UNIVERSIDADE FEDERAL DE SANTA MARIA
COLÉGIO POLITÉCNICO DA UFSM
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

Cezar Augusto Crummenauer

MOTIVAÇÕES PARA O USO DE SISTEMAS NOSQL

Santa Maria, RS
2020

Cezar Augusto Crummenauer

MOTIVAÇÕES PARA O USO DE SISTEMAS NOSQL

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Sistemas para Internet da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Tecnólogo em Sistemas para Internet**

Orientador: Prof. Dr. (UFSM) Daniel Lichtnow

Santa Maria, RS

2020

Crummenauer, Cezar Augusto

Motivações para o uso de Sistemas NoSQL / por Cezar Augusto Crummenauer. – 2020.
74 f.: il.; 30 cm.

Orientador: Daniel Lichtnow
Trabalho de Conclusão de Curso - Universidade Federal de Santa Maria, Colégio Politécnico da UFSM, Curso Superior de Tecnologia em Sistemas para Internet, RS, 2020.

1. NoSQL. 2. Dados. 3. Modelo Relacional. I. Lichtnow, Daniel. II.Motivações para o uso de Sistemas NoSQL.

E-mail: crummenauerca@gmail.com

Cezar Augusto Crummenauer

MOTIVAÇÕES PARA O USO DE SISTEMAS NOSQL

Trabalho de Conclusão de Curso apresentado ao Curso Superior de Tecnologia em Sistemas para Internet da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de **Tecnólogo em Sistemas para Internet**

Aprovado em 02 de Outubro de 2020:

Daniel Lichtnow, Dr. (UFSM)
(Presidente/Orientador)

Marcos Alexandre Rose Silva, Dr. (UFSM)

Rafael Gressler Milbradt, Dr. (UFSM)

Santa Maria, RS

2020

DEDICATÓRIA

Aos meus pais, por tudo que já dedicaram a mim
Aos meus irmãos, por todas as batalhas que enfrentamos juntos

AGRADECIMENTOS

Agradeço aos meus professores de Sistemas para Internet por todo o conhecimento compartilhado, em especial ao professor Daniel Lichtnow, pela orientação e paciência durante o desenvolvimento deste trabalho.

Agradeço também aos meus amigos e colegas Augusto, Diana, Enrico, Guilherme, Íris, Lucas, Marcelo, Marcos, Natiele e Vanessa, pelo apoio e companheirismo.

“A persistência é o caminho do êxito”
(CHARLES CHAPLIN)

RESUMO

MOTIVAÇÕES PARA O USO DE SISTEMAS NOSQL

AUTOR: CEZAR AUGUSTO CRUMMENAUER
ORIENTADOR: DANIEL LICHTNOW

A expansão acelerada da internet e o aumento natural do uso de aplicações como *e-commerces, streaming* de multimídia e redes sociais têm gerado um volume cada vez maior de dados. Nesse sentido, pode-se observar que sistemas NoSQL estão se tornando uma nova tendência, uma vez que, quando comparados com os sistemas baseados no Modelo Relacional, possibilitam a implementação de estruturas de dados mais flexíveis e dinâmicas, que podem impactar positivamente na eficiência das aplicações. Apesar disso, mesmo que os sistemas NoSQL tragam novas possibilidades, cada um deles foi criado para lidar com necessidades específicas e podem ser inadequados para a construção de aplicações com necessidades e contextos diferentes. Considerando essa situação, este trabalho apresenta um estudo sobre conceitos, tecnologias e possibilidades sobre persistência poliglota de dados, visando esclarecer quais são os principais contextos em que os diferentes sistemas de dados podem ser utilizados ou combinados para melhor atender às crescentes demandas na área de gerenciamento de dados.

Palavras-chave: NoSQL. Dados. Modelo Relacional.

ABSTRACT

MOTIVATIONS FOR THE USE OF NOSQL SYSTEMS

AUTHOR: CEZAR AUGUSTO CRUMMENAUER
ADVISOR: DANIEL LICHTNOW

The accelerated expansion of the internet and the natural increase in the use of applications such as e-commerce, multimedia streaming and social networks have generated an increasing volume of data. In this sense, it can be seen that NoSQL systems are becoming a new trend, once, when compared to systems based on the Relational Model, they enable the implementation of more flexible and dynamic data structures, which can positively impact the efficiency of applications. However, even though NoSQL systems brings new possibilities, each one was created to deal with specific needs and may be unsuitable for building applications with different needs and contexts. Considering this situation, this work presents a study on concepts, technologies and possibilities on multilingual data persistence, aiming to clarify whose are the main contexts in which different data systems can be used or combined to handle better the growing demands in the area of data management.

Keywords: NoSQL. Data. Relational Model.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

UFSM	Universidade Federal de Santa Maria
NoSQL	*Not Only SQL*
BI	*Business Intelligence*
DW	*Data Warehouse*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
CAP	*Consistency, Availability and Partition tolerance*
DBMS	*Database Management System*
SGBD	Sistema de Gerenciamento de Banco de Dados
RDBMS	*Relational Database Management Systems*
SQL	*Structured Query Language*
CQL	*Cassandra Query Language*
CLI	*Command-line Interface*
XML	*Extensible Markup Language*
JSON	*JavaScript Object Notation*
SPOF	*Single Point of Failure*
BDOO	Banco de Dados Orientado a Objetos
BDOR	Banco de Dados Objeto-Relacionais
GUI	*Graphical User Interface*

SUMÁRIO

1 INTRODUÇÃO

"As atuais perspectivas computacionais, vindas sobretudo da Web, têm gerado novas demandas relacionadas ao gerenciamento de dados, principalmente em termos de volume, heterogeneidade e dinamismo" (CLAUDINO; SOUZA; SALGADO, 2015, p. 1).

Diversas aplicações atuais produzem e manipulam grandes quantidades de dados, denominados *Big Data*, e os bancos de dados tradicionais, especialmente os baseados no modelo relacional, não são adequados para esse tipo de situação (SCHREINER; DUARTE; SANTOS MELLO, 2015).

Essas novas condições sobre gerenciamento de dados, principalmente nas aplicações que necessitam de alto desempenho, têm fomentado o surgimento e a utilização dos sistemas denominados NoSQL, uma vez que esses possibilitam implementar estruturas de dados mais flexíveis e dinâmicas.

Mas isso não indica que os sistemas relacionais serão completamente substituídos pelas tecnologias NoSQL. Na realidade, o desenvolvimento de software está entrando em uma era de persistência poliglota, onde as aplicações utilizam um conjunto de diferentes tecnologias para o gerenciamento dos dados (SADALAGE; FOWLER, 2013).

Cada uma dessas tecnologias possui características convenientes para tratar diferentes problemas de persistência de dados. Dificilmente, no contexto de *Big Data*, será recomendado utilizar somente uma delas para implementar todo o gerenciamento de dados em uma aplicação.

Neste trabalho, são apresentados estudos sobre os principais sistemas NoSQL, considerando suas características, classificações e as principais motivações para a crescente adoção deles no gerenciamento de dados na atualidade. O trabalho também explora possibilidades relacionadas com a persistência poliglota de dados.

1.1 CONTEXTUALIZAÇÃO INICIAL

O acesso à internet, assim como o uso de computadores e dispositivos móveis, vem aumentando de forma acelerada. De acordo com o Internet World Stats [1], a internet cresceu mais de 1800% nos últimos 20 anos, atingindo em torno de 4 bilhões de pessoas, ou seja, mais da metade da população mundial.

[1] O Internet World Stats é um site que exibe estatísticas relacionadas com pesquisa de mercado da Internet - https://www.internetworldstats.com/. Acessado em 23 de outubro de 2019

Diante dessa realidade, o aumento natural do uso de aplicações como *e-commerces, streaming* de multimídia e redes sociais têm gerado um volume de dados cada vez maior e, por consequência, vem desafiando os provedores de serviços ao demandar abordagens mais robustas e rápidas para o armazenamento e manipulação desses dados.

1.2 OBJETIVOS

1.2.1 Objetivo geral

O objetivo deste trabalho é investigar as principais motivações para o uso de alguns dos bancos de dados NoSQL mais relevantes no mercado, em quais situações eles são convenientes e como eles podem ser utilizados para aprimorar o funcionamento das aplicações, fazendo comparações com o tradicional modelo relacional.

1.2.2 Objetivos específicos

Como objetivos específicos para o desenvolvimento deste trabalho, podem ser listados os seguintes:

- Apresentar os principais conceitos relacionados com gerenciamento de dados;
- Expor as principais motivações para a utilização dos diferentes tipos sistemas NoSQL;
- Testar alguns dos principais bancos de dados NoSQL, fazendo comparações com o tradicional modelo relacional;
- Construir uma aplicação com persistência poliglota para apresentar algumas possibilidades relacionadas com a utilização de diferentes bancos de dados em uma única aplicação;
- Abordar sobre os bancos de dados multimodelos (sistemas que agregam diferentes modelos de dados em apenas um banco de dados), que podem facilitar a implementação da persistência poliglota em aplicações;

1.3 JUSTIFICATIVA

Com a expansão do acesso à informações e serviços pela internet e do consequente aumento no volume de dados gerados na interação entre os usuários e as aplicações, o aprimo-

ramento das abordagens na gerência de dados está se tornando cada vez mais importante.

A utilização de abordagens e tecnologias mais adequadas para cada contexto pode reduzir o consumo de recursos nos sistemas computacionais e, consequentemente, permitir que esses recursos economizados sejam redirecionados para atender outras demandas.

É nesse sentido que os bancos de dados NoSQL podem oferecer possibilidades para uma gerência de dados mais conveniente, permitindo que os provedores de serviços possam atender uma quantidade maior de usuários de forma mais eficiente e sustentável.

1.4 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado em cinco capítulos:

- O capítulo 2 apresenta alguns conceitos importantes relacionados com gerência de dados;
- O capítulo 3, por sua vez, apresenta os tipos de sistemas NoSQL, juntamente com um exemplar de tecnologia relevante para cada um desses tipos e como eles podem ser comparados com o Modelo Relacional. O capítulo 3 também apresenta critérios gerais para a escolha de bancos de dados de acordo com as necessidades de cada aplicação;
- O capítulo 4 apresenta possibilidades relacionadas com a persistência poliglota de dados (utilização de diferentes bancos de dados em uma aplicação). Nessa parte do trabalho também é exemplificada a implementação de persistência poliglota em uma aplicação;
- Por fim, o capítulo 5 expõe as considerações finais deste trabalho.

Até aqui, foram expostos os objetivos do trabalho juntamente com uma introdução inicial sobre o assunto abordado por ele. No próximo capítulo, serão apresentados os conceitos básicos sobre bancos de dados e como as novas demandas nesta área fomentaram a utilização de novas tecnologias para o gerenciamento de dados.

2 CONCEITOS BÁSICOS

Como o estudo sobre bancos de dados representa uma área bastante abrangente, este capítulo visa apresentar os conceitos básicos sobre esse assunto juntamente com contextualizações históricas relevantes. O presente capítulo aborda tópicos relacionados com os bancos de dados relacionais, suas características e como as novas demandas de dados impulsionaram o surgimento de tecnologias alternativas para o gerenciamento de dados: os sistemas NoSQL.

2.1 SISTEMA DE GERENCIAMENTO DE BANCO DE DADOS

Um banco de dados ou Sistema de Gerenciamento de Banco de Dados (SGBD) — do inglês Database Management System (DBMS) — é um conjunto de programas que busca prover o gerenciamento conveniente e eficiente de dados, que, por sua vez, formam coleções de informações como listas telefônicas, registros de vendas, dados cadastrais de alunos e cursos universitários, etc (SILBERSCHATZ; SUNDARSHAN; KORTH, 2016).

Os primeiros SGBDs foram desenvolvidos por volta de 1960 com base nos sistemas de arquivos (*file system*) para permitir a manipulação de dados com segurança, tratamento de falhas e possibilitar o controle de concorrência, restrições e integridade dos dados (LÓSCIO; OLIVEIRA; PONTES, 2011). Eles surgiram como uma opção mais apropriada para gerir informações que eram frequentemente armazenadas em simples arquivos de computador ou até mesmo em folhas de papel.

2.1.1 Bancos de dados Relacionais

No início dos anos 70, surgiram os primeiros bancos de dados relacionais e esses se firmaram como solução comercial para o gerenciamento de dados com estruturas fixas e bem definidas como as encontradas em aplicações para controle de estoque ou folhas de pagamento (LÓSCIO; OLIVEIRA; PONTES, 2011).

Os bancos de dados relacionais utilizam esquemas de tabelas para armazenar os dados e as relações entre esses dados. As tabelas, campos de dados e os relacionamentos entre dados precisam ser planejados e estruturados antes da inserção dos dados. A estruturação e manipulação dos dados em bancos de dados relacionais normalmente é realizada por meio da linguagem SQL (*Structured Query Language*).

A padronização de conceitos, a base formal e a facilidade proporcionada pela linguagem SQL são alguns dos motivos para o sucesso dos bancos de dados relacionais (LÓSCIO; OLIVEIRA; PONTES, 2011). Os bancos de dados relacionais estão em amadurecimento contínuo e ainda são predominantes como solução para a gerência de dados na atualidade.

2.1.1.1 Transações ACID

Uma transação corresponde a uma sequência de operações que são executadas como uma unidade de trabalho. Normalmente os bancos de dados relacionais trabalham com transações ACID, que possibilitam a manipulação consistente de quaisquer combinações de dados em uma única transação (SADALAGE; FOWLER, 2013). De acordo com Silberschatz, korth e Sudarshan (2016), ACID é um acrônimo para Atomicidade, Consistência, Isolamento e Durabilidade conforme será detalhado a seguir:

- Atomicidade: Todas as operações de uma transação devem ser executadas corretamente. Se uma ou mais operações da transação falharem, a transação inteira deve ser cancelada;
- Consistência: A execução de transações devem manter estados consistentes do banco de dados. Se houver falhas na transação, todas as alterações realizadas devem ser desfeitas;
- Isolamento: Transações, quando executadas simultaneamente, não podem interferir nas operações umas das outras;
- Durabilidade: As alterações realizadas por uma transação bem sucedida, devem ser persistidas para que possam ser acessadas posteriormente.

As transações ACID, de forma geral, através de um conjunto complexo de operações internas, buscam preservar a integridade e consistência dos dados e suas relações em um sistema de banco de dados.

2.2 NOVAS DEMANDAS DE DADOS

Passou-se muitos anos no mundo da computação, ocorreram muitas mudanças em termos de linguagens, arquiteturas, plataformas e processos, mas uma coisa permaneceu constante nesse tempo: sistemas relacionais armazenam dados (SADALAGE; FOWLER, 2013).

De forma geral, a simplicidade do modelo relacional contribuiu para a sua grande adoção e disseminação, mas diante da necessidade de manipular tipos complexos de dados, outras

soluções como os bancos de dados orientados a objetos (BDOO) e os bancos de dados objeto-relacionais (BDOR) foram propostas (LÓSCIO; OLIVEIRA; PONTES, 2011). Os BDOOs e os BDORs, por possuírem mapeamentos de dados próximos das linguagens de programação orientadas a objetos, aumentam a produtividade no desenvolvimento de software.

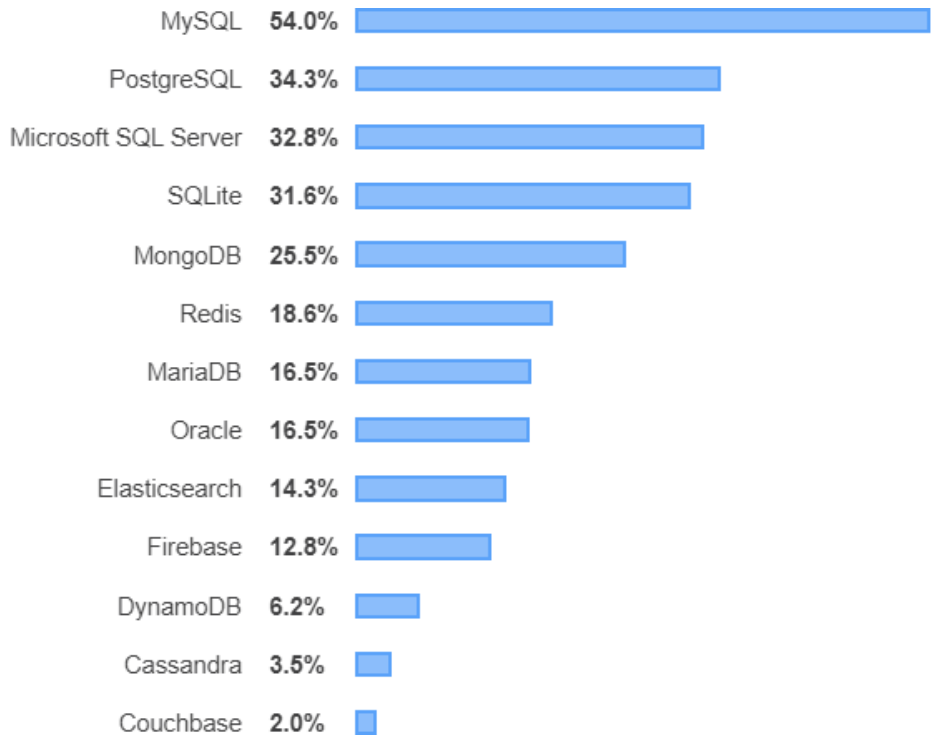
Posteriormente, foram criadas as primeiras propostas de bancos de dados NoSQL (*Not Only SQL* ou, em português, Não apenas SQL) para aprimorar o desempenho e a manipulação dados não estruturados ou semi-estruturados provenientes do surgimento da Web (LÓSCIO; OLIVEIRA; PONTES, 2011). A nomenclatura NoSQL não está relacionada a um modelo de dados específico, mas sim ao agrupamento de tecnologias que se diferenciam da abordagem do modelo relacional (CLAUDINO; SOUZA; SALGADO, 2015).

As novas demandas de dados relacionadas com as aplicações Web estão fomentando a busca de novas tecnologias. Uma pesquisa realizada com aproximadamente 90 mil desenvolvedores pelo Stack Overflow[2] em 2019 aponta que os sistemas NoSQL estão se destacando apesar dos bancos de dados relacionais continuarem sendo os mais utilizados na atualidade.

Essa pesquisa indica que tecnologias NoSQL como o MongoDB e Redis já atingem uma fatia considerável dos sistemas mais utilizados (25,5% e 18,6% respectivamente) conforme pode ser observado na Figura 2.1. Somado a isso, através da mesma pesquisa do Stack Overflow, como pode-se perceber na Figura 2.2, existe uma quantidade considerável de desenvolvedores que desejam utilizar tecnologias NoSQL como o MongoDB, Redis e Cassandra.

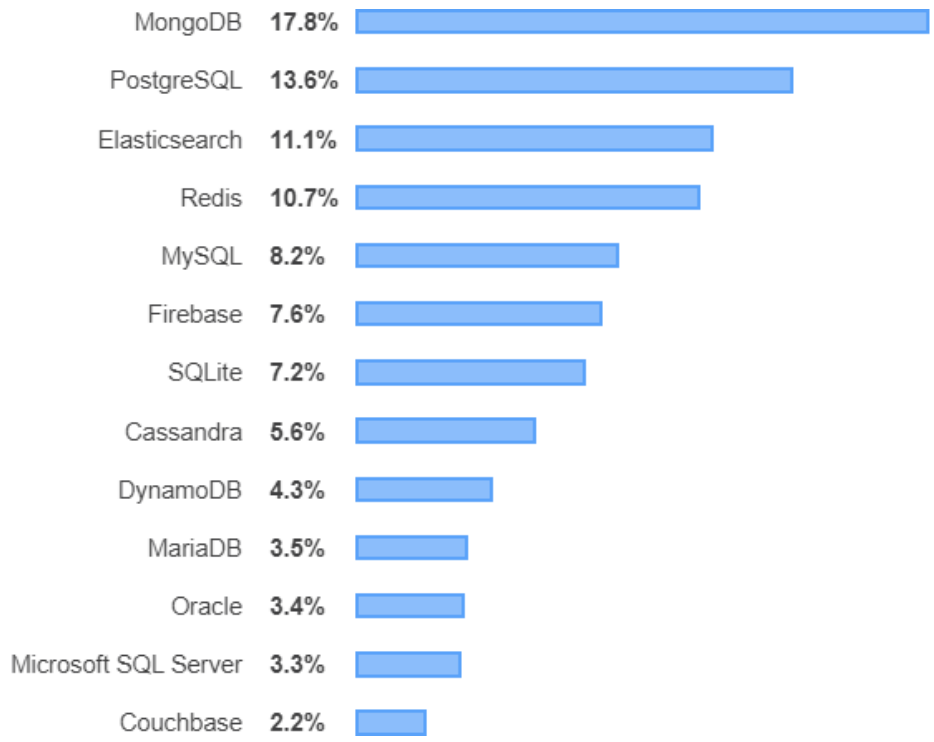
[2] Stack Overflow é um site de perguntas e respostas relacionadas com o desenvolvimento de sistemas de software - https://pt.stackoverflow.com/. Acessado em 20 de outubro de 2019

Figura 2.1: Bancos de dados mais usados em 2019



Fonte: https://insights.stackoverflow.com/survey/2019

Figura 2.2: Bancos de dados mais desejados em 2019



Fonte: https://insights.stackoverflow.com/survey/2019

2.2.1 Bancos de dados NoSQL

Os sistemas NoSQL possibilitam a criação de estruturas de dados mais dinâmicas, flexíveis e intuitivas. Eles têm sido amplamente utilizados por empresas como Facebook e Google para atender as demandas de escalabilidade, disponibilidade e tratamento de dados não estruturados (LÓSCIO; OLIVEIRA; PONTES, 2011).

Ao contrário dos sistemas relacionais, comumente não é necessário definir as estruturas de dados antes de utilizá-las nos sistemas NoSQL. Essa abordagem é um atrativo devido a melhoria de produtividade no desenvolvimento de software incremental e ágil, uma vez que tende a facilitar mudanças nas estruturas de dados já criadas para atender as alterações e incrementos nos projetos de software.

Os bancos de dados relacionais oferecem uma série de recursos para controle de concorrência, redundância, integridade e segurança dos dados armazenados. Esse grande conjunto de recursos incluindo as propriedades ACID pode ser mais do que o necessário para suprir as necessidades de alguns tipos dados em determinadas situações (POKORNY, 2013).

Frequentemente, os sistemas NoSQL buscam potencializar outras características importantes como desempenho e escalabilidade ao afrouxar uma ou mais propriedades ACID e outros recursos de proteção conforme as necessidades de cada aplicação.

É importante salientar que os sistemas NoSQL não estão sendo desenvolvidos com o objetivo de substituir os bancos de dados do modelo relacional. Na realidade, eles objetivam oferecer soluções de persistência de dados que podem ser aplicadas em situações específicas.

2.2.2 Escalabilidade

A escalabilidade é como um sistema recebe melhorias de desempenho para suportar uma demanda maior. A escalabilidade pode ser vertical, que consiste no aumento da capacidade e tecnologia de hardware em um sistema computacional, ou pode ser horizontal, que, por outro lado, consiste em unir o poder de diferentes sistemas computacionais em um só (LÓSCIO; OLIVEIRA; PONTES, 2011).

Os sistemas relacionais, devido a sua complexidade de operação e implementação, são mais custosos em termos de processamento e são normalmente executados sobre um único sistema computacional. Para aumentar o desempenho desses sistemas, geralmente é usado escalonamento vertical, ou seja, procura-se substituir ou incrementar as tecnologias de hardware

(processador, memória principal e secundária, etc) com o intuito de aumentar a capacidade e desempenho do banco de dados relacional.

Os sistemas NoSQL, por não possuírem estruturas fixas e serem mais flexíveis em termos de consistência e propriedades ACID, tendem a permitir escalonamento horizontal mais facilmente. Esse escalonamento pode ocorrer pela replicação ou particionamento de dados entre servidores distribuídos permitindo que as aplicações sejam preparadas para o aumento de usuários e volume de dados mesmo que seja um aumento súbito como ocorre, por exemplo, nas *black fridays*, lançamentos de resultados do ENEM e em aplicações que se difundem rapidamente como as redes sociais.

O escalonamento horizontal tende a ser mais barato do que o vertical por permitir a criação de *clusters* com sistemas computacionais mais modestos e caracteristicamente diferentes, mas que juntos podem superar um supercomputador caro e centralizado proveniente de um escalonamento vertical.

Além disso, o número de servidores envolvidos no escalonamento horizontal pode crescer indefinidamente e oferecer desempenho sob demanda (otimizar os custos operacionais ao ativar ou desativar servidores envolvidos no *cluster* de acordo com a demanda atual) enquanto o escalonamento vertical tende a ser limitado pelo máximo de incrementos possíveis pela tecnologia de hardware disponível.

2.2.2.1 Teorema CAP

A sigla CAP representa três características:

- Consistência (Consistency);
- Disponibilidade (Availability);
- Tolerância à partição (Partition tolerance).

O teorema CAP, segundo Sadalage e Flowler (2013), é a possibilidade de obter somente duas dessas três características simultaneamente. A Tolerância à partição ocorre quando os dados de uma aplicação estão distribuídos em diferentes servidores interligados (*cluster*) e continua funcionando mesmo que nem todos os servidores estejam acessíveis por causa de possíveis falhas. Em sistemas com tolerância à partição, é preciso fazer trocas entre consistência ou disponibilidade dos dados conforme é explicado a seguir:

- Para manter a consistência, mas comprometendo a disponibilidade, o sistema pode recusar a gravação de novos dados, já que esses não poderão ser sincronizados com todos os servidores. As consultas aos dados já existentes podem ocorrer normalmente mesmo que esses possam estar desatualizados.

- Para aumentar a disponibilidade, mas permitir que possíveis inconsistências ocorram, novos dados podem ser recebidos mesmo que nem todos os servidores estejam acessíveis. Quando a comunicação é restabelecida, os dados poderão ser sincronizados entre os servidores e possíveis conflitos de alterações precisam ser tratados (*merge*).

Neste capítulo, foram vistos os conceitos e histórico básicos sobre bancos de dados, facilidades proporcionadas pela utilização dos bancos de dados relacionais e como as novas demandas de dados fomentaram a utilização de novas tecnologias como os sistemas NoSQL. No próximo capítulo, serão apresentados os diferentes tipos de bancos de dados NoSQL e onde cada um desses tipos é mais adequado, fazendo comparações com o tradicional modelo relacional e, posteriormente, apresentando critérios para a escolha de bancos de dados.

3 TIPOS DE BANCOS DE DADOS NOSQL

Os sistemas NoSQL podem ser classificados, de acordo com o modelo de dados, em: Chave-valor, Documentos, Grafos e Colunas. Cada um desses tipos será detalhado juntamente com um exemplo de tecnologia mais relevante de acordo com a adoção no mercado e com a pesquisa do Stack Overflow apresentada anteriormente no Capítulo 2.

3.1 MODELO CHAVE-VALOR

O modelo Chave-valor é o que possui a representação mais simples dentre os sistemas de bancos de dados NoSQL (CLAUDINO; SOUZA; SALGADO, 2015). Basicamente, esse tipo de banco de dados é composto por um conjunto de chaves que são associadas a um único valor (LÓSCIO; OLIVEIRA; PONTES, 2011).

Nesse modelo, os dados são mantidos em uma única estrutura de dados e, devido a sua maior simplicidade, facilita a escalabilidade horizontal. A Tabela 3.1 exemplifica o armazenamento de chaves associadas aos seus valores.

Tabela 3.1: Exemplo de organização dos dados em um sistema Chave-valor

chave única	**valor**
curtidas	101
visualizações	1194

Memcached[3], Riak[4] e o Redis[5] são exemplos de tecnologias NoSQL do tipo Chave-valor. O Redis será abordado com mais detalhes a seguir.

3.1.1 Banco de dados Redis

Redis é um banco de dados em memória, de código aberto e que suporta tipos de dados como strings, listas, conjuntos e hashs (REDIS, 2019). Por possibilitar o armazenamento em memória, o Redis pode servir de cache em situações onde os valores sofrem constantes leituras ou atualizações como o número de curtidas ou visualizações em postagens nas redes sociais.

O uso do cache pode reduzir o número de operações em disco e, por consequência, impactar positivamente no desempenho da aplicação e reduzir os custos relacionados com a

[3] https://memcached.org/
[4] https://riak.com/
[5] https://redis.io/

computação em nuvem. Entretanto, devido a volatilidade da memória, perdas de dados podem ocorrer se houver falha no sistema computacional ou uma queda na energia elétrica.

Como pôde-se perceber, esse é um caso onde uma das propriedades ACID, a durabilidade, sofre um afrouxamento objetivando melhorar o desempenho. Para amenizar os riscos relacionados com perdas de dados, é possível ajustar a quantidade de escritas que o Redis realiza em memória antes de gravar os dados em disco (SAMPAIO; OLIVEIRA KNOP, 2015).

A Figura 3.1 exemplifica como inserir, recuperar, incrementar e remover um valor no banco de dados Redis utilizando os comandos SET, GET, INCR e DEL respectivamente.

Figura 3.1: Manipulação de um valor no Redis

```
127.0.0.1:6379> SET curtidas 101
OK
127.0.0.1:6379> GET curtidas
"101"
127.0.0.1:6379> INCR curtidas
(integer) 102
127.0.0.1:6379> GET curtidas
"102"
127.0.0.1:6379> DEL curtidas
(integer) 1
```

Fonte: Elaborada pelo autor.

Somado a isso, a Figura 3.2, demonstra como manipular mais de um valor por vez utilizando os comandos MSET, MGET e DEL respectivamente para inserir, recuperar, e remover mais de um valor por comando.

Figura 3.2: Manipulação de múltiplos valores no Redis

```
127.0.0.1:6379> MSET curtidas 101 visualizacoes 1194
OK
127.0.0.1:6379> MGET curtidas visualizacoes
1) "101"
2) "1194"
127.0.0.1:6379> INCR curtidas
(integer) 102
127.0.0.1:6379> INCR visualizacoes
(integer) 1195
127.0.0.1:6379> MGET curtidas visualizacoes
1) "102"
2) "1195"
127.0.0.1:6379> DEL curtidas visualizacoes
(integer) 2
```

Fonte: Elaborada pelo autor.

O Redis permite que suas chaves sejam associadas com valores complexos como listas e conjuntos. A Figura 3.3 apresenta a manipulação de uma lista no Redis.

O comando LPUSH insere valores na esquerda da lista. O comando RPUSH, por outro lado, insere valores na direita da lista. A sequência de utilizações dos comandos LPUSH e RPUSH usados na Figura 3.3 formam a seguinte lista: [0, 1, 2, 3].

O comando LRANGE permite que um intervalo definido dentro da lista seja recuperado. Usar "LRANGE lista 1 -2" recupera os itens a partir do segundo item da lista até o penúltimo.

Figura 3.3: Manipulando listas de dados no Redis

```
127.0.0.1:6379> LPUSH lista 1
(integer) 1
127.0.0.1:6379> LPUSH lista 0
(integer) 2
127.0.0.1:6379> RPUSH lista 2
(integer) 3
127.0.0.1:6379> RPUSH lista 3
(integer) 4
127.0.0.1:6379> LRANGE lista 1 -2
1) "1"
2) "2"
127.0.0.1:6379> DEL lista
(integer) 1
```

Fonte: Elaborada pelo autor.

A Figura 3.4 apresenta a manipulação de um conjunto no banco de dados Redis. O comando SADD permite definir, desconsiderando repetições, um conjunto de valores.

Figura 3.4: Manipulando conjuntos de dados no Redis

```
127.0.0.1:6379> SADD conjunto 1 1 1 2 2 3 2 3 1 2
(integer) 3
127.0.0.1:6379> SMEMBERS conjunto
1) "1"
2) "2"
3) "3"
```

Fonte: Elaborada pelo autor.

A Figura 3.5 apresenta como listar as chaves ainda válidas através do comando KEYS. No exemplo dessa figura, a única chave ainda válida é "conjunto", tendo em vista que as outras chaves, das figuras anteriores sobre Redis, foram removidas após o uso. Um conjunto de dados também pode ser removido usando DEL.

É possível definir o tempo de expiração de uma chave. A expressão "EX 60", por exemplo, ao final de uma operação com SET, indica que a chave expira 60 segundos depois de sua definição. Por fim, o comando FLUSHALL pode ser utilizado para limpar todas as chaves do banco de dados Redis.

Figura 3.5: Definindo chaves que expiram no Redis

```
127.0.0.1:6379> KEYS *
1) "conjunto"
127.0.0.1:6379> SET temp60 "Expira em 1 minuto" EX 60
OK
127.0.0.1:6379> FLUSHALL
OK
```

Fonte: Elaborada pelo autor.

3.1.1.1 Modelo Chave-valor (Redis) versus modelo Relacional

Os bancos de dados do modelo Chave-valor como o Redis podem ser muito convenientes para o armazenamento de dados transitórios. Como ele é um banco de dados em memória, proporciona alta velocidade de leitura e escrita de dados e pode ser usado para reduzir o acesso a disco. Entretanto, por possuir uma estruturação de dados limitada, não é adequado para propósito geral como ocorre com os bancos de dados do modelo Relacional.

3.2 MODELO EM DOCUMENTOS

O modelo em documentos possui implementação semelhante ao modelo chave-valor por possuir chaves e valores, mas nesse caso, a dados são organizados em coleções de documentos com notação XML ou JSON ao invés de utilizar uma única estrutura de dados (MCMURTRY et al., 2013 apud CLAUDINO, SOUZA e SALGADO, 2015).

Como esse modelo também não depende de um esquema rígido como ocorre nos bancos de dados relacionais, é possível alterar a estrutura de um documento sem influenciar outros documentos da mesma coleção (LÓSCIO; OLIVEIRA; PONTES, 2011). Permitir a criação de documentos diferentes uns dos outros pode facilitar o armazenamento de dados heterogêneos como as diversas variações de características nos produtos de *e-commerces*.

Como exemplos de bancos de dados desse tipo podem ser citados o CouchDB[6], Ra-

[6] https://couchdb.apache.org/

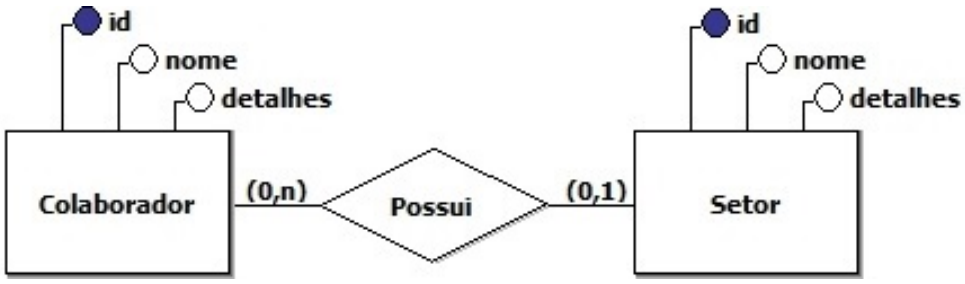
venDB[7] e o MongoDB[8]. O MongoDB será melhor detalhado a seguir.

3.2.1 Banco de dados MongoDB

O MongoDB é um banco de dados que, além de permitir o armazenamento de documentos do tipo JSON em estruturas flexíveis e dinâmicas, permite a ordenação e filtragem dos dados com base em qualquer campo dos documentos armazenados nele (MONGODB, 2019). O MongoDB, por armazenar dados em notação JSON, funciona de forma harmoniosa com linguagens de programação que suportam objetos em notação JSON nativamente como o JavaScript.

A Figura 3.6 apresenta o modelo conceitual que será utilizado para fazer alguns experimentos com o MongoDB.

Figura 3.6: Modelo conceitual para experimentos com MongoDB



Fonte: Elaborada pelo autor.

Para introduzir os experimentos, a Figura 3.7 apresenta a inserção de documentos simples (sem relacionamentos entre os documentos) em notação JSON no banco de dados MongoDB. Nessa figura, existem quatro comandos:

- O primeiro comando insere um documento na coleção setores;
- O segundo comando, por sua vez, insere um *array* com dois documentos na coleção colaboradores;
- O terceiro e quarto comandos, através do find(), exibem todos os documentos, da forma que foram inseridos, das coleções setores e colaboradores respectivamente. Os identificadores são gerados automaticamente quando omitidos, eles são úteis para, por exemplo, criar relações entre os dados.

[7] https://ravendb.net/
[8] https://www.mongodb.com/

Figura 3.7: Inserindo documentos no MongoDB (sem relacionamentos)

```
db.setores.insert({
    "_id": ObjectId("5dcabc2007c9ee115fea8c01"),
    "nome": "Docência em Informática",
    "detalhes": "Ensinar informática no ensino técnico e superior..."
})

db.colaboradores.insert([{
    "_id": ObjectId("5dcabc2007c9ee115fea8c02"),
    "nome": "Daniel",
    "detalhes": "Possui doutorado em Ciência da Computação..."
}, {
    "_id": ObjectId("5dcabc2007c9ee115fea8c03"),
    "nome": "Juçara",
    "detalhes": "Possui especialização em Sistemas de Computação..."
}])

db.setores.find().pretty()

db.colaboradores.find().pretty()
```

Fonte: Elaborada pelo autor.

As relações entre dados no MongoDB podem ser estruturadas de maneira encaixada (*embedded*) ou por referência (*reference*). A abordagem de relacionamento encaixada mantém todos os dados da relação em um único documento. Isso possibilita que os dados possam ser retornados em apenas uma consulta e sem o custo de algo como *joins* entre diferentes tabelas de um banco de dados relacional. Como será apresentado nos próximos exemplos, para criar as relações entre os dados, é conveniente que os documentos sejam estruturados conforme serão consultados posteriormente.

A Figura 3.8 apresenta um relacionamento encaixado entre colaboradores e setores que permite a consulta dos colaboradores juntamente com seus setores de atuação. Essa figura mostra a inserção de um *array* com dois documentos. Cada um desses documentos representa um colaborador que possui os dados do setor onde ele atua encaixados no mesmo documento. A figura também apresenta o uso do findOne() para encontrar e consultar todos os dados, da mesma forma que foram inseridos, do documento que representa o primeiro colaborador com nome "Juçara".

Figura 3.8: Relacionamento encaixado entre dados no MongoDB 1

```
db.colaboradoresSetoresEncaixados.insert([{
    "_id": ObjectId("5dcabc2007c9ee115fea8c04"),
    "nome": "Daniel",
    "detalhes": "Possui doutorado em Ciência da Computação...",
    "setor": {
        "nome": "Docência em Informática",
        "detalhes": "Ensinar informática no ensino técnico e superior..."
    }
}, {
    "_id": ObjectId("5dcabc2007c9ee115fea8c05"),
    "nome": "Juçara",
    "detalhes": "Possui especialização em Sistemas de Computação...",
    "setor": {
        "nome": "Docência em Informática",
        "detalhes": "Ensinar informática no ensino técnico e superior..."
    }
}])

db.colaboradoresSetoresEncaixados.findOne({
    "nome": "Juçara"
})
```

Fonte: Elaborada pelo autor.

A Figura 3.9 apresenta um relacionamento encaixado entre setores e colaboradores que, ao contrário da figura anterior, permite a consulta dos setores juntamente com seus colaboradores. Essa nova figura mostra a inserção de um documento que representa um setor e que possui os dados dos seus colaboradores encaixados no mesmo documento. Essa figura também apresenta o uso do findOne() para consultar todos os dados, da mesma forma que foram inseridos, do documento que representa o primeiro setor com nome "Docência em Informática". A consulta também retorna os dados dos colaboradores do setor consultado.

Figura 3.9: Relacionamento encaixado entre dados no MongoDB 2

```
db.setoresColaboradoresEncaixados.insert({
    "_id": ObjectId("5dcabc2007c9ee115fea8c06"),
    "nome": "Docência em Informática",
    "detalhes": "Ensinar informática no ensino técnico e superior...",
    "colaboradores": [{
        "nome": "Daniel",
        "detalhes": "Possui doutorado em Ciência da Computação..."
    }, {
        "nome": "Juçara",
        "detalhes": "Possui especialização em Sistemas de Computação..."
    }]
})

db.setoresColaboradoresEncaixados.findOne({
    "nome": "Docência em Informática"
})
```

Fonte: Elaborada pelo autor.

Entretanto, apesar da facilidade de obter os dados com apenas uma consulta, os relacionamentos encaixados tendem a gerar demasiada redundância (não normalização), ou seja repetição dos dados em diferentes documentos que, além de utilizar mais espaço de armazenamento, torna a escrita mais custosa do que a leitura de dados e precisa ser controlada pelo código da aplicação.

A abordagem de relacionamentos referenciados, por outro lado, objetiva a redução da redundância (normalização). Os dados são gravados em documentos separados e suas relações são referenciadas pelos identificadores dos documentos como é exemplificado nas Figuras 3.10 e 3.12.

A Figura 3.10 apresenta a inserção dos dados de dois colaboradores onde cada colaborador possui uma referência para um setor onde atua. O setor referenciado nesses documentos ("5dcabc2007c9ee115fea8c01") foi inserido anteriormente na Figura 3.7. Neste caso, novamente há a possibilidade de consultar colaboradores juntamente com seus setores de atuação.

Figura 3.10: Relacionamento referenciado entre documentos do MongoDB 1

```
db.colaboradoresSetoresReferenciados.insert([{
    "_id": ObjectId("5dcabc2007c9ee115fea8c07"),
    "nome": "Daniel",
    "detalhes": "Possui doutorado em Ciência da Computação...",
    "idSetor": ObjectId("5dcabc2007c9ee115fea8c01")
}, {
    "_id": ObjectId("5dcabc2007c9ee115fea8c08"),
    "nome": "Juçara",
    "detalhes": "Possui especialização em Sistemas de Computação...",
    "idSetor": ObjectId("5dcabc2007c9ee115fea8c01")
}])
```

Fonte: Elaborada pelo autor.

A Figura 3.11 Demonstra como consultar dados no relacionamento referenciado apresentado anteriormente. Nessa figura, existem 3 comandos:

- O primeiro comando apresenta, através do findOne(), como consultar os dados do primeiro colaborador com nome "Daniel";
- O segundo comando objetiva armazenar os dados do primeiro colaborador com nome "Daniel" em uma variável de nome "tmp";
- O terceiro comando, usando o identificador de setor obtido no segundo comando (tmp.idSetor), consulta, via findOne() os dados do setor onde o primeiro colaborador de nome "Daniel" atua.

Figura 3.11: Consulta de dados em um relacionamento referenciado no MongoDB 1

```
db.colaboradoresSetoresReferenciados.findOne({
    "nome" : "Daniel"
})

var tmp = db.colaboradoresSetoresReferenciados.findOne({
    "nome": "Daniel"
})

db.setores.findOne({
    "_id": tmp.idSetor
})
```

Fonte: Elaborada pelo autor.

A Figura 3.12 apresenta a inserção dos dados de um setor que possui referências de colaboradores que atuam nesse setor armazenados em um *array*. Os colaboradores referenciados nesse documentos ("5dcabc2007c9ee115fea8c02" e "5dcabc2007c9ee115fea8c03") foram inseridos anteriormente na Figura 3.7. Neste caso, novamente há a possibilidade de consultar setores juntamente com seus colaboradores.

Figura 3.12: Relacionamento referenciado entre documentos do MongoDB 2

```
db.setoresColaboradoresReferenciados.insert({
    "_id": ObjectId("5dcabc2007c9ee115fea8c09"),
    "nome": "Docência em Informática",
    "detalhes": "Ensinar informática no ensino técnico e superior...",
    "idsColaboradores": [
        ObjectId("5dcabc2007c9ee115fea8c02"),
        ObjectId("5dcabc2007c9ee115fea8c03")
    ]
})
```

Fonte: Elaborada pelo autor.

A Figura 3.13 Demonstra como obter dados do último relacionamento referenciado apresentado. Nessa figura, existem 3 comandos:

- O primeiro comando apresenta, através do findOne(), como consultar os dados do primeiro setor com nome "Docência em Informática";
- O segundo comando objetiva armazenar as referências de colaboradores presentes no primeiro setor com nome "Docência em Informática" em um *array* de nome tmp;
- O terceiro comando, usando o *array* de identificadores de colaboradores obtido no segundo comando (tmp), consulta, via find() os dados de todos os colaboradores do primeiro setor com nome "Docência em Informática".

Figura 3.13: Consulta de dados em um relacionamento referenciado no MongoDB 2

```
db.setoresColaboradoresReferenciados.findOne({
    "nome" : "Docência em Informática"
})

var tmp = db.setoresColaboradoresReferenciados.findOne({
    "nome": "Docência em Informática"
}, {
    "idsColaboradores": 1
})

db.colaboradores.find({
    "_id": {
        "$in": tmp["idsColaboradores"]
    }
}).pretty()
```

Fonte: Elaborada pelo autor.

Apesar da redução da redundância de dados, as consultas em relacionamentos por referência tendem a serem mais complexas como foi apresentado nas Figuras 3.11 e 3.13. Também é possível haver quebra da integridade referencial (normalmente presente em bancos de dados relacionais), uma vez que é possível referenciar identificadores de documentos que ainda não foram criados.

Os experimentos foram criados considerando relações 1:N entre colaboradores e setores conforme mostrado no modelo conceitual da Figura 3.6. Supondo que a relação entre colaboradores e setores seja N:N, onde um setor pode ter vários colaboradores e um colaborador pode atuar em vários setores e que seja necessário obter todos os dados das entidades envolvidas como mostrado anteriormente nas consultas encaixadas e referenciadas no MongoDB deste trabalho, seria necessário também armazenar um *array* de setores em cada colaborador.

A dificuldade de controle desse tipo de situação aumenta ainda mais quando existem atributos de relação como a data de admissão de um colaborador em um determinado setor. Essas situações exemplificadas procuram tornar claro que o MongoDB e outros bancos orientados a documentos não são atraentes para manter relacionamentos complexos entre dados.

3.2.1.1 Modelo em Documentos (MongoDB) versus Modelo Relacional

As novas versões do mongoDB também suportam transações ACID e, além disso, os bancos de dados do tipo Documentos como o MongoDB, quando comparados com os sistemas do Modelo Relacional, apresentam vantagens como:

- Possibilitam a criação de estruturas de dados mais flexíveis, que facilitam a escalabilidade horizontal e que podem ser moldadas de acordo com as necessidades de cada aplicação;
- Permitem a criação de entidades de dados heterogêneas, facilitando mudanças na estrutura de determinadas entidades de dados sem a necessidade de afetar as demais entidades;
- Proporcionam a redução do custo de acesso aos dados através da desnormalização em relacionamentos encaixados.

Entretanto, apesar das limitações do modelo relacional em relação a escalabilidade, os bancos de dados relacionais são, de forma geral, mais convenientes para manter relacionamentos entre dados quando comparados com os bancos de dados NoSQL do tipo Documentos, tendo em vista as facilidades e abstrações proporcionadas pela linguagem de consulta SQL. De certa forma, os bancos de dados relacionais também podem armazenar entidades de dados heterogêneas ao usar campos para dados em notações como XML ou JSON. A desnormalização de dados também pode ser aplicada em bancos de dados relacionais com o intuito de reduzir os custos de consultas conforme as necessidades de cada aplicação.

3.3 MODELO EM GRAFOS

O modelo em grafo, como o nome indica, representa os dados em forma de grafos. Segundo Lóscio, Oliveira e Pontes (2011), o modelo orientado a grafos é composto basicamente de três componentes:

- Os nós, que são representados como os vértices do grafo;
- Os relacionamentos, que são representados pelas arestas do grafo;
- As propriedades ou atributos dos relacionamentos, que são representados pelas informações contidas nas arestas do grafo.

Consultas de dados em situações como "obter o chefe do chefe de um funcionário", "encontrar amigos de amigos em uma rede social" ou "recomendar músicas populares entre amigos" tendem a serem muito complexas no modelo relacional por envolverem muitas junções em tabelas. São nessas situações ou semelhantes, geralmente relacionadas com sistemas de recomendação, que os bancos de dados baseados em grafos ganham destaque, uma vez que os relacionamentos entre dados usando grafos tornam esse tipo de consulta mais simples e rápido.

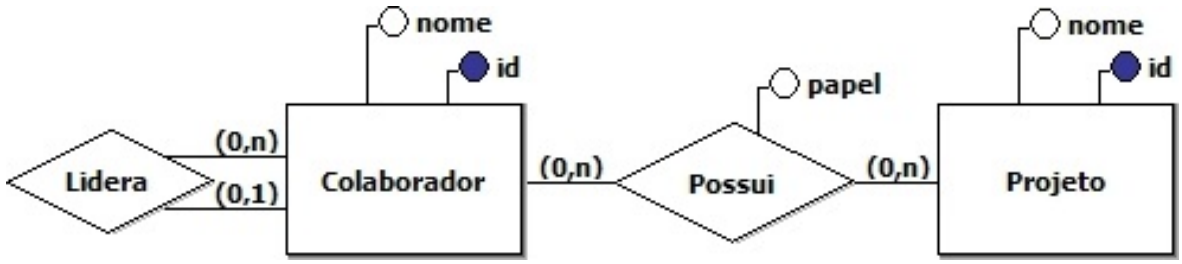
Exemplos de bancos de dados baseados em grafos: InfiniteGraph[9], AllegroGraph[10] e Neo4j[11]. O Neo4j, por ser muito popular nesse tipo de banco de dados, será abordado com mais detalhes a seguir.

3.3.1 Banco de dados Neo4j

O Neo4j é um banco de dados com mecanismo nativo de armazenamento e processamento de grafos, altamente escalável, intuitivo, flexível, seguro e criado especificamente para trabalhar não apenas com dados, como também com relacionamentos entre dados (NEO4J, 2019). O Neo4j, conforme será apresentado nos próximos experimentos, é bastante conveniente para manter relacionamentos, complexos ou não, entre entidades de dados.

A Figura 3.14 apresenta o modelo conceitual que será utilizado para fazer alguns experimentos com a linguagem de consulta Cypher (*Cypher Query Language*) do Neo4j. Essa figura, entre outros detalhes, apresenta um autorrelacionamento (Lidera). Esse tipo de relacionamento é pouco intuitivo de ser implementado em um banco de dados relacional e pode ser encontrado em situações onde uma entidade pode se relacionar com outras entidades de mesmo tipo como seguidores e amigos em redes sociais.

Figura 3.14: Modelo conceitual para experimentos com Neo4j



Fonte: Elaborada pelo autor.

[9] InfiniteGraph - https://www.objectivity.com/products/infinitegraph/. Acessado em 17 de outubro de 2019
[10] AllegroGraph - https://allegrograph.com/. Acessado em 29 de outubro de 2019
[11] Neo4j - https://neo4j.com/. Acessado em 26 de outubro de 2019

A Figura 3.15 apresenta como fazer a inserção de dados no Neo4j. Nesse banco de dados, as entidades são representadas pelos nós do grafo. Nesse caso, são inseridos três nós do tipo Colaborador usando o comando CREATE: um nó para "Daniel", um nó para "Moacir" e um nó para "Valmir".

Figura 3.15: Inserindo dados no Neo4j

```
CREATE (cA:Colaborador {id: 1, nome: "Daniel"})
CREATE (cB:Colaborador {id: 2, nome: "Moacir"})
CREATE (cC:Colaborador {id: 3, nome: "Valmir"})
```

Fonte: Elaborada pelo autor.

A Figura 3.16 apresenta os resultados da execução do comando "MATCH (n:Colaborador) RETURN n", que exibe todos os nós do tipo Colaborador (3 itens).

Figura 3.16: Consultando dados no Neo4j

Fonte: Elaborada pelo autor.

As relações entre dados no Neo4j são definidas através da criação de arestas direcionadas entre os nós. A Figura 3.17 apresenta como criar relações entre os dados. Nessa figura, estão dispostos 7 comandos.

- Os três primeiros comandos consultam os três nós já adicionados (Figura 3.15);
- O quarto comando cria mais um nó do tipo Colaborador, agora com nome "Juçara";
- Os comandos restantes criam relações de liderança entre os nós criados anteriormente. O nó cA ("Valmir") passa a liderar o nó cB ("Moacir"), enquanto o nó cB ("Moacir") passa a liderar os nós cC e cD ("Daniel" e "Juçara").

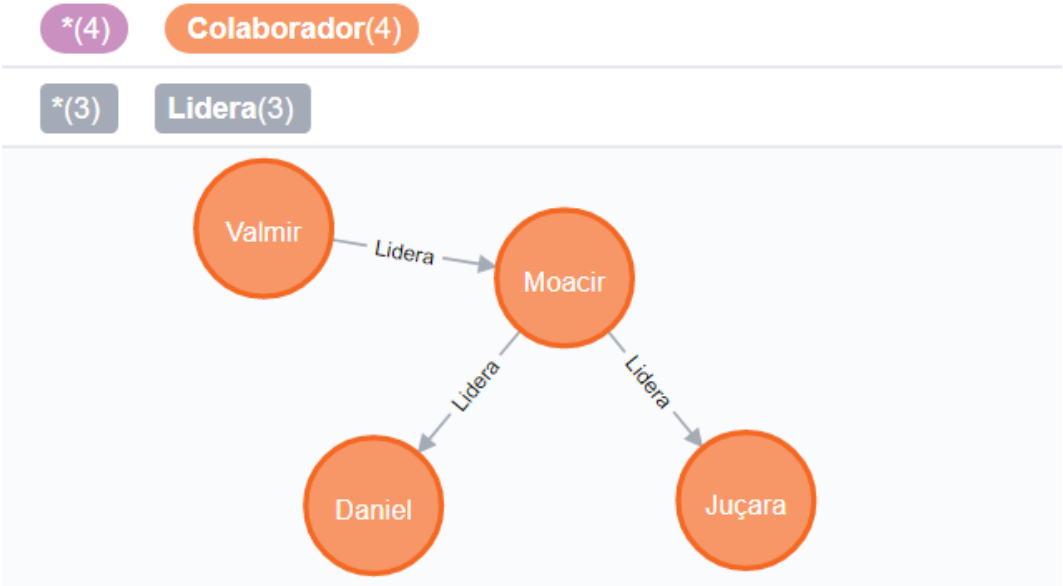
Figura 3.17: Criando relações entre dados no Neo4j

```
MATCH (cA:Colaborador) WHERE cA.nome = "Valmir"
MATCH (cB:Colaborador) WHERE cB.nome = "Moacir"
MATCH (cC:Colaborador) WHERE cC.nome = "Daniel"

CREATE (cD:Colaborador {id: 4, nome: "Juçara"})

CREATE (cA)-[:Lidera]->(cB)
CREATE (cB)-[:Lidera]->(cC)
CREATE (cB)-[:Lidera]->(cD)
```

Fonte: Elaborada pelo autor.

A Figura 3.18 apresenta o resultado da execução do comando "MATCH n=()-[r:Lidera]->() RETURN n", que exibe todos os nós envolvidos nas relações do tipo "Lidera".

Figura 3.18: Exibindo relações entre dados no Neo4j



Fonte: Elaborada pelo autor.

A Figura 3.19 Apresenta a criação de três projetos e associa alguns colaboradores com os projetos criados através dos últimos três 4 comandos:

- O colaborador "Daniel" passa a atuar como orientador nos projetos "TCC 1 do Cezar" e "TCC 2 do Cezar".

- Os colaboradores "Daniel" e "Juçara" passam a atuar no projeto "CSI - Coordenação" ("Daniel" no papel de "Coordenador substituto" e "Juçara" no papel de "Coordenadora").

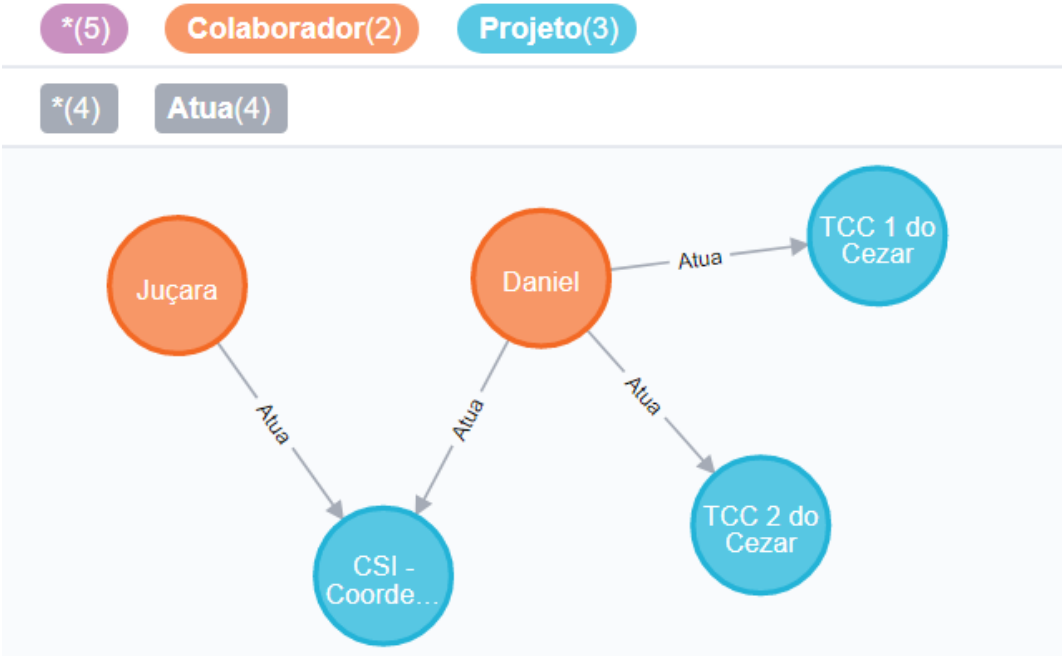
Figura 3.19: Criando relações com atributos no Neo4j

```
MATCH (cC:Colaborador) WHERE cC.nome = "Daniel"
MATCH (cD:Colaborador) WHERE cD.nome = "Juçara"

CREATE (pA:Projeto {id: 1, nome: "TCC 1 do Cezar"})
CREATE (pB:Projeto {id: 2, nome: "TCC 2 do Cezar"})
CREATE (pC:Projeto {id: 3, nome: "CSI - Coordenação"})

CREATE (cC)-[:Atua {papel: "Orientador"}]->(pA)
CREATE (cC)-[:Atua {papel: "Orientador"}]->(pB)
CREATE (cC)-[:Atua {papel: "Coordenador substituto"}]->(pC)
CREATE (cD)-[:Atua {papel: "Coordenadora"}]->(pC)
```

Fonte: Elaborada pelo autor.

A Figura 3.20 apresenta o resultado da execução do comando "MATCH n=()-[r:Atua]->() RETURN n", que exibe todos os nós envolvidos em relações do tipo "Atua".

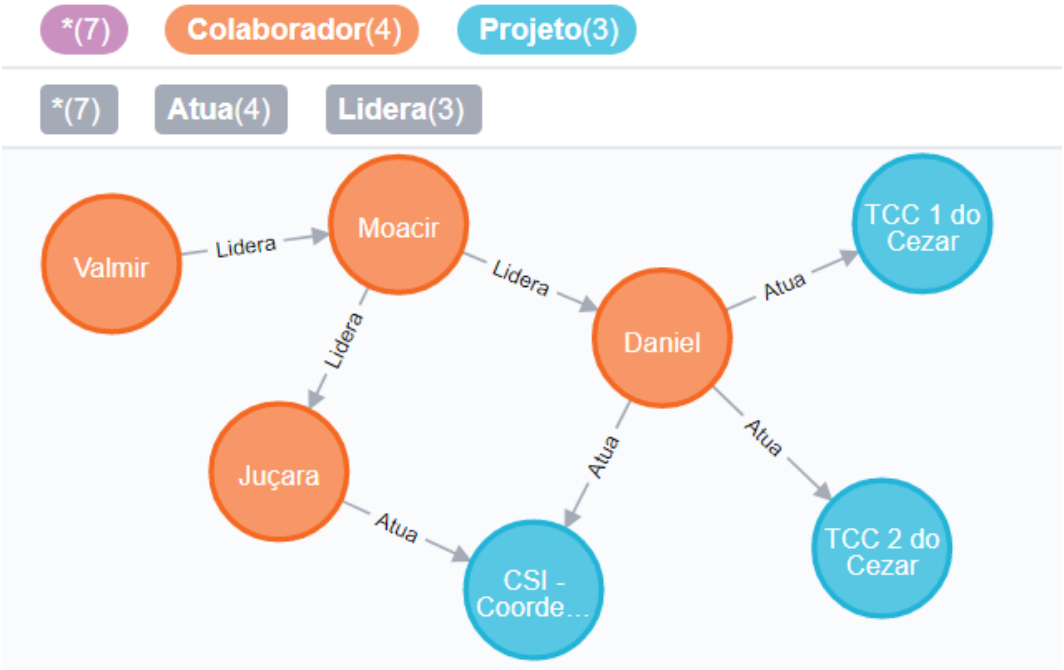
Figura 3.20: Exibindo relações entre dados no Neo4j



Fonte: Elaborada pelo autor.

A Figura 3.21 apresenta o resultado da execução do comando "MATCH n=()–>() RETURN n", que exibe todos os nós envolvidos em algum tipo de relação. A mesma consulta também pode ser visualizada de forma textual na Figura 3.22.

Figura 3.21: Exibindo múltiplas relações entre dados no Neo4j



Fonte: Elaborada pelo autor.

Figura 3.22: Exibindo atributos de múltiplas relações entre dados no Neo4j


```
[{"nome":"Moacir","id":2},{},{"nome":"Daniel","id":1}]
[{"nome":"Valmir","id":3},{},{"nome":"Moacir","id":2}]
[{"nome":"Moacir","id":2},{},{"nome":"Juçara","id":4}]
[{"nome":"Daniel","id":1},{"papel":"Orientador"},{"nome":"TCC 1 do Cezar","id":1}]
[{"nome":"Daniel","id":1},{"papel":"Orientador"},{"nome":"TCC 2 do Cezar","id":2}]
[{"nome":"Juçara","id":4},{"papel":"Coordenadora"},{"nome":"CSI - Coordenação","id
":3}]
[{"nome":"Daniel","id":1},{"papel":"Coordenador substituto"},{"nome":"CSI - Coorde
nação","id":3}]
```


Fonte: Elaborada pelo autor.

A figura anterior (Figura 3.22), também resultante do comando "MATCH n=()–>() RETURN n", explicita os atributos de relações entre os nós. As relações do tipo "Lidera" não possuem atributos, mas as relações de tipo "Atua" possuem o atributo "papel".

Por fim, a Figura 3.23 exemplifica outros dois comandos. O primeiro comando remove todas as relações do tipo "Lidera". Já o segundo comando, remove todas as relações e todos os nós do grafo. O Neo4j implementa as propriedades ACID e, dessa forma, não é possível remover um nó se este ainda estiver envolvido em um relacionamento.

Figura 3.23: Removendo dados e relações entre dados no Neo4j

```
MATCH n=()-[r:Lidera]->() DELETE r

MATCH r=()-->() DELETE r
```

Fonte: Elaborada pelo autor.

3.3.1.1 Modelo em Grafos (Neo4j) versus Modelo Relacional

Os bancos de dados baseados em grafos como o Neo4j, quando comparados com o Modelo Relacional, apresentam vantagens como:

- Tornam simples a visualização dos dados e como eles se relacionam;
- São convenientes para autorrelacionamento, ou seja, são adequados para manter relacionamentos entre entidades de dados do mesmo tipo. Autorrelacionamentos são comuns em sistemas de recomendação;
- As consultas aos dados são intuitivas e executadas rapidamente, mesmo quando esses estão envolvidos em relacionamentos complexos;
- Não é necessário criar um elemento extra para manter relações do tipo N:N (muitos para muitos) como ocorre nos bancos de dados relacionais (tabela adicional para o N:N).

Entretanto, os bancos de dados relacionais ainda são mais convenientes para armazenar dados naturalmente tabulares como listas telefônicas ou folhas de pagamento. Além disso, é possível usar a desnormalização nos bancos de dados relacionais para reduzir a necessidade de relacionamentos entre dados e, consequentemente, diminuir a necessidade do uso de *joins* entre tabelas, mesmo que isso gere mais custos em termos de escrita e armazenamento por causa da redundância provocada pela desnormalização dos dados.

3.4 MODELO EM COLUNAS

Esse modelo é o que mais se parece com o relacional, tendo em vista que também é estruturado por meio de linhas e colunas (CLAUDINO; SOUZA; SALGADO, 2015). Esse modelo de banco de dados usa o conceito de família de colunas, ou seja, procura agrupar colunas que armazenam o mesmo campo de dados (LÓSCIO; OLIVEIRA; PONTES, 2011). Isso possibilita que consultas a subconjuntos de dados sejam mais eficientes, mas torna a obtenção de entidades inteiras mais custosa (CLAUDINO; SOUZA; SALGADO, 2015).

O agrupamento por colunas permite, por exemplo, ordenações, filtragens e contabilizações mais rápidas de dados, pois proporciona que somente os campos desejados de cada entidade sejam envolvidos na operação ao invés de envolver as entidades inteiras como ocorre no modelo relacional.

Como exemplos de bancos de dados desse tipo, pode-se citar o HBase[12], Hypertable[13] e o Cassandra[14]. O Cassandra será melhor detalhado a seguir.

3.4.1 Banco de dados Cassandra

O banco de dados Cassandra é distribuído, descentralizado, tolerante a falhas e objetiva manter altos níveis de desempenho, escalabilidade e disponibilidade (CASSANDRA, 2019). Foi pensado para armazenar grandes volumes de dados, que podem ser redistribuídos de acordo com a alocação ou desalocação de *datacenters* em escala de desempenho linear usando arquitetura *peer-to-peer* (ABRAMOVA; BERNARDINO, 2013).

No Cassandra, devido a arquitetura que prioriza a disponibilidade e escalabilidade, não existe o conceito de relacionamentos entre entidades. A gravação dos dados pode ser organizada de forma que esses possam ser consultados com o menor custo possível para atender o máximo de requisições provenientes de milhares, milhões ou até bilhões de usuários de uma aplicação como ocorre no Facebook, mesmo que isso gere demasiada redundância. O Cassandra utiliza a linguagem CQL (*Cassandra Query Language*) que é bastante similar a linguagem SQL (*Structured Query Language*) dos bancos de dados relacionais.

A Figura 3.24 apresenta o modelo conceitual que será utilizado para fazer alguns experimentos com o Cassandra.

[12] https://hbase.apache.org/
[13] https://hypertable.org/
[14] http://cassandra.apache.org/

Figura 3.24: Modelo conceitual para experimentos com Cassandra

Fonte: Elaborada pelo autor.

A Figura 3.25 apresenta a definição de uma estrutura de dados básica no Cassandra:

- O primeiro comando apresentado nessa figura lista todos os *keyspaces* já definidos;
- O segundo comando cria um novo *keyspace* de nome "cassandra_tcc1_cezar";
- O terceiro comando seleciona o *keyspace* criado anteriormente para uso;
- O quarto comando cria uma estrutura de dados simples para armazenar os alunos;
- O quinto e último comando dessa figura exibe os detalhes (no caso, a tabela criada, as chaves e tipagens dos dados suportados por cada chave) do *keyspace* "cassandra_tcc1_cezar".

Figura 3.25: Definindo uma estrutura básica de dados no Cassandra

```
DESC KEYSPACES;

CREATE KEYSPACE cassandra_tcc1_cezar WITH replication = {
    'class': 'SimpleStrategy',
    'replication_factor': 1
} AND durable_writes = true;

USE cassandra_tcc1_cezar;

CREATE TABLE aluno (
    id text,
    nome text,
    curso text,
    PRIMARY KEY(id)
);

DESC KEYSPACE cassandra_tcc1_cezar;
```

Fonte: Elaborada pelo autor.

Os *keyspaces* do Cassandra equivalem aos *databases* do modelo relacional. O *SimpleStrategy* é usado quando se deseja manter o banco de dados sobre apenas um sistema computacional, enquanto o *NetworkTopologyStrategy* é recomendado para aplicações que podem ser expandidas para vários *datacenters* quando isso for necessário (DATASTAX, 2019).

A Figura 3.26 apresenta a inserção de três alunos no banco de dados Cassandra.

Figura 3.26: Inserindo dados no Cassandra

```
INSERT INTO aluno(id, nome, curso)
    VALUES ('0001', 'Enrico', 'Sistemas para Internet');

INSERT INTO aluno(id, nome, curso)
    VALUES ('0002', 'Marcos', 'Eng Controle e Automação');

INSERT INTO aluno(id, nome, curso)
    VALUES('0003', 'Lucas', 'Redes de Computadores');
```

Fonte: Elaborada pelo autor.

A Figura 3.27 mostra como os dados podem ser consultados e alterados no Cassandra:

- O primeiro comando, da mesma forma que os bancos de dados relacionais normalmente fazem, consulta todos os alunos e os exibem em formato de tabela;
- O segundo comando, por outro lado, exibe apenas os nomes dos alunos;
- O terceiro comando altera o curso do aluno com o id "0002";
- O último comando dessa figura consulta somente o nome do aluno com id "0002".

Figura 3.27: Consultando e manipulando dados no Cassandra

```
SELECT * FROM aluno;

SELECT nome FROM aluno;

UPDATE aluno SET curso = 'Engenharia de Controle e Automação'
    WHERE id = '0002';

SELECT nome FROM aluno WHERE id = '0002';
```

Fonte: Elaborada pelo autor.

Por fim, para encerrar os testes com o Cassandra, a Figura 3.28 apresenta como remover dados do banco de dados Cassandra:

- O primeiro comando dessa figura, remove o campo curso do aluno com id "0003" (o campo fica marcado como nulo e é desconsiderado nas próximas operações de consulta);
- Já o último comando da figura, remove o aluno inteiro com id "0003".

Figura 3.28: Removendo dados do Cassandra

```
DELETE curso FROM aluno WHERE id = '0003';

DELETE FROM aluno WHERE id = '0003';
```

Fonte: Elaborada pelo autor.

3.4.1.1 Modelo em Colunas (Cassandra) versus Modelo Relacional

Os bancos de dados do Modelo em Colunas como o Cassandra, diferentemente dos bancos de dados relacionais, deixam o conceito de relacionamento de lado para maximizar o atendimento de altas demandas de dados como ocorre em grandes redes sociais mesmo que isso gere demasiada redundância e maior complexidade de implementação. Entretanto, se não houver uma grande demanda de dados, os bancos de dados relacionais ainda podem ser a melhor escolha, tendo em vista a maior simplicidade de implementação e operação.

3.5 CRITÉRIOS PARA A ESCOLHA DE BANCOS DE DADOS

O capítulo anterior procurou apresentar o funcionamento básico dos diferentes tipos de bancos de dados NoSQL e como eles podem ser comparados com os bancos de dados relacionais. Para a escolha de um banco de dados, vários critérios podem ser levados em consideração, desde a quantidade de usuários que irão utilizar a aplicação, até a natureza das atividades que serão desempenhadas nessa aplicação.

3.5.1 Escolha baseada na demanda de dados

De forma geral, um banco de dados relacional é uma boa escolha para a maioria dos casos onde não há uma grande demanda de dados. Os bancos de dados relacionais, conforme já

abordado neste trabalho, oferecem muitos recursos e abstrações que ajudam a manter a simplicidade de uso e a consistência dos dados. Caso a aplicação tenha necessidades específicas, os próximos tópicos podem ajudar a escolher outro tipo de banco de dados.

3.5.2 Escolha baseada nas características dos bancos de dados

A Tabela 3.2 objetiva resumir as principais diferenças entre os diferentes tipos de bancos de dados abordados no capítulo anterior com o intuito de ajudar a escolher um banco de dados.

Tabela 3.2: Comparação geral entre os tipos de bancos de dados

	Relacional	Chave-valor (Redis)	Documentos (MongoDB)	Grafos (Neo4j)	Colunas (Cassandra)
Relacionamentos entre entidades de dados	Sim	Não	Limitado *	Avançado	Não
Entidades heterogêneas	Não *	Não	Sim	Sim	Não
Escalonamento horizontal	Não	Sim	Sim	Sim	Sim
Transações ACID	Sim	Não	Sim	Sim	Não
Normalização de dados	Sim	Não	Sim *	Sim	Não
Flexibilidade para mudanças na estrutura de dados	Baixa	Alta	Alta	Alta	Baixa

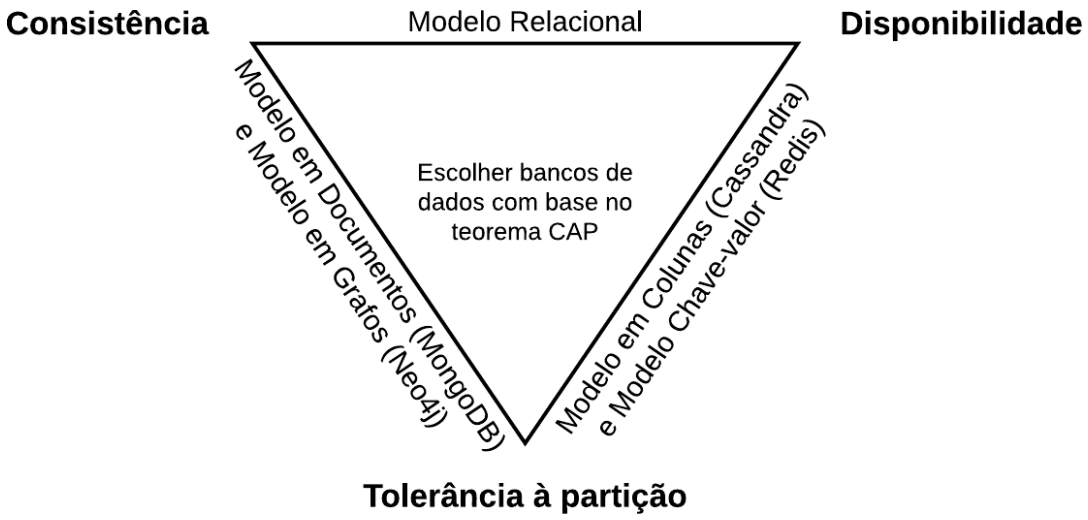
Na tabela anterior, alguns campos foram marcados com um * para que as seguintes observações pudessem ser passadas:

- Relacionamentos entre entidades de dados foram definidos como "Limitado *" no Modelo em Documentos (MongoDB), tendo em vista que esse banco de dados não conta com *joins* e os relacionamentos precisam ser tratados no código da aplicação;
- O uso de entidades heterogêneas foi marcado como "Não *" no Modelo Relacional, tendo em vista que é possível contornar isso através do uso de campos para dados em notação como XML ou JSON no Modelo Relacional;
- A normalização de dados foi marcada como "Sim *" no Modelo em Documentos (MongoDB), pois para isso, usa-se relacionamentos referenciados (*referenced*) que são mais complexos e também precisam ser tratados no código da aplicação.

3.5.3 Escolha de bancos de dados baseada no teorema CAP

Dependendo do tamanho da demanda que a aplicação deve suportar, pode-se considerar o uso ou não da escalabilidade horizontal. Com base no teorema CAP, explorado anteriormente neste trabalho, podemos escolher diferentes bancos de dados de acordo com as possíveis combinações entre consistência, disponibilidade e tolerância à partição. A Figura 3.29 apresenta os bancos de dados que possuem arquiteturas mais favoráveis para cada combinação.

Figura 3.29: Escolher bancos de dados com base no teorema CAP



Fonte: Elaborada pelo autor.

Bancos de dados relacionais são de difícil particionamento, mas mantêm a consistência e disponibilidade dos dados. Os bancos de dados Neo4j e MongoDB, quando particionados em um escalonamento horizontal, tendem a priorizar a consistência em detrimento da disponibilidade. Por fim, o banco de dados Cassandra e Redis tendem a priorizar a disponibilidade em detrimento da consistência quando particionado.

3.5.4 Escolha de bancos de dados baseada em necessidades temporais

Algumas aplicações podem precisar recolher, armazenar e processar dados em períodos regulares de tempo, mantendo a ordem de captura desses dados. Esse tipo de situação pode ocorrer, por exemplo, em sistemas de sensoriamento em IOT (*Internet of Things*). InfluxDB [15] é um exemplo de banco de dados utilizado para criar e operar aplicações de séries temporais.

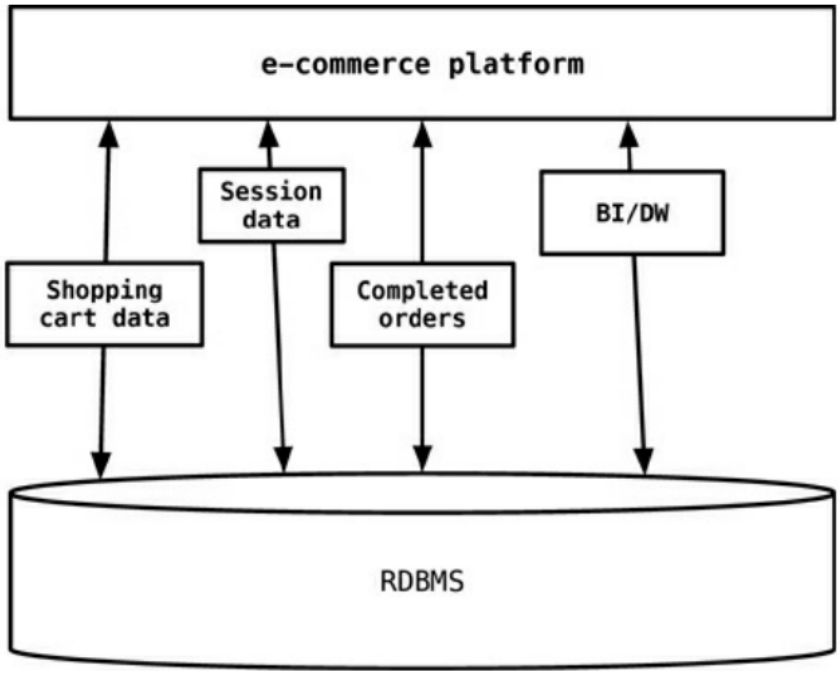
[15] https://www.influxdata.com/

Neste capítulo, foram abordados os diferentes tipos de bancos de dados juntamente com um exemplar de tecnologia para cada um desses tipos. Também foram apresentados alguns critérios para a escolha das tecnologias mais adequadas para cada situação. Os diferentes sistemas de dados foram criados para atender necessidades específicas e esses podem ser combinados, como é mostrado no próximo capítulo, para se obter as vantagens de cada um desses sistemas.

4 PERSISTÊNCIA POLIGLOTA DE DADOS

Muitas organizações inclinam-se a utilizar somente uma solução, frequentemente um banco de dados relacional, para armazenamento de sessões, transações, relatórios, BI / DW (*Business Intelligence* / *Data Warehouse*), e outras informações como pode ser observado na Figura 4.1 (SADALAGE; FOWLER, 2013).

Figura 4.1: Armazenamento de dados monoglótico



Fonte: (SADALAGE; FOWLER, 2013)

Entretanto, dependendo das necessidades do sistema, essa abordagem pode impactar negativamente o desempenho e a escalabilidade da aplicação, visto que nem todos os dados podem necessitar dos mesmos tratamentos de consistência e segurança.

Como foi tratado no capítulo anterior, existem diferentes bancos de dados para propósitos diferentes. As vantagens e conveniências de cada banco de dados podem ser combinadas na implementação de um sistema de persistência poliglota.

Por exemplo, um banco de dados do tipo Chave-valor pode ser usado para o armazenamento de sessões de usuário, informações associadas a carrinhos de compras não confirmadas em *e-commerces* e outros dados transitórios que não precisam ser persistidos diretamente em bancos de dados relacionais (SADALAGE; FOWLER, 2013).

4.1 USANDO PERSISTÊNCIA POLIGLOTA DE DADOS EM UMA APLICAÇÃO

Com o intuito de exemplificar a utilização dos diferentes tipos de bancos de dados em uma aplicação, este capítulo apresenta a criação de um sistema de software simples, que implementa persistência poliglota através do uso de três bancos de dados. Cada um desses bancos de dados será responsável por tratar os diferentes tipos de dados presentes no fluxo de compras de produtos em uma aplicação de e-commerce.

4.1.1 Descrição da aplicação

O sistema criado é uma aplicação cliente/servidor, que utiliza Node.Js na parte do *back end* para a criação de uma API, enquanto a parte do *front end* utiliza apenas HTML, CSS e JavaScript puros para a construção da interface de usuário (GUI). Esse sistema possui uma implementação simples, onde detalhes com a autenticação e segurança foram ignorados para que o foco seja direcionado para a persistência poliglota.

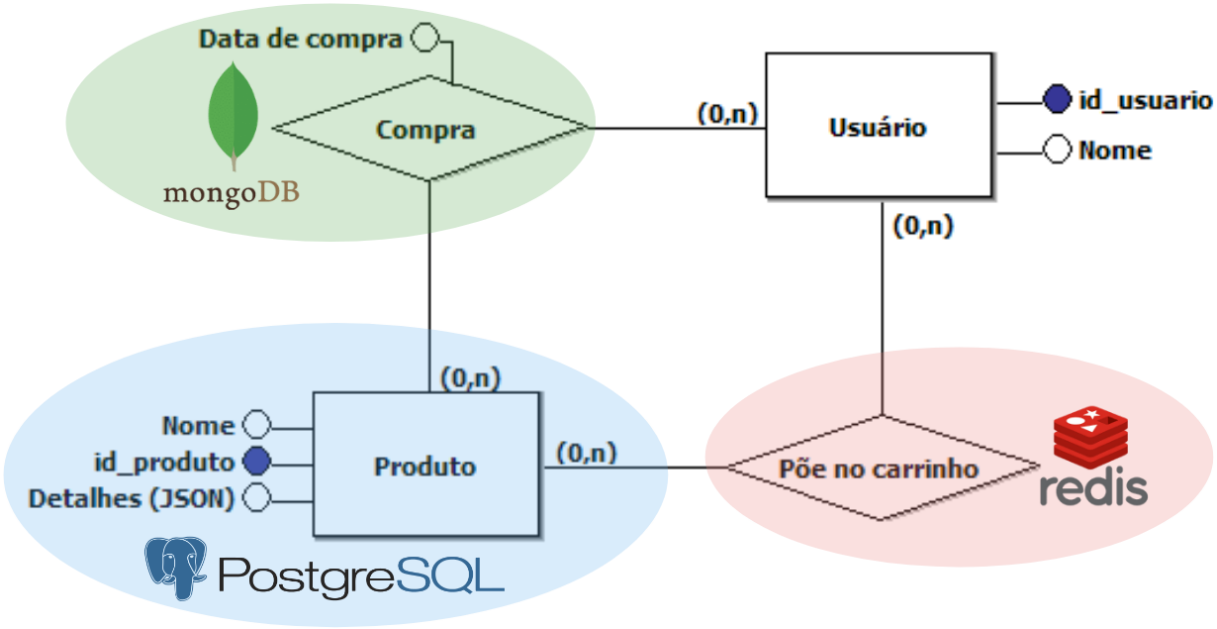
4.1.2 Modelo de dados da aplicação

Os dados da aplicação serão distribuídos nos bancos de dados PostgreSQL (Relacional), MongoDB (Documentos) e Redis (Chave-valor) conforme será explicado a seguir:

- O banco de dados PostgreSQL será responsável por armazenar os produtos do e-commerce, uma vez que ele é adequado para a criação de possíveis relacionamentos como a categorização dos produtos que podem ser usados para incrementar a aplicação futuramente;
- O MongoDB será usado para armazenar as compras já realizadas pelos clientes, tendo em vista que essas compras já realizadas são apenas armazenadas e acessadas (não são editadas ou removidas);
- O Redis será usado para armazenar o carrinho de compras, pois ele é conveniente para o armazenamento de dados transitórios. O Redis também poderia ser usado como cache no PostgreSQL, mas isso não será implementado por uma questão de simplicidade.

A Figura 4.2 apresenta o modelo conceitual usado como uma referência de organização de dados na aplicação.

Figura 4.2: Modelo conceitual de dados da aplicação



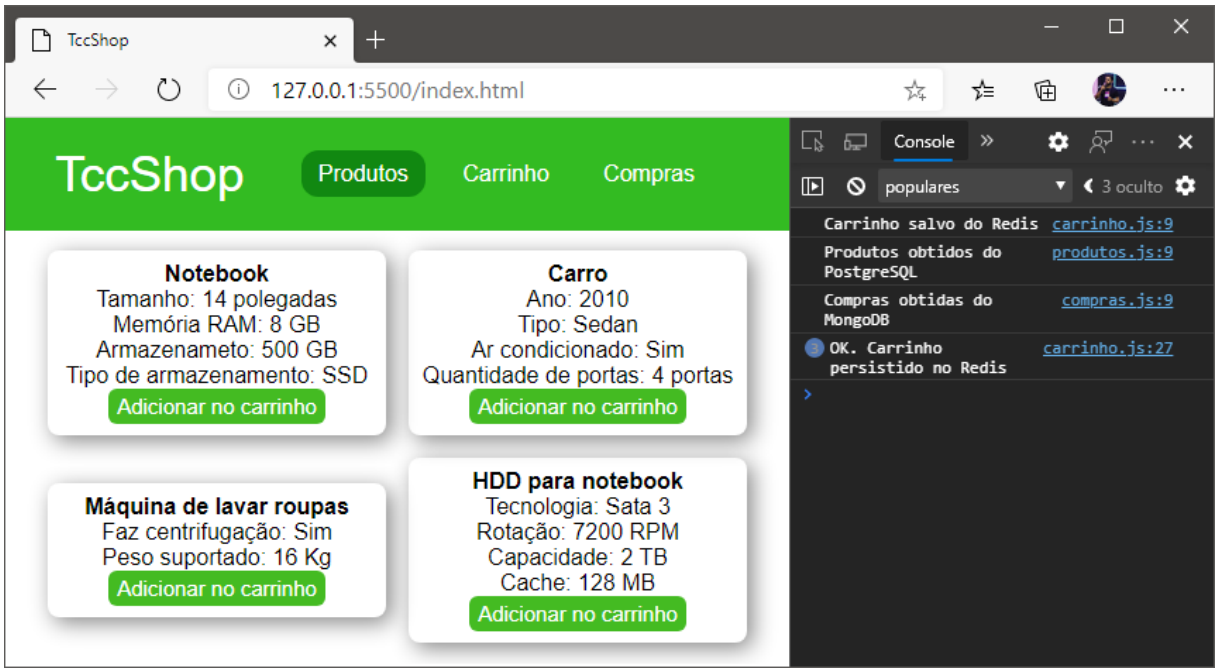
Fonte: Elaborada pelo autor.

Neste caso, distribuir os dados em diferentes bancos de dados ajuda a eliminar boa parte dos relacionamentos que seriam necessários caso a aplicação fosse construída usando apenas um banco de dados e, consequentemente, eliminar grande parte da necessidade do uso de *joins* mesmo que isso gere uma maior redundância de dados.

4.1.3 Descrição do funcionamento

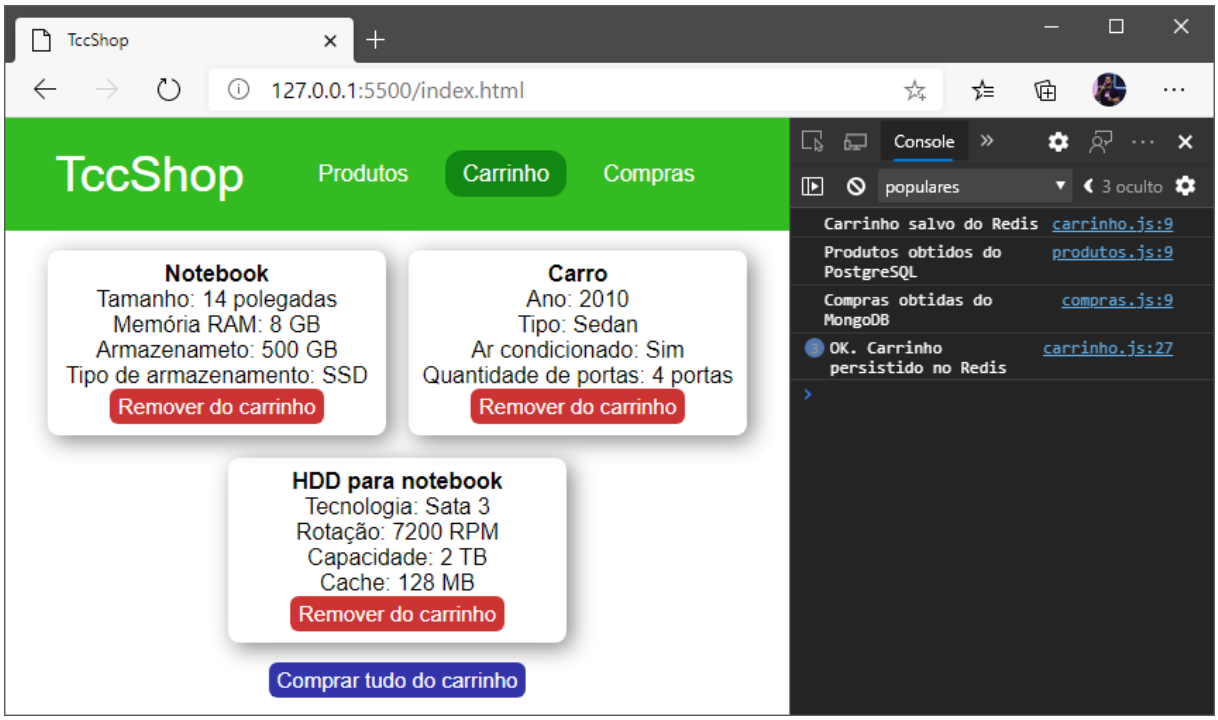
Nesta parte do trabalho, será descrito como os componentes do sistema foram estruturados e como eles interagem entre si. A aplicação inteira (*back end* + *front end*) possibilita que um usuário possa selecionar itens em uma lista de produtos (Figura 4.3) para pôr no carrinho de compras temporário desse usuário (Figura 4.4). Após colocar os produtos desejados no carrinho, o usuário poderá ajustar quais são os produtos que ele realmente deseja e finalizar a compra. Os produtos comprados ficam dispostos em uma listagem para consultas posteriores (Figura 4.5).

Figura 4.3: Seleção de produtos na aplicação



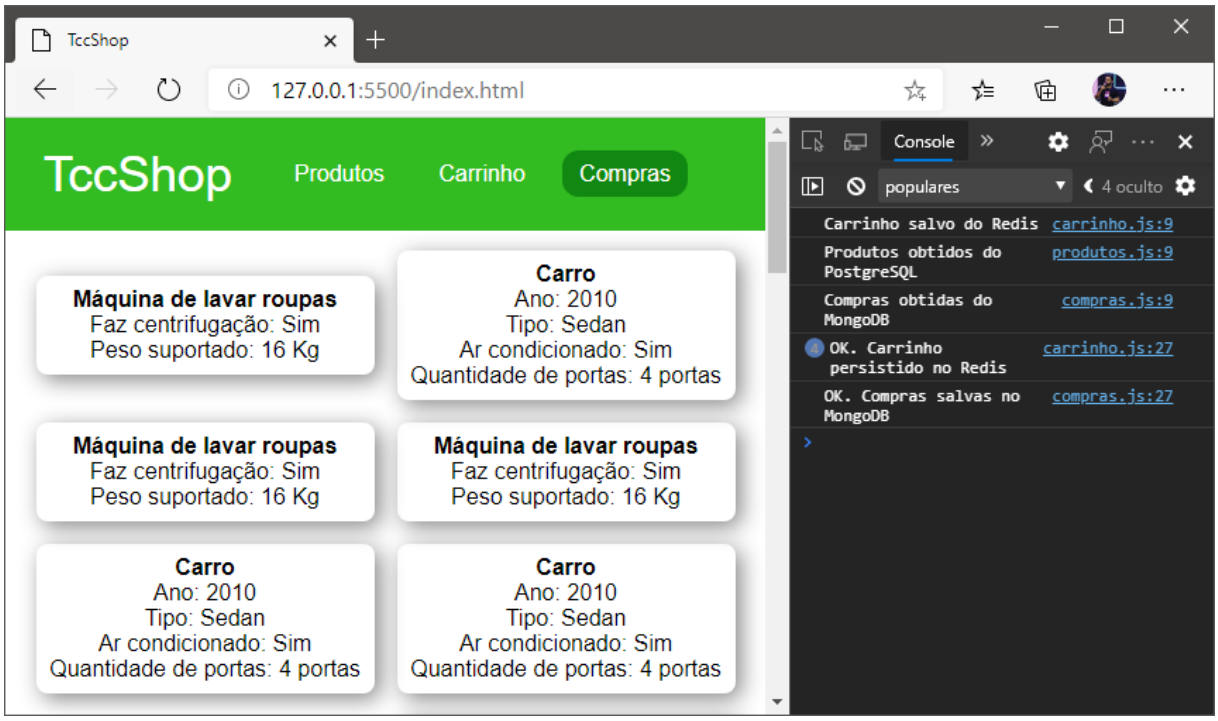
Fonte: Elaborada pelo autor.

Figura 4.4: Visualização e ajustes dos produtos no carrinho da aplicação



Fonte: Elaborada pelo autor.

Figura 4.5: Visualização de compras já efetuadas na aplicação

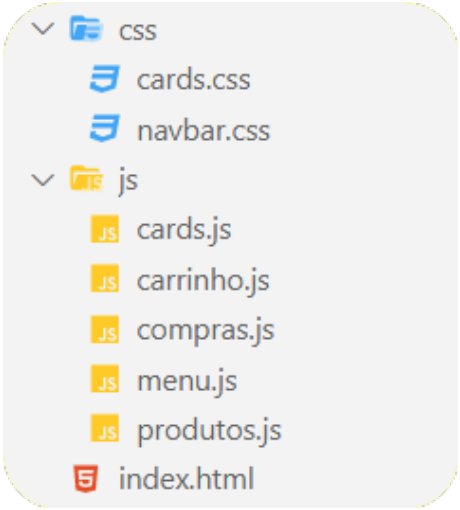


Fonte: Elaborada pelo autor.

4.1.3.1 Front end da aplicação (HTML, CSS e JavaScript puros)

A Figura 4.6 apresenta a hierarquia de pastas e arquivos do *front end* da aplicação.

Figura 4.6: Hierarquia de pastas e arquivos do *front end* da aplicação



Fonte: Elaborada pelo autor.

A Figura 4.7 mostra o conteúdo do arquivo index.html, que será manipulado dinamicamente via JavaScript para que a troca de conteúdo seja realizada sem o redirecionamento de páginas. Esse conteúdo é estilizado por dois arquivos de estilos como é mostrado na Figura 4.8.

Figura 4.7: Interface de usuário dinâmica no *front end* da aplicação

index.html > ...

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <title>TccShop</title>
  <link rel="stylesheet" href="css/cards.css">
  <link rel="stylesheet" href="css/navbar.css">
</head>

<body>
  <nav>
    <a class="logo" href="index.html">TccShop</a>
    <ul>
      <li><a id="menuProdutos" onclick="goToProdutos()">Produtos</a></li>
      <li><a id="menuCarrinho" onclick="goToCarrinho()">Carrinho</a></li>
      <li><a id="menuCompras" onclick="goToCompras()">Compras</a></li>
    </ul>
  </nav>

  <noscript>Este site precisa de JavaScript para funcionar!</noscript>

  <div id="divProdutos"></div>

  <button id="buy" onclick="comprarTudo()">Comprar tudo do carrinho</button>

  <script src="js/menu.js"></script>
  <script src="js/cards.js"></script>

  <script src="js/produtos.js"></script>
  <script src="js/carrinho.js"></script>
  <script src="js/compras.js"></script>

  <script>goToProdutos()</script>

  <script>getCarrinho(false)</script>
  <script>getCompras(false)</script>
</body>

</html>
```

Fonte: Elaborada pelo autor.

Figura 4.8: Folhas de estilho do *front end* da aplicação

css > cards.css > ...

```css
#divProdutos {
  padding: 6px;
  display: flex;
  flex-wrap: wrap;
  text-align: center;
  align-items: center;
  flex-direction: 'row';
  justify-content: center;
}

.card {
  margin: 8px;
  padding: 8px;
  width: 240px;
  transition: 0.1s;
  border-radius: 8px;
  box-shadow: 4px 4px 16px #888;
}

.card:hover {
  transform: scale(1.04);
}

button {
  font-size: 15px;
  padding: 4px 6px;
  color: #fff;
  border: none;
  border-radius: 6px;
}

button:hover {
  opacity: 0.75;
  cursor: pointer;
}

#buy {
  display: none;
  background: #33a;
}

.addToChart {
  background: #4b2;
}

.remove {
  background: #c33;
}
```

css > navbar.css > ...

```css
* {
  margin: 0;
  padding: 0;
  text-align: center;
  box-sizing: border-box;
  font-family: sans-serif;
}

nav {
  height: 80px;
  background: #3b2;
}

nav ul {
  float: right;
  margin-right: 50px;
}

nav ul li {
  margin: 0 5px;
  list-style: none;
  line-height: 80px;
  display: inline-block;
}

a {
  color: #fff;
  font-size: 16px;
  cursor: pointer;
  padding: 8px 12px;
  border-radius: 12px;
  text-decoration: none;
  transition: .5s;
}

a.active, a:hover {
  background: #181;
}

.logo {
  color: white;
  font-size: 35px;
  line-height: 80px;
}
```

Fonte: Elaborada pelo autor.

A Figura 4.9 exibe como ocorre as mudanças dinâmicas no conteúdo da aplicação, onde os elementos da página são capturados pelo id e alterados para se adequarem ao contexto atual da aplicação. O uso da variável "ultimaEscolha" serve como uma maneira simples de garantir que somente a última requisição assíncrona de mudança de estado na aplicação seja considerada.

Figura 4.9: Controle dinâmico da interface de usuário no *front end* da aplicação

js > menu.js > ...

```
1   const menuProdutos = document.getElementById('menuProdutos')
2   const menuCarrinho = document.getElementById('menuCarrinho')
3   const menuCompras = document.getElementById('menuCompras')
4   const divProdutos = document.getElementById('divProdutos')
5   var ultimaEscolha = ''
6
7   function goToProdutos() {
8     menuProdutos.setAttribute('class', 'active')
9     menuCarrinho.removeAttribute('class', 'active')
10    menuCompras.removeAttribute('class', 'active')
11
12    ultimaEscolha = 'produtos'
13    divProdutos.innerHTML = 'Carregando produtos...'
14    getProdutos(true)
15  }
16
17  function goToCarrinho() {
18    menuProdutos.removeAttribute('class', 'active')
19    menuCarrinho.setAttribute('class', 'active')
20    menuCompras.removeAttribute('class', 'active')
21
22    ultimaEscolha = 'carrinho'
23    divProdutos.innerHTML = 'Carregando carrinho...'
24    getCarrinho(true)
25  }
26
27  function goToCompras() {
28    menuProdutos.removeAttribute('class', 'active')
29    menuCarrinho.removeAttribute('class', 'active')
30    menuCompras.setAttribute('class', 'active')
31
32    ultimaEscolha = 'compras'
33    divProdutos.innerHTML = 'Carregando compras...'
34    getCompras(true)
35  }
```

Fonte: Elaborada pelo autor.

A Figura 4.10 apresenta como as listas de produtos são geradas e como elas são personalizadas de acordo com o tipo de listagem (produtos, itens de carrinho ou compras já realizadas),

onde até os botões são inseridos ou removidos de acordo com o contexto da aplicação.

Figura 4.10: Geração das listas de produtos no *front end* da aplicação

js > cards.js > ...

```
const buttonBuy = document.getElementById('buy')
function mostrarProdutos(produtos, tipo) {
  divProdutos.innerHTML = ''
  if (produtos.length == 0) {
    divProdutos.innerHTML = `Não há produtos em ${tipo} neste momento`
  } else {
    produtos.forEach(item => {
      let detalhes = JSON.stringify(item.detalhes)
      detalhes = detalhes.replace(/[{",:}]/g, match => {
        if (match == ':') return ': '
        if (match == ',') return '\n'
        return ''
      })
      let divCardProduto = document.createElement('div')
      divCardProduto.setAttribute('class', 'card')
      let nomeProduto = document.createElement('h4')
      nomeProduto.appendChild(document.createTextNode(item.nome))
      divCardProduto.appendChild(nomeProduto)
      let contentPre = document.createElement('pre')
      contentPre.appendChild(document.createTextNode(detalhes))
      divCardProduto.appendChild(contentPre)
      let button = document.createElement('button')
      item = JSON.stringify(item)
      if (tipo == 'produtos') {
        button.setAttribute('class', 'addToChart')
        button.appendChild(document.createTextNode('Adicionar no carrinho'))
        button.setAttribute('onclick', 'porNoCarrinho(' + item + ')')
        divCardProduto.appendChild(button)
      } else if (tipo == 'carrinho') {
        button.setAttribute('class', 'remove')
        button.appendChild(document.createTextNode('Remover do carrinho'))
        button.setAttribute('onclick', 'removerDoCarrinho(' + item + ')')
        divCardProduto.appendChild(button)
      }
      divProdutos.appendChild(divCardProduto)
    })
    if (tipo == 'carrinho') {
      buttonBuy.style.display = 'initial'
    } else {
      buttonBuy.style.display = 'none'
    }
  }
}
```

Fonte: Elaborada pelo autor.

A Figura 4.11 mostra como transitam entre *back end* e o *front end* da aplicação através de requisições criadas com o fetch. A função postProdutos() não é usada durante os experimentos, mas exemplifica como novos produtos poderiam ser criados no servidor.

Figura 4.11: Gestão dos produtos no *front end* da aplicação

js > produtos.js > ...

```
1   var produtos = null;
2
3   async function getProdutos(mostrar) {
4     try {
5       if (produtos == null) {
6         let response = await fetch('http://localhost:3000/produtos')
7         let data = await response.json()
8         produtos = data != null ? data : []
9         console.log('Produtos obtidos do PostgreSQL')
10      }
11
12      if (mostrar && ultimaEscolha == 'produtos') {
13        mostrarProdutos(produtos, 'produtos')
14      }
15    } catch (error) {
16      console.log(error)
17    }
18  }
19
20  async function postProdutos(data) {
21    try {
22      let response = await fetch('http://localhost:3000/produtos', {
23        headers: { 'Content-Type': 'application/json' },
24        body: JSON.stringify(data),
25        method: 'POST'
26      })
27      console.log(await response.json())
28    } catch (error) {
29      console.log(error)
30    }
31  }
```

Fonte: Elaborada pelo autor.

A Figura 4.12, mostra que, da mesma forma como ocorre com os produtos, as informações do carrinho de compras também transitam entre o *back end* e o *front end* através de requisições usando o fetch. Nesse caso, um id de usuário ("clienteA") também é informado na requisição para que o *back end* saiba quais dados precisam ser acessados no servidor.

Figura 4.12: Gestão do carrinho de compras no *front end* da aplicação (parte A)

js > carrinho.js > ...

```
1   var carrinho = null
2
3   async function getCarrinho(mostrar) {
4     try {
5       if (carrinho == null) {
6         let response = await fetch('http://localhost:3000/carrinho/clienteA')
7         data = await response.json()
8         carrinho = data != null ? data : []
9         console.log('Carrinho salvo do Redis')
10      }
11
12      if (mostrar && ultimaEscolha == 'carrinho') {
13        mostrarProdutos(carrinho, 'carrinho')
14      }
15    } catch (error) {
16      console.log(error)
17    }
18  }
19
20  async function postCarrinho(data) {
21    try {
22      let response = await fetch('http://localhost:3000/carrinho/clienteA', {
23        headers: { 'Content-Type': 'application/json' },
24        body: JSON.stringify(data),
25        method: 'POST'
26      })
27      console.log(await response.json() + '. Carrinho persistido no Redis')
28    } catch (error) {
29      console.log(error)
30      alert('Falha ao adicionar produto ao carrinho :(')
31    }
32  }
```

Fonte: Elaborada pelo autor.

A Figura 4.13 exibe o restante do código fonte contido em carrinho.js. Nessa parte, estão dispostas as funções responsáveis por permitir que o usuário possa inserir e remover itens do carrinho de compras.

Figura 4.13: Gestão do carrinho de compras no *front end* da aplicação (parte B)

js > carrinho.js > ...

```
34  function porNoCarrinho(produto) {
35    let existeNoCarrinho = false
36    carrinho.forEach(itemCarrinho => {
37      if (JSON.stringify(itemCarrinho) === JSON.stringify(produto)) {
38        alert('Já adicionado ao carrinho!')
39        existeNoCarrinho = true
40      }
41    })
42
43    if (!existeNoCarrinho) {
44      carrinho.push(produto)
45      alert('Produto adicionado ao carrinho :)')
46      postCarrinho(carrinho)
47    }
48  }
49
50  function removerDoCarrinho(produto) {
51    let novoCarrinho = []
52    carrinho.forEach(itemCarrinho => {
53      if (JSON.stringify(itemCarrinho) !== JSON.stringify(produto)) {
54        novoCarrinho.push(itemCarrinho)
55      }
56    })
57
58    carrinho = novoCarrinho
59    alert('Produto removido do carrinho :)')
60    postCarrinho(carrinho)
61    mostrarProdutos(carrinho, 'carrinho')
62  }
```

Fonte: Elaborada pelo autor.

A Figura 4.14 exibe como os dados das compras são tratados de forma semelhante aos dados do carrinho de compras como é mostrado na figura anterior. A função comprarTudo() requisita modificações em dois bancos de dados sobre o mesmo conjunto de informações. Nesse caso, não está sendo realizado o controle adequado para garantir que as compras realizadas realmente sejam removidas do carrinho e sejam armazenadas na área de compras confirmadas, podendo gerar problemas de consistência de dados na aplicação. Essa figura também mostra como a data é obtida para se tornar um atributo de relação na compra do produto, que é obtida diretamente no *front end* da aplicação para manter a simplicidade.

Figura 4.14: Gestão de compras no *front end* da aplicação

js > compras.js > ...

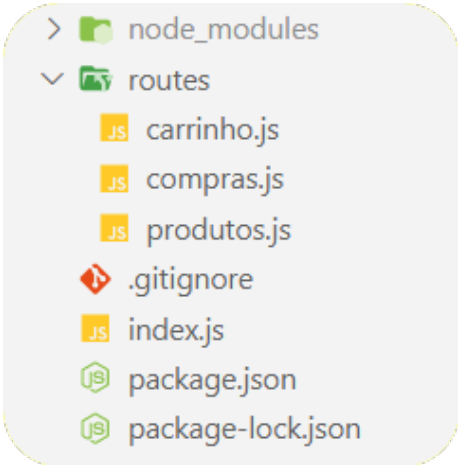
```
var compras = null

async function getCompras(mostrar) {
  try {
    if (compras == null) {
      let response = await fetch('http://localhost:3000/compras/clienteA')
      let data = await response.json()
      compras = data != null ? data.compras : []
      console.log('Compras obtidas do MongoDB')
    }

    if (mostrar && ultimaEscolha == 'compras') {
      mostrarProdutos(compras, 'compras')
    }
  } catch (error) {
    console.log(error)
  }
}

async function postCompras(data) {
  try {
    let response = await fetch('http://localhost:3000/compras/clienteA', {
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data),
      method: 'POST'
    })
    console.log(await response.json() + '. Compras salvas no MongoDB')
  } catch (error) {
    console.log(error)
  }
}

function comprarTudo() {
  let date = new Date()
  date.toLocaleString('pt-BR', { timeZone: 'America/Sao_Paulo' });
  carrinho.forEach(itemCarrinho => {
    itemCarrinho.detalhes["Data de compra"] = date.toLocaleDateString()
    compras.push(itemCarrinho)
  })
  postCompras(compras)

  carrinho = []
  goToCarrinho(true)
  postCarrinho(carrinho)
  alert('Os produtos do carrinho foram comprados')
}
```

Fonte: Elaborada pelo autor.

4.1.3.2 Back end da aplicação (API em Node.js)

A Figura 4.15 apresenta a hierarquia de pastas e arquivos na parte do *back end* da aplicação. A estrutura inicial foi criada usando o comando "npm init" do Node.Js para, em seguida, serem instalados os recursos adicionais e criados os diretórios e arquivos novos.

Figura 4.15: Hierarquia de pastas e arquivos no *back end* da aplicação



Fonte: Elaborada pelo autor.

A Figura 4.16 apresenta as informações básicas do projeto e quais foram os recursos usados para o desenvolvimento conforme explicado a seguir:

- A biblioteca "cors" é utilizada para permitir que o cliente acesse os recursos da API;
- O *framework* "express" facilita a criação da API ao permitir o tratamento de rotas nas solicitações HTTP e a utilização de *middlewares*, que simplificam a execução de funções entre as requisições HTTP e as respostas que são retornadas pelo servidor;
- As bibliotecas "mongoose", "pg" e "redis" permitem que a aplicação interaja com os bancos de dados MongoDB, PostgreSLQ e Redis respectivamente;
- A biblioteca "nodemon" é usada para reiniciar o servidor automaticamente após alterações no código do *back end*.

Esses recursos são chamados de dependências do projeto, foram instalados pelos comandos "npm install express cors mongoose pg redis" e "npm install nodemon -D" e são armazenados dentro da pasta "node_modules". O "-D", no caso do nodemon, indica que ele é uma

dependência de desenvolvimento, ou seja, é utilizado somente durante a codificação e não será útil em produção.

Figura 4.16: Informações e bibliotecas do *back end* da aplicação

package.json > ...

```
{
  "name": "tcc-crummenauerca",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "author": "Cezar Augusto Crummenauer",
  "description": "A college work about databases",
  ▷ Debug
  "scripts": {
    "start": "nodemon index.js"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "mongoose": "^5.10.3",
    "pg": "^8.3.3",
    "redis": "^3.0.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.4"
  }
}
```

Fonte: Elaborada pelo autor.

A Figura 4.17 mostra como o servidor entra em funcionamento após a execução do comando "npm start", que dispara o *script* personalizado "nodemon index.js" da Figura 4.16.

Figura 4.17: Inicialização do servidor (*back end* da aplicação)

```
C:\Users\crumm\Work\TCC\0 Tex\apps\servidor>npm start

> tcc-crummenauerca@1.0.0 start C:\Users\crumm\Work\TCC\0 Tex\apps\servidor
> nodemon index.js

[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
O servidor está ativo
Conectado ao banco de dados Redis
Conectado ao banco de dados MongoDB
Conectado ao banco de dados PostgreSQL
```

Fonte: Elaborada pelo autor.

A Figura 4.18 apresenta como a estrutura básica do servidor foi criada utilizando o *framework* Express. Da linha 9 até a linha 13 dessa figura, são utilizados *middlewares* para definir o tratamento de rotas, utilização do cors() e definição do JSON como notação para a transição de dados. Nessa parte, também é possível encaixar outros *middlewares* para, por exemplo, fazer a validação de dados e verificar se o solicitante em cada requisição tem o direito ou não de acessar os recursos solicitados.

Figura 4.18: Arquivo principal do *back end* da aplicação

index.js > ...

```
1   const express = require("express")
2   const cors = require('cors')
3   const app = express()
4
5   const compras = require('./routes/compras')
6   const produtos = require('./routes/produtos')
7   const carrinho = require('./routes/carrinho')
8
9   app.use(cors())
10  app.use(express.json())
11  app.use('/compras', compras)
12  app.use('/produtos', produtos)
13  app.use('/carrinho', carrinho)
14
15  app.listen(3000, function () {
16    console.log("O servidor está ativo")
17  })
```

Fonte: Elaborada pelo autor.

Nessa figura, as requisições HTTP são encaminhadas para tratamento em arquivos separados. Por exemplo, todas as requisições relacionadas com o carrinho de compras são tratadas no arquivo routes/carrinho.js. O servidor passa a ouvir as requisições HTTP no endereço da máquina (*localhost*) na porta 3000 como mostrado nas linhas 15,16 e 17 dessa figura.

Para manter a simplicidade, não foram definidos tratamentos robustos para possíveis erros. Também não foram especificados códigos de status específicos para as respostas de requisições como o código HTTP "200 OK" de sucesso.

A Figura 4.19 apresenta como é realizada a conexão com o banco de dados PostgreSQL e como ocorre o tratamento de rotas via GET e POST para a obtenção e salvamento de dados relacionados com os produtos. Neste caso, para suprir as necessidades dos experimentos, o método get retorna todos os produtos, não se preocupando com a paginação dos dados.

Figura 4.19: Gestão dos dados dos produtos no *back end* da aplicação

routes > produtos.js > ...

```
1   const express = require('express')
2   const router = express.Router()
3
4   const { Client } = require('pg')
5   const client = new Client({
6     user: "postgres",
7     password: "1234",
8     host: "localhost",
9     port: 5432,
10    database: "tcc-database"
11  })
12
13  client.connect().then(() => {
14    console.log('Conectado ao banco de dados PostgreSQL')
15  })
16
17  router.get("/", function (req, res) {
18    client.query("select * from produto")
19      .then(results => res.json(results.rows))
20      .catch(error => console.log(error))
21  })
22
23  router.post("/", function (req, res) {
24    client.query(`insert into produto (nome) values ('${req.body.nome}')`)
25      .then(() => res.json('OK'))
26  })
27
28  module.exports = router
```

Fonte: Elaborada pelo autor.

Nessa figura, dados sigilosos como a senha e nome de usuário do banco de dados são expostos. Definir variáveis de ambiente ajuda a evitar que dados sensíveis fiquem desprotegidos ao compartilhar o código fonte da aplicação com outros desenvolvedores.

A Figura 4.20 mostra qual foi o código SQL usado para os testes na aplicação. Embora pudesse haver relacionamentos como a categorização dos produtos, isso foi evitado por não ser o foco dos experimentos.

Figura 4.20: Código SQL usado para testes na aplicação

```sql
CREATE TABLE Produto (
  id SERIAL,
  nome VARCHAR(256),
  detalhes JSON,
  PRIMARY KEY(id)
);

INSERT INTO Produto (nome, detalhes)
VALUES ('Notebook', '{
  "Tamanho": "14 polegadas",
  "Memória RAM": "8 GB",
  "Armazenameto": "500 GB",
  "Tipo de armazenamento": "SSD"
}'), ('Carro', '{
  "Ano": 2010,
  "Tipo": "Sedan",
  "Ar condicionado": "Sim",
  "Quantidade de portas": "4 portas"
}'), ('Máquina de lavar roupas', '{
  "Faz centrifugação": "Sim",
  "Peso suportado": "16 Kg"
}'), ('HDD para notebook', '{
  "Tecnologia": "Sata 3",
  "Rotação": "7200 RPM",
  "Capacidade": "2 TB",
  "Cache": "128 MB"
}');
```

Fonte: Elaborada pelo autor.

O campo JSON possibilita o armazenamento de entidades heterogêneas em um banco de dados relacional. Isso é útil nesta aplicação, uma vez que os produtos são caracteristicamente diferentes. Embora existam algumas limitações, esse tipo de campo também pode ser indexado e envolvido em consultas utilizando a linguagem SQL, permitindo que atributos específicos do campo JSON sejam consultados, alterados ou removidos nas entidades desejadas.

A Figura 4.21 apresenta como é realizada a conexão com o banco de dados Redis. Nesta área há o tratamento dos métodos get e set relacionados com a obtenção e salvamento dos dados do carrinho de compras. Os dados são transformados em uma *string* e são salvos e recuperados com base em identificadores de usuário (usados como chaves no Redis) recebidos juntos com os dados das requisições vindas do *front end*. O tamanho máximo de *string* por chave no banco de dados Redis é 512 MB.

Figura 4.21: Gestão dos dados do carrinho no *back end* da aplicação

routes > carrinho.js > ...

```
1   const express = require('express')
2   const router = express.Router()
3
4   const redis = require('redis')
5   var client = redis.createClient()
6   client.on('connect', () => console.log('Conectado ao banco de dados Redis'))
7
8   router.get("/:id", function (req, res) {
9     client.get(req.params.id, (err, reply) => res.json(JSON.parse(reply)))
10  })
11
12  router.post("/:id", function (req, res) {
13    client.set(req.params.id, JSON.stringify(req.body), (err, reply) => {
14      console.log(`Carrinho de "${req.params.id}" salvo do Redis`)
15      res.json(reply)
16    })
17    client.expire(req.params.id, 3600 * 48); // Carrinho expira em 2 dias
18  })
19
20  module.exports = router
```

Fonte: Elaborada pelo autor.

O carrinho do usuário permanece salvo por 48 horas (48 * 3600 segundos de uma hora) após a última atualização. Dessa forma, se por algum motivo o cliente não efetivar a compra nesse período, o carrinho será eliminado automaticamente do banco de dados Redis.

A Figura 4.22 apresenta como é realizada a conexão com o banco de dados MongoDB. A biblioteca Mongoose possibilita a criação de um esquema fixo de dados para todos os documentos que são armazenados, mas esse não é o objetivo da aplicação e, dessa forma, o atributo "strict" foi definido como *false* para permitir que os documentos possam ser heterogêneos.

Nesta parte, também está presente o tratamento de rotas para acesso aos métodos get e post, mas agora para a obtenção e salvamento de dados de compras já efetuadas pelos usuários.

Os dados são salvos e recuperados no formato JSON e novamente usa-se o identificador de usuário que vem junto na requisição para fazer a identificação do documento desejado.

Figura 4.22: Gestão dos dados das compras no *back end* da aplicação

routes > compras.js > ...

```
1   const express = require('express')
2   const router = express.Router()
3
4   const mongoose = require('mongoose')
5   const Compras = mongoose.model('compras',
6     new mongoose.Schema(
7       {}, { strict: false }
8     )
9   )
10
11  mongoose.connect('mongodb://localhost:27017/tccDatabase', {
12    useNewUrlParser: true,
13    useUnifiedTopology: true
14  }, () => {
15    console.log('Conectado ao banco de dados MongoDB')
16  })
17
18  router.get("/:id", function (req, res) {
19    Compras.findOne({ cliente: req.params.id })
20      .then(data => res.json(data))
21  })
22
23  router.post("/:id", function (req, res) {
24    Compras.findOneAndUpdate(
25      { cliente: req.params.id },
26      { compras: req.body },
27      { upsert: true }
28    ).then(() => {
29      res.json('OK')
30      console.log(`Compras de "${req.params.id}" salvas no MongoDB`)
31    }).catch(error => {
32      console.log(error)
33    })
34  })
35
36  module.exports = router
```

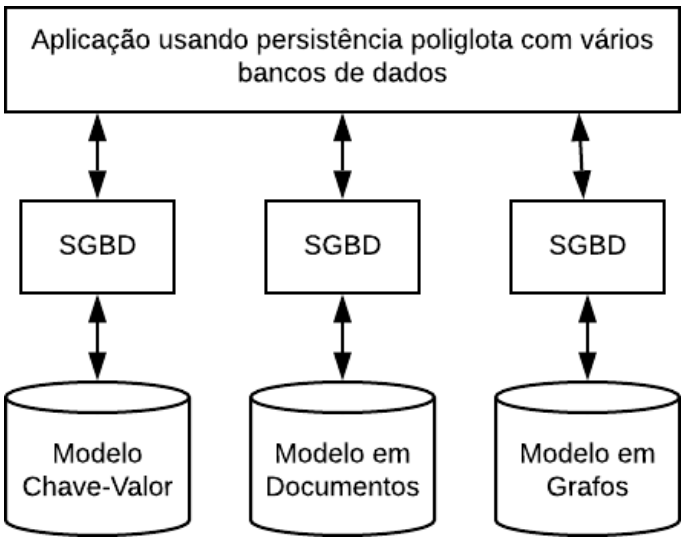
Fonte: Elaborada pelo autor.

4.2 BANCOS DE DADOS DO MULTIMODELOS

Em alguns casos, a implementação de persistência poliglota em determinadas aplicações pode ser realizada com o auxílio de bancos de dados multimodelos. Um banco de dados multimodelo se caracteriza por ser um único SGBD que armazena e consulta dados em modelos de dados NoSQL diferentes de forma centralizada, evitando a interação com vários SGBDs diferentes para gerenciar os dados de uma aplicação (AQUINO et al., 2019).

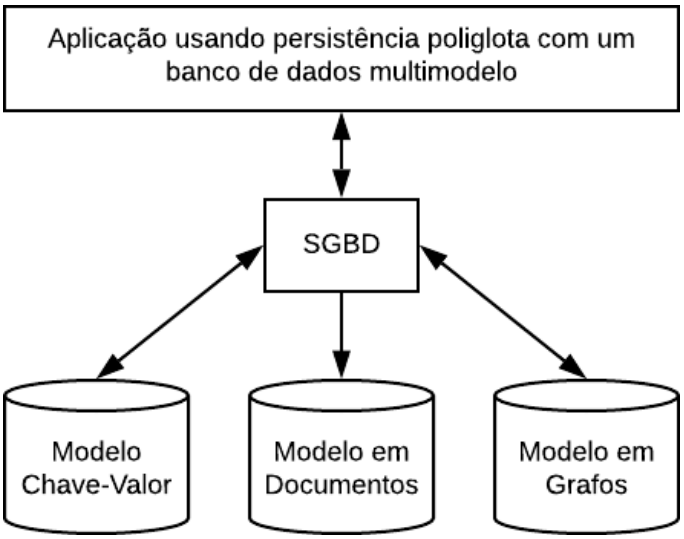
A Figura 4.23 esquematiza a persistência poliglota de dados usando vários bancos de dados, enquanto a Figura 4.24 esquematiza a persistência poliglota utilizando apenas um banco de dados multimodelo.

Figura 4.23: Persistência poliglota usando vários bancos de dados



Fonte: Elaborada pelo autor.

Figura 4.24: Persistência poliglota usando um banco de dados multimodelo



Fonte: Elaborada pelo autor.

ArangoDB[16] e OrientDB[17] são exemplos de bancos de dados multimodelos, que suportam os modelos Chave-valor, Documentos e Grafos.

Neste capítulo, foram exploradas possibilidades para a criação de uma aplicação usando persistência poliglota, onde os diferentes tipos de bancos de dados possuíram diferentes responsabilidades dentro da aplicação para aprimorar o funcionamento dessa aplicação. O próximo e último capítulo irá apresentar as considerações finais deste trabalho sobre bancos de dados NoSQL.

[16] ArangoDB - https://www.arangodb.com/. Acessado em 31 de agosto de 2020
[17] OrientDB - https://orientdb.org/. Acessado em 31 de agosto de 2020

5 CONSIDERAÇÕES FINAIS

Mesmo que existam muitas tecnologias para o gerenciamento de dados e que várias dessas já estejam bem consolidadas, a implementação de uma aplicação que possa gerenciar grandes volumes de dados de maneira rápida e confiável ainda é um grande desafio para os desenvolvedores.

Cada tecnologia de armazenamento de dados foi criada para atender necessidades específicas e é possível perceber que, no contexto de *Big Data*, não é adequado utilizar apenas uma delas para o desenvolvimento de aplicações.

Dentre todos os resultados observados durante a realização desse trabalho, podem ser citados os seguintes:

- Apesar dos problemas de desempenho em aplicações denominadas *Big Data*, os sistemas de banco de dados relacionais ainda possuem grande importância não apenas por proporcionarem uma linguagem de consulta e manipulação robusta dos dados, como também por implementar e abstrair uma série de mecanismos que buscam garantir a integridade e consistência dos dados;

- Os bancos de dados NoSQL do tipo Chave-Valor são adequados para a gerência de dados transitórios, que não precisam de garantias de durabilidade como sessões de usuário. No caso do Redis, por ser um banco de dados em memória, pode ser utilizado como cache de dados para reduzir o acesso ao disco e, como resultado, aprimorar drasticamente a velocidade de leitura e escrita desses dados;

- Bancos de dados NoSQL do tipo Documentos como o MongoDB são adequados para dados heterogêneos como produtos em um sistema de *e-commerce*, permitem a melhora da produtividade no desenvolvimento das aplicações pela alta flexibilidade para incrementos e alterações na estrutura de dados e possibilitam a melhora de desempenho ao evitar *joins* em situações específicas através da desnormalização facilitada dos dados. Entretanto não são muito adequados para a criação de relacionamentos complexos entre dados, tendo em vista que a implementação desses relacionamentos precisa ser definida no código da aplicação e podem tornar a escrita mais custosa do que a leitura em relacionamentos encaixados (*embedded*);

- Bancos de dados do tipo Colunas como o Cassandra são adequados para aplicações que precisam de alta performance para atender cargas massivas de requisições como ocorre em grandes redes sociais e, para isso, não suportam relacionamentos entre dados, sendo necessário a replicação em diferentes subconjuntos de dados específicos para atender as diversas funcionalidades possíveis em uma aplicação;

- Sistemas baseados no Modelo em Grafos como o Neo4j são muito convenientes para consultas envolvendo relacionamentos complexos de dados, em especial nos autorrelacionamentos muito comuns nos sistemas de recomendação;

- Bancos de dados multimodelos podem facilitar a implementação da persistência poliglota em aplicações, uma vez que acessam vários modelos de dados em um único SGBD;

- Bancos de dados como o InfluxDB podem ser convenientes para a gestão de dados em aplicações com necessidades temporais específicas.

Diante de todas as informações obtidas com os estudos e experimentos para a realização deste trabalho, foi possível observar que, de forma geral, a manipulação adequada de dados utilizando tecnologias convenientes para cada situação tem o potencial de tornar os sistemas mais eficientes, seguros e baratos para suportar um volume crescente de dados gerados por um número cada vez maior de usuários.

REFERÊNCIAS

ABRAMOVA, V.; BERNARDINO, J. NoSQL databases: mongodb vs cassandra. In: C* CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2013. p.14–22.

AQUINO, A. C. et al. **Uma análise comparativa de sistemas de gerenciamento de bancos de dados NoSQL multimodelo**. [S.l.]: Florianópolis, SC, 2019.

CASSANDRA. **Manage massive amounts of data, fast, without losing sleep**. Acessado em 14 de novembro de 2019, http://cassandra.apache.org/.

CLAUDINO, M.; SOUZA, D.; SALGADO, A. C. Mapeamentos conceituais entre os modelos Relacional e NoSQL: uma abordagem comparativa. **Revista Principia,(28)**, [S.l.], p.37–50, 2015.

DATASTAX. **Data replication**. Acessado em 19 de novembro de 2019, https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html.

LÓSCIO, B. F.; OLIVEIRA, H. d.; PONTES, J. d. S. NoSQL no desenvolvimento de aplicações Web colaborativas. **VIII Simpósio Brasileiro de Sistemas Colaborativos**, [S.l.], v.10, n.1, p.11, 2011.

MONGODB. **The database for modern applications**. Acessado em 23 de outubro de 2019, https://redis.io/.

NEO4J. **What Is Neo4j?** Acessado em 26 de outubro de 2019, https://neo4j.com/neo4j-graph-database/.

POKORNY, J. NoSQL databases: a step to database scalability in web environment. **International Journal of Web Information Systems**, [S.l.], v.9, n.1, p.69–82, 2013.

REDIS. **Introduction to Redis**. Acessado em 23 de outubro de 2019, https://redis.io/topics/introduction.

SADALAGE, P. J.; FOWLER, M. **NoSQL distilled**: a brief guide to the emerging world of polyglot persistence. [S.l.]: Pearson Education, 2013.

SAMPAIO, P. J.; OLIVEIRA KNOP, I. de. Desempenho de Aplicações Web: um estudo comparativo utilizando o software redis. **Caderno de Estudos em Sistemas de Informação**, [S.l.], v.2, n.2, 2015.

SCHREINER, G. A.; DUARTE, D.; SANTOS MELLO, R. dos. Sqltokeynosql: a layer for relational to key-based nosql database mapping. In: INTERNATIONAL CONFERENCE ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS & SERVICES, 17. **Proceedings...** [S.l.: s.n.], 2015. p.74.

SILBERSCHATZ, A.; SUNDARSHAN, S.; KORTH, H. F. **Sistema de banco de dados**. [S.l.]: Elsevier Brasil, 2016.