2019]. Por questões de limitação de espaço e para fins didáticos, os exemplos apresentados possuem poucos dados, mas os agregados pode ser milhares ou milhões.

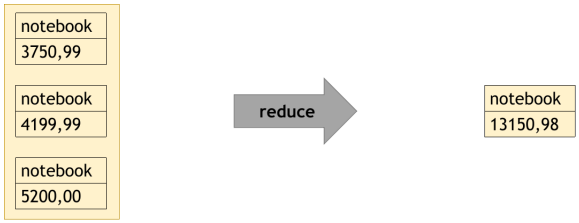



**Figura 3.15. A função *reduce* agrega pares chave-valor [Sadalage and Fowler 2019].**

## 3.5. Ajuste de desempenho em bancos de dados NoSQL

O projeto de banco de dados é essencial para o desenvolvimento de sistemas funcionais, confiáveis e seguros que possibilitem acessar e armazenar dados de forma eficiente. Contudo, a quantidade de dados e informações em uma empresa aumenta a cada dia. Além disso, suas necessidades podem mudar com o tempo. Assim, embora os SGBDs sejam projetados para suportar altas demandas de seus sistemas, o desempenho do sistema pode se tornar não aceitável com o passar do tempo [Mohammad 2016].

A fim de fornecer a continuidade do bom funcionamento do sistema e a qualidade dos serviços, é necessário o ajuste (*tuning*) do sistema de banco de dados. (*Tuning*) é um processo que envolve a configuração e ajuste de vários parâmetros e estruturas para otimizar o desempenho do SGBD. A falta de ajustes periódicos no SGBD podem trazer tempos de resposta lentos, processamento emperrado, dificuldades para conduzir operações corporativas que antes levavam menos tempo, etc [Mohammad 2016].

Essa seção discute as noções básicas de ajuste de desempenho apenas no banco de dados NoSQL MongoDB, por questões de limitação de espaço. A seção apresenta ferramentas, comandos e boas práticas que podem ajudar no desempenho do SGBD. Embora os comandos e ferramentas apresentados sejam específicos, a abordagem utilizada é genérica e pode ser facilmente reproduzida em outros SGBDs, inclusive em SGBDs relacionais.

### 3.5.1. Fatores que influenciam no desempenho

Para um bom desempenho das aplicações é necessário analisar o desempenho do sistema. A queda de desempenho de bancos de dados ocorre em função de vários fatores, tais como: falta de estratégias de acesso eficientes, pouca disponibilidade de *hardware*, número excessivo de conexões abertas e/ou projetos inadequados. Essa seção discute os principais tópicos a serem analisados em caso de queda do desempenho de um SGBD [MongoDB Documentation 2024].

#### 3.5.1.1. Bloqueios

Bloqueios são utilizados para manter a consistência dos dados durante o processamento concorrente de transações. Por exemplo, se algum usuário desejar modificar determinado dado no banco, ele deve bloquear esse dado antes da modificação para

impedir que outros usuários sobrescrevam as alterações dele sobre o dado. É certo que operações longas de bloqueio podem gerar longas filas de espera por acesso a dados, podendo degradar drasticamente o desempenho de um sistema [MongoDB Documentation 2024].

Os bloqueios realizados no MongoDB podem ser verificados através do comando *serverStatus*. Adicionalmente, esse comando provê uma visão geral do estado do sistema com informações sobre o servidor, configurações de segurança, armazenamento, memória, conexões, rede, dentre outras. A Listagem 3.32 ilustra como utilizar o comando *serverStatus* [MongoDB Documentation 2024].

```
1  db.serverStatus()
2
```

**Listagem 3.32. Comando para retornar informações gerais do estado do sistema.**

Por meio do comando *serverStatus*, podem ser obtidas informações gerais e específicas de bloqueios através dos atributos *globalLock* e *locks*, respectivamente. A Figura 3.16 mostra um exemplo de informações obtidas por meio do comando *serverStatus* (Listagem 3.32) [MongoDB Documentation 2024].

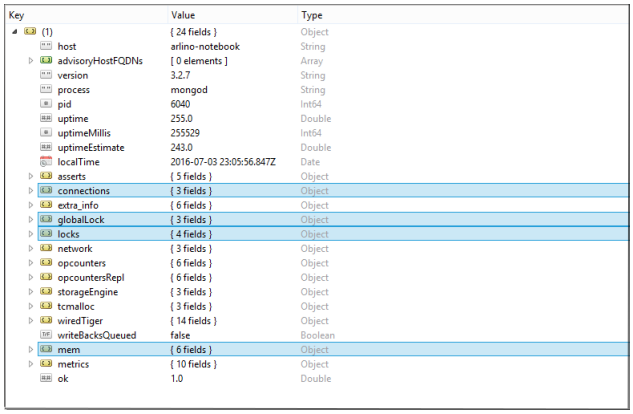



**Figura 3.16. Informações gerais sobre o sistema obtidas através do comando *serverStatus*.**

### 3.5.1.2. Memória

MongoDB possui um módulo de gerenciamento de memória que dado um conjunto de dados, o módulo irá alocar toda a memória necessária a esses dados no sistema. O comando *serverStatus* (Listagem 3.32) exibe em seu atributo *men* (Figura 3.16) a quantidade de memória mapeada e em uso no sistema, dentre outras informações [MongoDB Documentation 2024].

Se a quantidade de memória mapeada for maior que a quantidade de memória física disponível, poderão ocorrer *page faults*. Uma *page fault* acontece quando um dado que deve ser acessado não está localizado correntemente na memória física e, por esse motivo, é necessário fazer acesso ao disco. Muitas *page faults* podem prejudicar o desempenho do sistema devido ao aumento de acesso ao disco. O atributo *extra_info* da saída de *serverStatus* (Figura 3.16) possui informações sobre

as *page faults* ocorridas no MongoDB. Aumentar a quantidade de memória RAM pode ajudar a reduzir a frequência de *page faults*. Porém, se isso não for possível, a implementação de um *cluster* (ou adição de um novo nodo em um *cluster* já implantado) pode resolver o problema mediante o compartilhamento de dados e de recursos disponíveis [MongoDB Documentation 2024].

#### 3.5.1.3. Número de conexões abertas

Em alguns casos, o número de conexões entre as aplicações e o SGBD pode prejudicar a habilidade do servidor em responder às requisições. Se, em dado momento, o número de requisições for muito grande, o sistema poderá ter problema em atendê-las. Nesse caso, é necessário aumentar os recursos do servidor. Um *cluster* de banco de dados com problemas em atender a muitas conexões pode adicionar novos nodos parar aumentar os recursos disponíveis. Se há muitas requisições de leitura, é necessário aumentar o número nodos réplicas para distribuir o processamento entre os membros do *cluster*. Por outro lado, se há muitas requisições de escrita, é necessário aumentar o número de nodos de compartilhamento para dividir a carga de trabalho entre as instâncias do banco [MongoDB Documentation 2024].

### 3.5.2. Avaliação de operações

Essa seção descreve estratégias de como monitorar e avaliar o desempenho de operações no MongoDB com o objetivo de identificar gargalos de desempenho no sistema [MongoDB Documentation 2024].

#### 3.5.2.1. Coleta de operações lentas

A ferramenta *Database Profiler* pode ajudar a monitorar operações lentas no MongoDB. Essa ferramenta é capaz de identificar e armazenar operações lentas para poderem ser analisadas posteriormente. Ela possui níveis de configuração mostrados na Tabela 3.2. O nível de armazenamento de operações por meio da ferramenta *Database Profiler* pode ser ajustado através do comando *setProfilingLevel*, como ilustrado na Listagem 3.33. Nesse exemplo, o número 1 (um) indica que foi escolhido por armazenar apenas as operações lentas do sistema [MongoDB Documentation 2024].

| Nível | Configuração |
|---|---|
| 0 | Não armazena nada. |
| 1 | Armazena apenas as operações lentas. |
| 2 | Armazena todas as operações. |

**Tabela 3.2. Níveis de configuração da ferramenta *Database Profiler*.**

```
1    db.setProfilingLevel(1)
2
```

**Listagem 3.33. Comando de configuração da ferramenta *Database Profiler***

Opcionalmente, o comando *setProfilingLevel* pode possuir um segundo parâmetro numérico para informar qual o tempo mínimo (em milissegundos) que *Database Profiler* deve considerar como necessário para uma operação ser lenta. Por padrão, uma operação é considerada lenta se executar em, pelo menos, 100ms. Após as operações lentas serem identificadas e armazenadas, o comando *system.profile.find* (Listagem 3.34) deve ser utilizado para recuperá-las. A Figura 3.17 exibe uma lista com operações armazenadas por *Database Profiler*. Nessa figura, o atributo das informações da segunda operação está expandido e é possível observar que ela se trata de uma consulta feita sobre a coleção chamada de *foo* do banco *test* [MongoDB Documentation 2024].

```
1    db.system.profile.find()
2
```

**Listagem 3.34. Comando para recuperar as operações armazenadas por *Database Profiler*.**

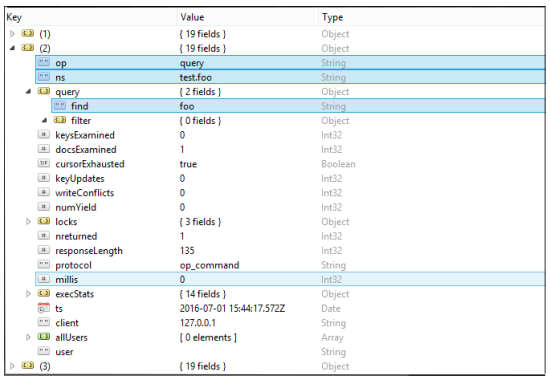



**Figura 3.17. Lista de operações armazenadas por meio de *Database Profiler*.**

A ferramenta *Database Profiler* deve ser utilizada com cuidado, pois pode afetar negativamente no desempenho do sistema. Por esse motivo, ela deve ser ativada apenas em intervalos de tempo estratégicos que não prejudiquem (ou prejudiquem minimamente) o sistema em produção [MongoDB Documentation 2024].

### 3.5.2.2. Operações em processo de execução

Em alguns casos, pode ser necessário monitorar as consultas enquanto estão sendo executadas. Para essa abordagem, MongoDB possui a função *currentOp*. Esse comando retorna as operações que estão em processo de execução no sistema. Por padrão, *currentOp* retorna apenas as operações importantes correntemente em execução. Opcionalmente, esse comando pode ter os parâmetros exibidos na Tabela 3.3 que o fazem retornar conjuntos de operações diferentes [MongoDB Documentation 2024].

A Figura 3.35 exibe o comando *currentOp* com o parâmetro *true* que permite mostrar todas as operações em execução no MongoDB. A Figura 3.18 ilustra um resultado obtido por *currentOp*, no qual a operação 6 está expandida exibindo informações sobre uma operação de conexão chamada de *conn3* cujo atributo *active* está

com o valor *false*, indicando que a conexão está ociosa [MongoDB Documentation 2024].

| Parâmetro | Retorno |
|---|---|
| *true* | Todas as operações, incluindo as de sistema e conexões ociosas. |
| registro BSON | Conjunto de operações especificadas no registro. |

**Tabela 3.3. Parâmetros do comando *currentOp*.**

```
db.currentOp(true)

```

**Listagem 3.35. Comando para recuperar todas as operações em execução no MongoDB.**

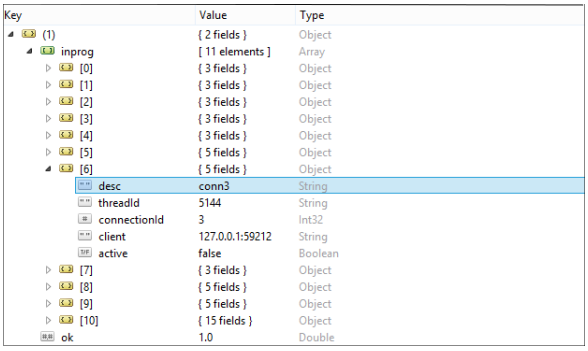



**Figura 3.18. Lista de todas as operações em execução no MongoDB.**

O comando *currentOP* ainda permite utilizar um BSON como parâmetro para restringir sua busca por operações específicas, tais como operações esperando por um bloqueio, ativas sem rendimento, ativas em um banco de dados específico, que utilizam índices, dentre outras. O exemplo do comando da Listagem 3.36 exibirá apenas as operações de escrita em execução que estão esperando por um bloqueio [MongoDB Documentation 2024].

```
db.currentOp({
  "waitingForLock": true,
  $or: [
  {"op": {"$in": ["insert", "update", "remove"] }},
  {"query.findandmodify": {$exists: true}}
  ]
})

```

**Listagem 3.36. Comando que retorna as operações em execução no MongoDB que estão esperando por um bloqueio.**

### 3.5.2.3. Plano de execução de consulta

Uma operação de banco de dados é formada por um conjunto de etapas para alcançar os resultados esperados. Esse conjunto de etapas é chamado de plano de

execução. Para uma determinada operação, podem haver vários planos de execução. O otimizador de consultas do SGBD avalia diferentes planos de execução e escolhe o que considera mais eficiente [MongoDB Documentation 2024].

O comando *explain* é capaz de retornar informações sobre o plano de execução de uma consulta selecionada pelo otimizador do MongoDB. Além disso, *explain* pode retornar informações estatísticas sobre o plano de execução da consulta. As informações retornadas por esse comando dependem dos parâmetros passados, descritos na Tabela 3.4. O parâmetro *queryPlanner* é o padrão de execução de *explain* [MongoDB Documentation 2024].

**Tabela 3.4. Modos de execução do comando *explain*.**

| Parâmetro | Retorno |
|---|---|
| *queryPlanner* | Informações sobre o plano de execução escolhido. |
| *executionStats* | Informações sobre o plano de execução escolhido, incluindo suas informações estatísticas. |
| *allPlansExecution* | Informações sobre o plano de execução escolhido e informações estatísticas desse plano e de outros planos de execução encontrados pelo otimizador no processo de seleção. |

Para exemplificar o uso do comando explain, considere a consulta C1 da Listagem 3.37. Essa consulta retorna todas a tuplas da coleção *foo*. A Listagem 3.38 mostra como utilizar *explain* para retornar o plano de execução escolhido para a consulta C1. A Figura 3.19 exibe o plano de execução da consulta C1 gerado por *explain*. Nesse plano, a estratégia utilizada para retornar as tuplas de C1 foi *CollScan* (percorrer a coleção inteira) [MongoDB Documentation 2024].

```
1    db.foo.find()
2
```

**Listagem 3.37. Consulta C1.**

```
1    db.foo.find().explain()
2
```

**Listagem 3.38. Comando para retornar o plano de execução da consulta C1.**

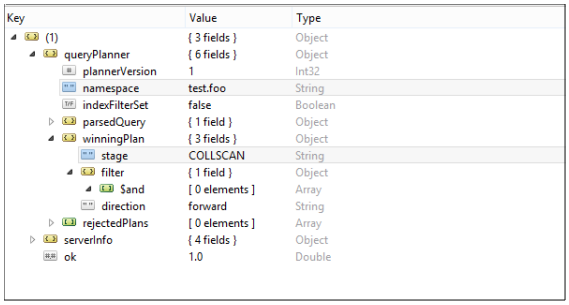



**Figura 3.19. Plano de execução da consulta C1.**

### 3.5.3. Otimização de desempenho de consultas

#### 3.5.3.1. Criação de índices

Índices são estruturas que utilizam estratégias de acesso rápido aos dados. Consultas lentas podem ter um melhor desempenho se índices forem criados para seus atributos, visto que percorrer índices pode ser mais rápido do que percorrer uma coleção de dados. A criação de índices deve ser muito bem planejada. Apesar de um índice melhorar o desempenho de uma determinada consulta, ele pode prejudicar o desempenho da execução de outras operações. Como consequência, o desempenho geral do sistema pode degradar [MongoDB Documentation 2024].

A Listagem 3.39 exemplifica como criar um índice para o campo *author_name* da coleção *posts*. Nesse caso, buscas pelo nome de autores na coleção *posts* podem melhorar o desempenho. Entretanto, operações de atualização no atributo *author_name* devem ficar mais lentas devido ao trabalho adicional de atualização na estrutura de índice. Índices são mais adequados a um sistema que faça muito mais consultas do que atualizações [MongoDB Documentation 2024].

```
1     db.posts.createIndex({author\_name: 1})
2
```

**Listagem 3.39. Criação de índice em uma coleção no MongoDB.**

#### 3.5.3.2. Limite do número de resultados de uma consulta

Limitar o número de resultados de uma consulta é uma boa estratégia para reduzir a quantidade de dados trafegados pela rede, pois retornar todos os dados de uma consulta pode ser desnecessário. Às vezes, o usuário pode não precisar utilizar todas as linhas retornadas por uma consulta. Além disso, carregar todas as linhas de uma consulta de uma só vez pode prejudicar a visualização dos resultados. No MongoDB, o número de resultados de uma consulta pode ser limitado utilizando a função *limit* cujo funcionamento é similar ao do comando *Limit* no SQL. A função *limit* recebe como parâmetro o número de linhas a serem retornadas. Na Figura 3.40 há um exemplo de como utilizar *limit* para uma consulta retornar apenas os 10 primeiros elementos [MongoDB Documentation 2024].

```
1     db.posts.find().limit(10)
2
```

**Listagem 3.40. Comando *limit* aplicado a uma consulta no MongoDB.**

Uma alternativa para reduzir o tráfego de dados na rede é o uso de projeções. Uma projeção permite que apenas um subconjunto dos campos de uma coleção seja retornado por uma consulta. O uso de projeções pode trazer ganhos de desempenho a uma consulta, visto que apenas os campos necessários são retornados, diminuindo a quantidade de dados recuperados. Por exemplo, na consulta da Listagem 3.41, ao invés de retornar a coleção *posts* inteira, apenas os campos *timestamp*, *title*, *author*, e *abstract* são recuperados da coleção [MongoDB Documentation 2024].

```
db.posts.find(
{
  timestamp: 1 ,
  title: 1 ,
  author: 1 ,
  abstract: 1
}
).sort({timestamp: -1})

```

**Listagem 3.41. Consulta com uso de projeção.**

#### 3.5.3.3. Uso de *hints* para selecionar um determinado índice

O otimizador do SGBD tenta selecionar o melhor plano de execução para uma consulta. Por exemplo, o otimizador pode escolher um plano de execução que usa os índices mais adequados à execução da consulta. Porém, nem sempre o otimizador consegue fazer isso. Nesses casos, uma alternativa é forçar o otimizador a selecionar um determinado plano de execução por meio de *hints* (dicas). *Hints* pode ser fornecidos pelos desenvolvedores nas consultas, a fim de melhorar o desempenho delas [MongoDB Documentation 2024].

Como exemplo, a consulta da Listagem 3.42 possui uma *hint* que faz a consulta utilizar o índice do campo *age* da coleção *users*. Alternativamente, uma *hint* pode especificar o nome do índice a ser utilizado na execução de uma consulta. A Listagem 3.43 ilustra uma *hint* que força a consulta sobre a coleção *users* utilizar o índice *age_1* [MongoDB Documentation 2024].

```
db.users.find().hint({ age: 1 })

```

**Listagem 3.42. *Hint* que força a consulta a utilizar o índice de um determinado campo de uma coleção.**

```
db.users.find().hint("age\_1")

```

**Listagem 3.43. *Hint* que força a consulta a utilizar um determinado índice.**

## 3.6. Conclusões

Esse minicurso elucidou as principais questões relacionadas a sistemas de bancos de dados NoSQL, principalmente as relacionadas ao projeto e ajuste de desempenho. Para isso, o trabalho provê uma visão geral da tecnologia NoSQL, como produtos, linguagens de acesso, manipulação e processamento dos dados. Em seguida foram discutidos os principais tópicos a se pensar em um projeto de sistema NoSQL: agregados, relacionamentos, distribuição de dados, consistência e durabilidade e, por fim, execução distribuída de grandes quantidades de dados. O minicurso ainda discutiu noções básicas de ajuste de desempenho em bancos de dados NoSQL, mostrando ferramentas, comandos e boas práticas que podem ajudar no desempenho.

## Referências


[Ambler 2000] Ambler, S. W. (2000). Mapping objects to relational databases. *White Paper, Ronin International*.

[Benymol and Sajimon 2017] Benymol, J. and Sajimon, A. (2017). Exploring the merits of nosql: A study based on mongodb. *International Conference on Networks & Advances in Computational Technologies (NetACT)*, pages 20–22.

[Berman and Garay 1993] Berman, P. and Garay, J. A. (1993). Cloture votes: n/4-resilient distributed consensus in t+ 1 rounds. *Mathematical systems theory*, 26:3–19.

[Bernstein and Goodman 1981] Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.

[Bernstein et al. 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[Bondi 2000] Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*, pages 195–203. ACM.

[Boscarioli et al. 2006] Boscarioli, C., Bezerra, A., BENEDICTO, M. d., and Delmiro, G. (2006). Uma reflexão sobre banco de dados orientados a objetos. In *Congresso de Tecnologias para Gestão de Dados e Metadados do Cone Sul, Paraná, Brasil*. sn.

[Boudaoud et al. 2022] Boudaoud, A., Mahfoud, H., and Chikh, A. (2022). Towards a complete direct mapping from relational databases to property graphs. In Fournier-Viger, P., Yousef, A. H., and Bellatreche, L., editors, *Model and Data Engineering: 11th International Conference, MEDI 2022, Cairo, Egypt, November 21-24, 2022, Proceedings*, volume 13761 of *Lecture Notes in Computer Science*, pages 222–235. Springer.

[Burrows 2006] Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350.

[Cassandra Database 2024] Cassandra Database (2024). Open Source Nosql Database. Disponível em: "https://cassandra.apache.org/". Acessado em: 01/05/2024.

[Cassandra Documentation 2024] Cassandra Documentation (2024). Welcome to apache cassandras documentation! Disponível em: "https://cassandra.apache.org/doc/latest/". Acessado em: 01/05/2024.

[Chandra et al. 2007] Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407.

[Davoudian et al. 2018] Davoudian, A., Chen, L., and Liu, M. (2018). A survey on nosql stores. *ACM Comput. Surv.*, 51(2):40:1–40:43.

[Elmasri and Navathe 2000] Elmasri, R. and Navathe, S. B. (2000). *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman.

[Faerber et al. 2017] Faerber, F., Kemper, A., Larson, P., Levandoski, J. J., Neumann, T., and Pavlo, A. (2017). Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130.

[Frozza et al. 2022] Frozza, A. A., Schreiner, G. A., and dos Santos Mello, R. (2022). Projeto de bancos de dados nosql. *37th Brazilian Symposium on Data Bases*.

[Gray et al. 1975] Gray, J., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. (1975). Granularity of locks in a large shared data base. In Kerr, D. S., editor, *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, pages 428–451. ACM.

[Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[Gueidi et al. 2021] Gueidi, A., Gharsellaoui, H., and Ahmed, S. B. (2021). Towards unified modeling for nosql solution based on mapping approach. In Watróbski, J., Salabun, W., Toro, C., Zanni-Merk, C., Howlett, R. J., and Jain, L. C., editors, *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 25th International Conference KES-2021, Virtual Event / Szczecin, Poland, 8-10 September 2021*, volume 192 of *Procedia Computer Science*, pages 3637–3646. Elsevier.

[Härder and Reuter 1983] Härder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317.

[Harrison 2015] Harrison, G. (2015). *Next Generation Databases: NoSQLand Big Data*. Apress Berkeley, CA.

[Heuser 2009] Heuser, C. A. (2009). *Projeto de banco de dados: Volume 4 da Série Livros didáticos informática UFRGS*. Bookman Editora.

[Hill 1990] Hill, M. D. (1990). What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21.

[Imielinski and Jr. 1984] Imielinski, T. and Jr., W. L. (1984). The relational model of data and cylindric algebras. *J. Comput. Syst. Sci.*, 28(1):80–102.

[Lazoti 2016] Lazoti, R. (2016). *Armazenando dados com Redis*. Casa do Código.

[Magalhães et al. 2023] Magalhães, A., Brayner, Â., and Monteiro, J. M. (2023). Bancos de dados em memória e suas estratégias de recuperação após falhas. *Sociedade Brasileira de Computação*.

[Magalhães et al. 2023a] Magalhães, A., Brayner, A., and Monteiro, J. M. (2023a). Main memory database instant recovery. In *Anais Estendidos do XXXVIII Simpósio Brasileiro de Bancos de Dados*, pages 255–269. SBC.

[Magalhães et al. 2023b] Magalhães, A., Brayner, A., and Monteiro, J. M. (2023b). Main memory database recovery strategies. In Das, S., Pandis, I., Candan, K. S., and Amer-Yahia, S., editors, *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 31–35. ACM.

[Magalhães et al. 2018] Magalhães, A., Monteiro, J. M., and Brayner, A. (2018). Gerenciamento e processamento de big data com bancos de dados em memória. In *I Jornada latino-americana de atualização em informática , JOLAI 2017, São Paulo, SP, Brazil, 2018*.

[Magalhães et al. 2021] Magalhães, A., Monteiro, J. M., and Brayner, A. (2021). Main memory database recovery: A survey. *ACM Comput. Surv.*, 54(2):46:1–46:36.

[Marquesone 2016] Marquesone, R. (2016). *Big Data: Técnicas e tecnologias para extração de valor dos dados*. Editora Casa do Código.

[Michael et al. 2007] Michael, M. M., Moreira, J. E., Shiloach, D., and Wisniewski, R. W. (2007). Scale-up x scale-out: A case study using nutch/lucene. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8. IEEE.

[Mohammad 2016] Mohammad, S. (2016). *Self-tuning for cloud database clusters*. PhD thesis, University of Magdeburg, Germany.

[MongoDB Database 2024] MongoDB Database (2024). MongoDB: The Developer Data Platform | MongoDB. Disponível em: "https://www.mongodb.com". Acessado em: 01/05/2024.

[MongoDB Documentation 2024] MongoDB Documentation (2024). Mongodb documentation. Disponível em: "https://www.mongodb.com/docs/". Acessado em: 01/05/2024.

[Neo4J Database 2024] Neo4J Database (2024). Neo4j Graph Database & Analytics. Disponível em: "https://neo4j.com". Acessado em: 01/05/2024.

[Neo4J Documentation 2024] Neo4J Documentation (2024). Neo4j documentation. Disponível em: "https://neo4j.com/docs/". Acessado em: 01/05/2024.

[Özsu and Valduriez 1996] Özsu, M. T. and Valduriez, P. (1996). Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128.

[Paniz 2016] Paniz, D. (2016). *NoSQL: Como armazenar os dados de uma aplicação moderna*. Casa do Código.

[Passos et al. 2023] Passos, P. V., de Oliveira, L. S., Schreiner, G. A., Machado, V. L., and dos Santos Mello, R. (2023). Sql2neo: Uma interface de acesso SQL para o neo4j. In *Proceedings of the 38th Brazilian Symposium on Databases, SBBD 2023, Belo Horizonte, MG, Brazil, September 25-29, 2023*, pages 179–191. SBC.

[Phillips et al. 2015] Phillips, D. J., McGlaughlin, A., Ruth, D., Jager, L. R., and Soldan, A. (2015). Graph theoretic analysis of structural connectivity across the spectrum of Alzheimer's disease: The importance of graph creation methods. *NeuroImage: Clinical*, 7:377–390.

[Ramakrishnan and Gehrke 2003] Ramakrishnan, R. and Gehrke, J. (2003). *Database management systems (3. ed.)*. McGraw-Hill.

[Redis Database 2024] Redis Database (2024). Redis - the real-time data platform. Disponível em: "https://redis.io". Acessado em: 01/05/2024.

[Redis Documentation 2024] Redis Documentation (2024). Redis documentation! Disponível em: "https://redis.io/docs/latest/". Acessado em: 01/05/2024.

[Redis Insight 2024] Redis Insight (2024). Redis insight the best redis gui. Disponível em: "https://studio3t.com". Acessado em: 01/05/2024.

[Rodrigues et al. 2017] Rodrigues, W. B. et al. (2017). Nosql: a análise da modelagem e consistência dos dados na era do big data.

[Sadalage and Fowler 2019] Sadalage, P. J. and Fowler, M. (2019). *NoSQL essencial: um guia conciso para o mundo emergente da persistência poliglota*. Novatec Editora.

[Silva et al. 2021] Silva, L. F. C., Riva, A. D., and Rosa, G. A. (2021). *Banco de Dados Não Relacional*. Minha Biblioteca.

[Strauch et al. 2011] Strauch, C., Sites, U.-L. S., and Kriha, W. (2011). Nosql databases. *Lecture Notes, Stuttgart Media University*, 20(24):79.

[Studio 3T 2024] Studio 3T (2024). Studio 3T, the Ultimate GUI for MongoDB. Disponível em: "https://studio3t.com". Acessado em: 01/05/2024.

[Taft et al. 2014] Taft, R., Mansour, E., Serafini, M., Duggan, J., Elmore, A. J., Aboulnaga, A., Pavlo, A., and Stonebraker, M. (2014). E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proceedings of the VLDB Endowment*, 8(3):245–256.

[Vieira et al. 2012] Vieira, M. R., FIGUEIREDO, J. M. d., Liberatti, G., and Viebrantz, A. F. M. (2012). Bancos de dados nosql: conceitos, ferramentas, linguagens e estudos de casos no contexto de big data. *Simpósio Brasileiro de Bancos de Dados*, 27:1–30.

[VolDB Documentation 2020] VolDB Documentation (2020). Voldb documentation. Disponível em: "https://docs.voltdb.com". Acessado em: 01/05/2024.