UNIFACCAMP
Centro Universitário Campo Limpo Paulista

Proposta e Implementação de Padrão ORM-DAO com Foco em Otimização do Consumo de Memória e do Tempo de Processamento

Denilson Silva Marçal

Dezembro / 2024

Dissertação de Mestrado em Ciência da Computação

Proposta e Implementação de Padrão ORM-DAO com Foco em Otimização do Consumo de Memória e do Tempo de Processamento

Esse documento corresponde à Dissertação apresentado à Banca Examinadora no curso de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista.

Campo Limpo Paulista, 13 de Dezembro de 2024.

Denilson Silva Marçal

Profa. Dra. Maria do Carmo Nicoletti
Orientadora

O presente trabalho foi realizado com apoio da
Coordenação de Aperfeiçoamento de Pessoal de Nível
Superior - Brasil (CAPES) - Codigo de Financiamento 001

Agradecimentos

A Deus, criador dos céus e da terra, fonte de toda sabedoria e inspiração, meu eterno agradecimento por me conceder a graça de concluir esta jornada.

Agradeço à minha esposa Elaine, pelo amor, paciência e constante apoio ao longo de toda essa jornada. À minha filha Thayna e ao meu filho Samuel, por serem minha maior inspiração e força para seguir em frente. Aos meus pais, Dório e Genisete, meu eterno reconhecimento por acreditarem em mim desde sempre e pelo incentivo incondicional aos meus estudos.

Expresso também minha profunda gratidão à minha orientadora, Professora Doutora Maria do Carmo Nicoletti, por sua dedicação, conhecimento compartilhado e orientação valiosa, que foram fundamentais para a concretização deste trabalho, e aos professores doutores Luis Mariano del Val Cura e Josué Junior Guimarães Ramos, membros da banca examinadora, pelas contribuições finais e valiosas sugestões que enriqueceram esta dissertação.

“A melhor mesa para se sentar é aquela na qual você é amado.”

Denilson Silva Marçal

Resumo. *Essa pesquisa apresenta o desenvolvimento de um padrão híbrido que integra o mapeamento objeto-relacional (ORM) com o padrão de design Data Access Object (DAO) para melhorar o consumo de memória e o tempo de processamento em aplicações de software. O estudo investiga estratégias eficientes de modelagem de base de dados, destacando o papel crucial das melhores práticas de programação para alcançar um desempenho ideal do sistema. Diferente dos ORMs tradicionais, que automatizam grande parte do mapeamento objeto-relacional e da geração de SQL, a abordagem baseada em DAO concede aos desenvolvedores maior controle sobre a sobrecarga de memória e processamento, facilitando o gerenciamento mais preciso dos recursos. Esse controle aprimorado resulta em um desempenho mais rápido e maior eficiência, especialmente em cenários com demandas de carga específicas. Ao eliminar camadas intermediárias comumente encontradas em sistemas ORM automatizados, o padrão proposto alcança uma redução significativa no consumo de+ recursos, melhorando o tempo de resposta da aplicação. Além disso, a pesquisa aborda os principais princípios de design e de técnicas para subsidiar o desenvolvimento do padrão ORM-DAO, com foco em ajustes de desempenho e otimização de memória, tornando-o atraente para desenvolvedores que buscam construir aplicações de alto desempenho e eficientes no uso de recursos.*

Abstract: *This research presents the development of a hybrid pattern that integrates Object-Relational Mapping (ORM) with the Data Access Object (DAO) design pattern to improve memory consumption and processing time in software applications. The study investigates efficient database modeling strategies, highlighting the crucial role of programming best practices to achieve optimal system performance. Unlike traditional ORMs, which automate much of the object-relational mapping and SQL generation, the DAO-based approach gives developers greater control over memory and processing overhead, facilitating more precise resource management. This enhanced control results in faster performance and greater efficiency, especially in scenarios with specific load demands. By eliminating intermediate layers commonly found in automated ORM systems, the proposed pattern achieves a significant reduction in resource consumption, improving application response time. Furthermore, the research addresses key design principles and techniques to support the development of the ORM-DAO pattern, with a focus on performance tuning and memory optimization, making it attractive to developers seeking to build high-performance and resource-efficient applications.*

Sumário

GLOSSÁRIO

ATM (Automated Teller Machine): Máquina de autoatendimento bancário que permite aos clientes realizar transações financeiras, como saques e depósitos.

DBOO (Database Object-Oriented): Tipo de banco de dados que combina conceitos de banco de dados com programação orientada a objetos, permitindo armazenar dados na forma de objetos, incluindo herança, encapsulamento e polimorfismo.

Cache: Área de armazenamento temporário que permite acesso rápido a dados frequentemente acessados.

Caching: Processo de armazenar dados em cache para melhorar a eficiência e a velocidade de recuperação dos dados.

Call: Chamada de função ou método em programação.

Calling: Ato de invocar uma função ou método em programação.

CRUD: (Create, Read, Update, Delete): Conjunto de operações básicas de manipulação de dados em uma base de dados.

DAO: (Data Access Object): Padrão de design que abstrai e encapsula todas as operações de acesso a dados.

DB: (Database): Conjunto organizado de dados armazenados e acessados eletronicamente.

DBMS: (Database Management System): Sistema de gerenciamento de banco de dados que interage com usuários e aplicações para capturar e analisar dados.

DDB: (Distributed Database): Banco de dados cujo armazenamento de dados é distribuído em diferentes locais geográficos.

DDBS: (Distributed Database System): Conjunto de bancos de dados distribuídos que funcionam como um sistema único.

GUI: (Graphical User Interface): Interface gráfica do usuário que permite interação com dispositivos eletrônicos por meio de elementos visuais.

HashMap: Estrutura de dados que mapeia chaves únicas a valores, permitindo acesso rápido aos dados.

HCE: (Highly Concurrent Environment): Ambiente de sistema onde muitas operações ou processos são executados simultaneamente.

HDBMS: (Hierarchical Database Management System): Sistema de gerenciamento de banco de dados hierárquico, onde os dados são organizados em uma estrutura de árvore.

HQL: (Hibernate Query Language): Linguagem de consulta orientada a objetos utilizada para realizar consultas em bancos de dados com o framework Hibernate.

Impedance Mismatch: Desajuste de impedância refere-se às diferenças entre os modelos de dados de sistemas de base de dados relacionais e os sistemas de programação orientada a objetos.

JDBC: (Java Database Connectivity): API do Java que permite a execução de operações em banco de dados a partir de programas escritos em Java.

JOIN: Operação em bancos de dados que combina colunas de duas ou mais tabelas com base em uma condição relacionada.

JPA: (Java Persistence API): Framework que permite mapear objetos Java para bancos de dados relacionais de forma transparente.

Lazy Loading: Técnica de carregamento de dados sob demanda, carregando-os apenas quando realmente necessário.

Matching: Processo de encontrar correspondências entre conjuntos de dados ou elementos.

MVC (Model-View-Controller): Padrão de design de software que separa a aplicação em três componentes principais: Model (modelo), View (visão) e Controller (controlador).

MS (Milessegundos): Unidade de tempo equivalente a um milésimo de segundo (1/1000 de segundo).

OLAP (Online Analytical Processing): Tecnologia que permite a análise rápida e interativa de dados multidimensionais.

OLTP (Online Transaction Processing): Categoria de sistemas de informação que facilita e gerencia aplicações orientadas a transações.

OO (Object-Oriented): Orientado a Objetos. Paradigma de programação que organiza o software em objetos, que encapsulam dados e comportamentos relacionados.

OOP (Object-Oriented Programming): Programação Orientada a Objetos. Método de programação baseado no conceito de "objetos", que podem conter dados e código para manipular esses dados.

ORM (Object-Relational Mapping): Mapeamento Objeto-Relacional. Técnica de programação que converte dados em objetos.

Query: Consulta em uma base de dados para recuperar informações específicas.

Query Language: Linguagem utilizada para fazer consultas em um base de dados, como SQL.

RDB (Relational Database): Banco de Dados Relacional. Tipo de base de dados que armazena e fornece acesso a dados relacionados de forma estruturada, utilizando tabelas.

Schema: Estrutura ou organização lógica de uma base de dados, definindo como os dados são armazenados e relacionados.

SOLID: Conjunto de princípios de design de software que visa tornar os sistemas mais compreensíveis, flexíveis e fáceis de manter.

SQL (Structured Query Language): Linguagem padrão para gerenciamento e manipulação de bancos de dados relacionais.

Stakeholders: Partes interessadas que têm interesse ou influência em um projeto ou organização.

UML (Unified Modeling Language): Linguagem de modelagem unificada utilizada para especificação, visualização, construção e documentação de sistemas de software.

Views: Visões em uma base de dados que são consultas armazenadas que podem ser tratadas como tabelas virtuais.

Lista de Tabelas

Lista de Figuras

Capítulo 1
Introdução

1.1 Introdução

A pesquisa desenvolvida e descrita nessa dissertação de mestrado teve como objetivo investigar e propor um padrão de ORM (*Object-Relational Mapping*) baseado em DAO (*Data Access Object*) que fosse eficiente em termos de consumo de memória e tempo de processamento.

É fato que, cada vez mais, novas metodologias, modelos, linguagens de programação, arquiteturas de base de dados, aplicativos e muitas outras ferramentas estão sendo constantemente disponibilizadas aos desenvolvedores de software. Se por um lado a adoção de muitas dessas novas tendências/softwares contribuem para tornar o desenvolvimento de novos sistemas computacionais mais ágil, com benefícios óbvios, por outro podem impactar, de maneira significativa, o tempo e esforços empregados por desenvolvedores, na troca ou na adaptação dos sistemas correntemente ativos.

O uso de *Object-Oriented Programming* (OOP) em um ambiente computacional que emprega *Relational DataBase* (RDB) é uma tarefa difícil de ser realizada e requer um investimento anterior em formação e treinamento de desenvolvedores. Essa constatação, de certa forma, traz incertezas com relação aos resultados do emprego dessa combinação, particularmente aqueles relacionados ao próprio desenvolvimento do software pretendido e ao cumprimento desta tarefa dentro do tempo planejado.

Uma das estratégias adotada no trabalho de pesquisa realizado foi a de introduzir um identificador único para todas as tabelas da base de dados relacional, incluindo aquelas geradas a partir de relacionamentos N para N. Essa estratégia foi implementada por meio da criação de chaves únicas e do uso de chaves estrangeiras como chaves alternativas. Ambas as estratégias propostas nesta pesquisa podem ser

consideradas um avanço significativo para a promoção da compatibilidade entre o modelo de classes de tabela e o modelo relacional da base de dados.

O trabalho realizado investiu também na implementação de recursos da Álgebra Relacional que contribuíram para tornar buscas mais flexíveis e eficientes nas consultas SQL, o que introduziu maior versatilidade quando da manipulação de condições complexas, o que é crucial em cenários em que a integridade dos dados é fundamental.

Durante a realização da pesquisa foi utilizado o software HashMap, definido por [Bloch 2017], para implementar o *caching* de objetos, que tende a oferecer melhorias significativas no consumo de memória e na eficiência do sistema. Ao armazenar objetos em um HashMap, o acesso rápido e eficiente às instâncias previamente carregadas será garantido. Além disso, o uso do HashMap é ideal para lidar com operações de cache devido à sua capacidade de associar chaves únicas a valores correspondentes de maneira rápida e eficiente. Detalhes sobre o HashMap e análise de seu bom desempenho estão descritos no Capítulo 4.

A pesquisa realizada teve uma fase que contemplou um levantamento bibliográfico inicial, revisitando muitos dos conceitos envolvidos em Base de Dados Relacional e características intrínsecas à Programação Orientada a Objetos, como uma preparação para o estudo e desenvolvimento da estratégia pretendida, voltada ao uso de Mapeamento Objeto-Relacional.

A estratégia proposta e implementada foi a da criação e implementação de um padrão de desenvolvimento de software que englobasse ganhos em otimização do consumo de memória e do tempo de processamento, bem como promovesse a compatibilidade entre o modelo orientado a objetos e o modelo relacional, proporcionando uma base sólida para o desenvolvimento de aplicações robustas e eficientes.

Além desse capítulo de introdução, o texto apresentado nesta dissertação está organizado como segue.

Os dois primeiros capítulos que seguem abordam, respectivamente, conceitos e aspectos relevantes relacionados a DB e RDBs, que são tratados no Capítulo 2, e a

descrição do cenário envolvido em ambientes computacionais que adotam ORM, no Capítulo 3.

O Capítulo 4 apresenta as bases teóricas e práticas que sustentam a pesquisa desenvolvida nesta dissertação, com foco na implementação do padrão ORM-DAO, uma proposta que combina as vantagens do mapeamento objeto-relacional com a eficiência do padrão de acesso a dados (DAO). O trabalho explora como a introdução de identificadores únicos para cada t-upla em tabelas relacionais, aliada ao uso de estruturas como o HashMap em ambiente Java, pode otimizar a persistência e recuperação de dados. Além disso, o capítulo aborda a implementação de uma álgebra relacional personalizada, projetada para ampliar a flexibilidade e eficiência em consultas SQL, garantindo compatibilidade e desempenho em sistemas que integram modelos de classes e bases de dados relacionais.

O Capítulo 5 investe na avaliação experimental do padrão ORM-DAO proposto, por meio da realização de um conjunto de experimentos. Para fim de comparação de desempenho, os mesmos experimentos também foram realizados utilizando o padrão Spring Data JPA. Os resultados obtidos por ambos foram comparativamente analisados.

No Capítulo 6 é apresentado a conclusão e os trabalhos futuros.

Além dos seis capítulos, essa dissertação é composta também por seis anexos: o Anexo 1 aborda o conjunto das chamadas 12 regras de Codd, que substanciaram a proposta de RDBs, o Anexo 2 aborda uma linguagem diagramática adotada no texto, que é intitulada *Unified Modeling Language* (UML), o Anexo 3 aborda padrões de desenvolvimento de software, o Anexo 4 contempla o código DDL para criação do modelo de dados a ser usado, e por fim, o Anexo 5 contempla o código fonte de JAVA do padrão ORM-DAO.

A dissertação finaliza com a apresentação de Referências onde estão elencadas as publicações relacionadas às áreas de conhecimento envolvidas na pesquisa realizada, que foram consultadas/estudadas durante o desenvolvimento do trabalho e que subsidiaram, em algumas partes, a composição do texto dessa dissertação.

1.2 Revisão da Literatura

Para a identificação, análise e inclusão dos trabalhos relacionados nesta pesquisa, foi conduzida uma **revisão integrativa**, abrangendo publicações relevantes na área de desenvolvimento de software com ênfase em ORM. A revisão teve como objetivo identificar os principais avanços, desafios e contribuições teóricas e práticas relacionadas ao uso de ORM em ambientes de programação orientada a objetos (POO) e bancos de dados relacionais (RDB). O resultado da revisão subsidiou e direcionou o estabelecimento e o andamento da pesquisa.

A busca por fontes incluiu buscas em base de dados acadêmicas, artigos científicos e livros, com ênfase em obras amplamente reconhecidas, como os fundamentos ORdoM apresentados em [Elmasri et al., 2015] e a implementação pioneira do Hibernate descrita em [O'Neil, 2008]. Para capturar uma visão ampla e detalhada, foram selecionados estudos que abordam conceitos como modelagem, desafios de integração (como o *impedance mismatch* [Torres et al. 2017] [Teixeira 2017]) e estratégias de design e desempenho, conforme explorado em [Ambler 1997] [Keller 1997] [Lorenz et al. 2016].

Os critérios de inclusão priorizaram trabalhos que apresentassem análises críticas, propostas metodológicas ou estudos de caso com resultados práticos, como o uso de interfaces orientadas a objetos descrito em [Asenjo et al. 2023]. Para garantir a relevância e diversidade das fontes, foram excluídos estudos com enfoque limitado ou sem análise comparativa substancial. Essa metodologia assegurou uma base sólida para a compreensão dos desafios e avanços associados ao ORM no desenvolvimento de sistemas eficientes e escaláveis.

A motivação principal foi a de identificar trabalhos de pesquisa que fornecem um panorama geral e abrangente de tais propostas e, particularmente, dos recursos computacionais empregados por elas, que poderiam subsidiar o desenvolvimento de sistemas computacionais que necessitam de interações complexas com bancos de dados relacionais, como é o caso do projeto de pesquisa conduzido pelo autor.

O mapeamento objeto-relacional (ORM) é um tópico fundamental no desenvolvimento de software em ambientes de programação orientada a objetos (POO) e bancos de dados relacionais (RDB). Diversos autores exploraram os desafios

e nuances do ORM, fornecendo contribuições valiosas para a compreensão e evolução dessa abordagem [Elmasri et al. 2015].

A abordagem ORM foi implementada pela primeira vez no Hibernate, um projeto de código aberto para sistemas Java iniciado em 2002, com o objetivo de facilitar a integração entre objetos e bancos de dados relacionais [O'Neil 2008]. O ORM abstrai a complexidade da interação com o banco de dados, promovendo maior eficiência e clareza no desenvolvimento de sistemas.

Em [Byrne 2003], os autores analisam como o ORM pode modelar os papéis que objetos desempenham em um banco de dados relacional, contrastando com a abordagem tradicional baseada em entidade-relacionamento (ER). Eles argumentam que o ORM, com sua ênfase em atributos e relacionamentos, pode oferecer uma perspectiva mais prática e intuitiva sobre o design de bancos de dados relacionais, destacando suas vantagens na aplicação de restrições e conceitos de modelagem de dados.

Outro estudo relevante, apresentado em [Asenjo et al. 2023], detalha o desenvolvimento de uma interface de banco de dados orientada a objetos (BDOO) sobre um banco de dados relacional. Essa abordagem busca explorar conceitos abstratos do ORM e como eles podem facilitar a criação de projetos reais.

O conceito de *impedance mismatch*, que descreve as diferenças fundamentais entre os paradigmas POO e RDB, é explorado em profundidade por diversos autores, como [Torres et al. 2017] e [Teixeira 2017]. Esse conceito é considerado um dos principais desafios do ORM, uma vez que pode impactar significativamente a performance e a eficiência de sistemas. O uso de identificadores de objetos (OIDs) em bancos relacionais é uma das estratégias propostas para mitigar esses desafios, como destacado em [Ambler 1997].

Além disso, a importância dos padrões de design na implementação de ORM é amplamente discutida em [Torres et al. 2017], que identifica padrões arquiteturais e estruturais essenciais para a aplicação eficiente dessa tecnologia. A análise de ferramentas ORM e suas limitações também é apresentada em [Ogheneovo et al. 2013], com foco na necessidade de adaptação e avaliação de desempenho dos modelos de análise.

O desempenho do Hibernate, uma das ferramentas ORM mais utilizadas, é comparado com o do JDBC em [Ghandeharizadeh et at. 2014], enquanto em [Kisman et al. 2016], é introduzida a extensão Hibernate Criteria Extension (HCE), que simplifica consultas e otimiza o desenvolvimento. Adicionalmente, técnicas avançadas como *caching* e *lazy loading*, descritas em [Keller 1997] e [Lorenz et al. 2016], são abordadas como estratégias para melhorar o desempenho e escalabilidade de aplicações que utilizam ORM.

Capítulo 2
Aspectos Relevantes de Base de Dados Relacional

2.1 Sobre Bases de Dados

Muitos processos que envolvem tomadas de decisão efetivas em contextos organizacionais dependem da disponibilidade de dados confiáveis. A confiabilidade de dados é uma característica que depende e está associada a inúmeras outras, tais como precisão e atualização [Meeker *et al.* 2014].

Em um contexto organizacional à medida que novos sistemas computacionais são requisitados e desenvolvidos, muitos deles dependem da criação de novas estruturas de armazenamento, tanto para prover os sistemas computacionais com dados armazenados, quanto para a atualização e armazenamento dos dados gerados pelos sistemas.

Como comentam os autores em [Litwin *et al.* 1990], base de dados (DBs) foram propostas como uma solução ao problema de acesso compartilhado a arquivos heterogêneos criados por inúmeras aplicações autônomas, em um ambiente centralizado. Tais arquivos eram difíceis de serem mantidos e administrados e, usualmente, continham duplicações e vários tipos de heterogeneidade, tais como diferenças em nomeação, tipos de valores e estrutura de arquivo para dados similares. Tais inconveniências traziam dificuldades para conciliar consistência entre arquivos, privacidade e eficiência. Com o objetivo de superar essas dificuldades e tornar o uso de dados mais fácil e eficiente, arquivos autônomos foram substituídos por uma base de dados integrada globalmente, livre de duplicações e heterogeneidade e administrada sob um controle centralizado de um sistema de administração de base de dados, o que dava a cada aplicação a ilusão de ser o único usuário. De uma maneira simplista uma base de dados pode ser caracterizada como uma coleção de um imenso volume de dados.

Resumindo, pode ser dito que sistemas computacionais desenvolvidos em organizações usualmente são alimentados por dados armazenados em Bases de Dados (*Data Base - DB*). Essas DBs são gerenciadas por Sistemas de Administração de Base de Dados *(Data Base Management System - DBMS)* que, abordados de maneira simplista, podem ser considerados sistemas computacionais usados para gerenciar DBs, com funcionalidades de, entre outras (a) realizar cálculos complexos, (b) recuperar registros de dados com base em '*matching'* ou, então, uso de funções de comparação, (c) atualizar rapidamente altos volumes de registros.

DBMSs acessam e manipulam dados das DBs e atuam como uma espécie de interface entre o usuário final e a DB, permitindo que usuários possam criar, recuperar, atualizar e deletar dados na DB.

Nas operações de criar, atualizar e deletar dados, o conceito de transação é fundamental [Elmasri *et al.* 2015]. Uma transação pode ser definida como uma sequência de operações interligadas que são executadas como uma unidade indivisível [Ramakrishnan *et al.* 2003]. Isso significa que cada uma das operações que participa da transação deve ser bem-sucedida, caso contrário, a transação falha. Essa propriedade crucial garante a integridade e confiabilidade dos dados, mesmo em ambientes complexos e concorrentes [Gray *et al*. 2013].

Em [Jatana *et al.* 2012] os autores informam que a confiabilidade da DB é verificada com foco nas seguintes propriedades:

(1) *Atomicidade*, que pode ser traduzida como 'tudo ou nada'; se qualquer parte de uma transação está incompleta ou com problemas, a transação inteira falha,

(2) *Consistência*, que garante que antes e após qualquer transação a DB continua estável, em um estado válido,

(3) *Isolamento*, que garante a não interferência entre múltiplas transações sendo executadas ao mesmo tempo e, consequentemente, requerendo que transações concorrentes sejam serializadas,

(4) *Durabilidade*, que garante que uma vez que uma transação tenha sido processada ela permanece no mesmo estado *i.e.*, armazenada permanentemente

mesmo se aparecerem alguns erros ou mesmo se o sistema cair por alguma falha técnica ou por falta de energia.

Presentemente grandes organizações disponibilizam vários DBMSs autônomos que, usualmente, lidam com diferentes modelos de dados, diferentes linguagens *query* e diferentes esquemas; esses conceitos serão apresentados discutidos ao longo desse capítulo.

Com o objetivo de facilitar e organizar o acesso de usuários a tais sistemas é fundamental que o acesso compartilhado aos vários DBMs seja implementado por meio de um único modelo de dados e uma única linguagem *query*, o que pode ser concretizado e administrado por meio de um sistema heterogêneo de base de dados (HDBMS).

Como apresentado em [Grassmann *et al.* 1996], modelos apropriados de dados são utilizados para representar tabelas e suas relações associadas. O principal foco desse capítulo é apresentar e discutir o modelo de dados bem conhecido e largamente utilizado por organizações, identificado como modelo de dados relacional, que é adotado em Base de Dados Relacional (RDB), bases essas que são administradas por Sistemas de Base de Dados Relacional (RDBMS - *Relational Data Base Management System*).

Este capítulo apresenta os conceitos básicos relacionados a RDBs e RDBMSs com o objetivo de contextualizar a área em que a pesquisa sendo realizada se insere, e de formalizar várias definições relevantes relacionadas à área de pesquisa de RDBs.

Como comentado anteriormente, um DBMS pode ser considerado um sistema computacional que administra conjuntos de dados (DBs), em que os dados armazenados estão organizados de acordo com um modelo de dado. Dados são acessados pelo usuário por meio de uma interface identificada como *query language*, disponibilizada pelo DBMS. Nesse contexto, um *schema* representa a estrutura e a organização real do dado no sistema.

Um sistema de base de dados distribuído (DDBS) é composto por uma base de dados lógica associada, que é fisicamente distribuída, e um sistema de administração de base de dados distribuído (DDBMS), que fornece *queries* consistentes e

atualizações entre as bases de dados distribuídas (DDB). Além disso, em um DDBS todos os seus componentes físicos estão sob o controle do mesmo DDBMS, o que promove homogeneidade.

É importante realçar que um DDBMS tem apenas um modelo de dado, uma única linguagem *query* e seu *schema* é explicito. Um sistema de administração de DBs heterogêneos (HDBMS) pode ser abordado como um sistema de BDs distribuído, que tem como componentes bases de dados heterogêneas (HDB) e seus DBMSs são autônomos *i.e.*, são independentes uns dos outros. A heterogeneidade pode advir de arquiteturas computacionais diferentes, diferentes sistemas operacionais, modelos de dados, DBMSs, *schemas* e linguagem *query*.

As próximas seções deste capítulo foram substanciadas por um conjunto de pesquisas realizadas na área de DBs, evidenciadas por meio um levantamento bibliográfico, cujas principais fontes foram: [Codd 1970] [Litwin *et al.* 1990] [Sedighi 1993] [Grassmann *et al.* 1996] [Jatana *et al.* 2012] [Wikipedia 2024] [Wikipedia 2024a].

Resumindo o que foi comentado anteriormente, um DBMS pode ser considerado um sistema computacional que administra conjuntos de dados (DBs), em que os dados armazenados estão organizados de acordo com um modelo de dado. Dados são acessados pelo usuário por meio de uma interface identificada como *query language*, disponibilizada pelo DBMS, Nesse contexto, um *schema* representa a estrutura e a organização real do dado no sistema.

2.2 Caracterização e Principais Conceitos de Base de Dados Relacional (RDB)

Uma base de dados caracterizada como base de dados relacional (RDB) pode ser abordada de uma maneira simplista como uma base de dados que adota um modelo de dados identificado como relacional, como foi proposto em [Codd 1970]. O modelo relacional é subsidiado teoricamente por conceitos matemáticos, particularmente teoria dos conjuntos e teoria relacional. O modelo relacional tem por base que as estruturas lógicas de dados, tais como tabelas, *views* e índices estão separadas das estruturas de armazenamento físico.

O sistema computacional desenvolvido para manter RDBs é conhecido como sistema de administração de base de dados relacional (RDBMS). Assim como acontece com DBs, muitos RDBs encontrados junto ao mercado de produtos de software são disponibilizados com a opção de uso de uma SQL *(Structured Query Language)*, por meio da qual consultas e atualizações da RDB podem ser realizadas. A linguagem SQL será abordada com mais detalhes na Seção 2.5.

2.3 Estrutura Usada na Organização dos Dados em RDBs

Uma RDB pode ser caracterizada como uma BD que adota um modelo relacional de representação de dados como proposto em [Codd 1970]. O modelo relacional proposto organiza dados em uma ou mais tabelas (ou relações) com linhas (nomeadas também como registros ou t-uplas) e colunas (nomeadas como atributos ou t-uplas).

Uma relação é, pois, um conjunto de registros em que cada registro é descrito por um mesmo conjunto de atributos. Todos os valores associados a um determinado atributo compartilham o mesmo domínio dos dados associados ao atributo em questão e obedecem às mesmas restrições. Abordado de uma maneira simplista, cada registro da tabela representa a instanciação de um tipo de tabela como um vetor de valores de atributos, em que cada um dos valores está associado a um particular atributo que descreve o objeto instanciado.

O modelo relacional especifica que os registros que definem uma relação não têm uma ordem específica entre eles, assim como acontece com o conjunto de atributos que os descrevem. É importante informar, entretanto, que uma vez estabelecida a ordem como as colunas estão dispostas na tabela, tal ordem deve ser mantida.

Usualmente cada tabela/relação representa um *tipo de tabela* (*e.g.* Livro) e cada coluna representa um atributo que descreve o tipo de tabela considerado (*e.g.* código-interno, título do livro, código do autor, autor, data de nascimento do autor, papel do autor, ano da publicação, preço, código da editora, nome da editora) associado ao tipo de tabela sendo representada (‘Livro’, no caso).

Considere um exemplo bem simples de um sistema de registro de Livros de uma biblioteca. A Tabela 2.1 mostra a representação da tabela ‘Livro’ em que as dez colunas, da esquerda para a direita, representam os atributos que caracterizam a tabela

‘Livro’, a saber: ‘código’, ‘título do livro’, ‘código do autor’, ‘autor’, ‘data de nascimento do autor’, ‘papel do autor no livro’, ‘ano da publicação’, ‘preço’, ‘código da editora’ e ‘editora’. Registros (linhas da tabela) representam instâncias do tipo de tabela Livro e cada coluna representa um valor atribuído ao atributo associado à coluna. A Tabela 2.2 apresenta uma versão genérica da tabela Livro para exemplificar várias situações que serão utilizadas para ilustrar o processo de normalização. Nas próximas seções, será detalhada a normalização até a terceira forma normal, resultando na divisão da tabela Livro em quatro outras tabelas. Na Tabela 2.2 a tabela Livro não possui chave primária e não está normalizada. Os conceitos sobre tipos de chaves e normalização serão abordados nas seções subsequentes.

Tabela 2.1 Livros (tipo de tabela).

CÓDIGO_LIVRO	TÍTULO	CÓDIGO_AUTOR	AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR	ANO_PUB	PREÇO	CODIGO_EDITORA	EDITORA
CT123	Grafos	7, 1	T. Lima, A.Guarani	23/02/1972, 15/05/1970	PRINCIPAL, COAUTOR	1995	203,20	2	AZUL
RO487	Despedidas	3	D. Chapman	20/09/1968	PRINCIPAL	1998	197,80	4	VERDE
BI896	Abelhas	5	M. Delly	02/04/1964	PRINCIPAL	2021	180,00	2	AZUL
RO112	Lua Vermelha	3, 4	D. Chapman, E. Roland	20/09/1968, 18/08/1970	PRINCIPAL, COAUTOR	2023	276,00	3	ROXO
CT007	Cálculo	8	U.Wonder	14/04/1965	PRINCIPAL	1998	287,00	3	ROXO
EN369	Mecânica	6	N. Brasília	19/12/1973	PRINCIPAL	2023	194,43	1	AMARELO
CT995	Álgebra	8, 1	U.Wonder, A.Guarani	14/04/1965, 15/05/1970	PRINCIPAL, COAUTOR	1999	255,00	4	VERDE
BI452	Mamíferos	5, 2	M. Delly, C. Marinho	02/04/1964, 02/07/1965	PRINCIPAL, COAUTOR	2023	302,12	2	AZUL

Em RDBs as chamadas chaves primárias, estrangeiras e alternativas desempenham funções cruciais para garantir a integridade, consistência e confiabilidade dos dados armazenados. Cada tipo de chave possui características e aplicações específicas, conforme detalhado a seguir.

(1) Chave Primária (*Primary Key*): representa a identificação única de cada registro em uma tabela, impedindo duplicatas e garantindo a individualidade dos dados. Sua implementação, por meio de *constraints* (também chamadas de restrições), garante que cada registro possua um identificador único e não nulo [Date 2010].

(2) Chave Estrangeira (*Foreign Key*): estabelece relacionamentos entre tabelas, assegurando a integridade referencial. Elas garantem que os valores em uma tabela estejam presentes em outra tabela relacionada, evitando inconsistências e registros órfãos [Elmasri *et al.* 2015]. As restrições (*constraints*) associadas à chave estrangeira podem definir regras para atualizações e exclusões.

(3) Chave Alternativa (*Alternate Key*): embora tais chaves não sejam restrições por si só, representam colunas ou conjuntos de colunas que podem ser utilizadas para

identificar registros de forma única. Entretanto, não são a chave primária. Sua definição pode afetar as restrições de unicidade, além de oferecer flexibilidade na identificação de registros e otimizar o desempenho de consultas [Garcia-Molina *et al.* 2002].

A combinação das características e funcionalidades das chaves primárias, estrangeiras e alternativas, juntamente com a aplicação adequada de restrições, garante a qualidade e confiabilidade dos dados em RDBs, possibilitando assim a criação de modelos de dados robustos e eficientes para diversas aplicações.

Como benefícios do desenvolvimento e uso de base de dados seguindo o modelo relacional os autores em [Jatana *et al.* 2012] apontam:

(1) grande parte da informação é armazenada no DB e não na aplicação, de maneira que o DB é auto documentada;

(2) é fácil adicionar, atualizar ou deletar dados;

(3) traz as vantagens de sumarização, recuperação e relatórios sobre dados;

(4) A base de dados é estruturada de maneira tabular com tabelas fortemente relacionais; a natureza do DB é previsível e qualquer requisição de mudanças no *schema* da BD é razoavelmente simples.

Apesar dos benefícios que uma RDB pode trazer, traz junto alguns problemas, tais como:

(1) não é adequada para sustentar uma alta escalabilidade; até certo ponto o uso de um hardware melhor pode ser adotado, mas além desse ponto, a RDB deve ser distribuída;

(2) o fato de dados serem armazenados em uma RDB como tabelas, e essas estruturas poderem provocar alta complexidade, impossibilita dados serem facilmente encapsulados em tabela;

(3) muitos atributos que participam de tabelas em RDB não são utilizados e, portanto, contribuem para o custo bem como para aumento da complexidade da RDB;

(4) RDBs fazem uso de expressões em SQL que funcionam bem quando dados estão estruturados, mas podem se tornar altamente elaboradas quando os dados não são estruturados;

(5) quando o volume de dados se torna imenso, a RDB tem que ser particionada entre múltiplos servidores. Essa partição pode provocar vários problemas uma vez que a composição de tabelas não é uma tarefa fácil.

2.4 Normalização de Base de Dados

A normalização de base de dados é um processo fundamental para garantir a organização, confiabilidade e eficiência do armazenamento de dados. Por meio da aplicação de regras e princípios específicos, a normalização visa eliminar redundâncias, inconsistências e anomalias nos dados, assegurando a integridade e a qualidade das informações armazenadas, essenciais para pesquisas e análises complexas. Vale ressaltar que uma base de dados normalizada até a terceira forma normal (3FN) já está suficientemente organizado e pronto para uso, proporcionando uma estrutura sólida e eficaz para manipulação de dados [Codd 1970].

Pode ser notado na Tabela 2.1 uma significativa repetição de informações na tabela 'Livros', o que evidencia a não normalização da tabela. Uma tabela não normalizada é aquela que contém redundâncias e duplicações de dados, o que pode levar a inconsistências e a um maior consumo de espaço de armazenamento [Date 2010]. Na Tabela 2.1, as linhas 1 e 2 informam o mesmo livro, porém com autores diferente, onde fica claro a duplicidade das informações, repetindo informações como o título do livro e editora.

2.4.1 Dependência Funcional

Dependência funcional é a relação em que um atributo ou um conjunto de atributos em uma tabela de base de dados depende de outro atributo (ou conjunto de atributos) [Elmasri *et al.* 2015].

Dados dois elementos X e Y, podemos dizer que o elemento Y é dependente funcionalmente do elemento X se para cada valor de X encontrado obtemos os mesmos valores de Y.

Simbolicamente: X → Y, lido como “X determina funcionalmente Y”.

Exemplo na Tabela 2.2: Sendo **CÓDIGO_LIVRO** uma **chave candidata**, [Codd 1970] define que chave candidata em bancos de dados é um conjunto de um ou mais atributos que podem identificar de maneira única um registro. Pode-se dizer que todos os outros campos são dependentes funcionalmente do desse atributo.

Tabela 2.2 Livros (Dependência Funcional CÓDIGO).

CÓDIGO_LIVRO	TÍTULO	CÓDIGO_AUTOR	AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR	ANO_PUB	PREÇO	CODIGO_EDITORA	EDITORA
CT123	Grafos	7, 1	T. Lima, A.Guarani	23/02/1972, 15/05/1970	PRINCIPAL, COAUTOR	1995	203,20	2	AZUL
RO487	Despedidas	3	D. Chapman	20/09/1968	PRINCIPAL	1998	197,80	4	VERDE
BI896	Abelhas	5	M. Delly	02/04/1964	PRINCIPAL	2021	180,00	2	AZUL
RO112	Lua Vermelha	3, 4	D. Chapman, E. Roland	20/09/1968, 18/08/1970	PRINCIPAL, COAUTOR	2023	276,00	3	ROXO
CT007	Cálculo	8	U.Wonder	14/04/1965	PRINCIPAL	1998	287,00	3	ROXO
EN369	Mecânica	6	N. Brasilis	19/12/1973	PRINCIPAL	2023	194,43	1	AMARELO
CT995	Álgebra	8, 1	U.Wonder, A.Guarani	14/04/1965, 15/05/1970	PRINCIPAL, COAUTOR	1999	255,00	4	VERDE
BI452	Mamíferos	5, 2	M. Delly, C. Marinho	02/04/1964, 02/07/1965	PRINCIPAL, COAUTOR	2023	302,12	2	AZUL

Pode ser observado na Tabela 2.3 que toda vez que CÓDIGO_AUTOR se repete, também são repetidos AUTOR e DT_NASC_AUTOR.

Tabela 2.3 Livros (Dependência Funcional CÓDIGO_AUTOR).

CÓDIGO_LIVRO	TÍTULO	CÓDIGO_AUTOR	AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR	ANO_PUB	PREÇO	CODIGO_EDITORA	EDITORA
CT123	Grafos	7, 1	T. Lima, A.Guarani	23/02/1972, 15/05/1970	PRINCIPAL, COAUTOR	1995	203,20	2	AZUL
RO487	Despedidas	3	D. Chapman	20/09/1968	PRINCIPAL	1998	197,80	4	VERDE
BI896	Abelhas	5	M. Delly	02/04/1964	PRINCIPAL	2021	180,00	2	AZUL
RO112	Lua Vermelha	3, 4	D. Chapman, E. Roland	20/09/1968, 18/08/1970	PRINCIPAL, COAUTOR	2023	276,00	3	ROXO
CT007	Cálculo	8	U.Wonder	14/04/1965	PRINCIPAL	1998	287,00	3	ROXO
EN369	Mecânica	6	N. Brasilis	19/12/1973	PRINCIPAL	2023	194,43	1	AMARELO
CT995	Álgebra	8, 1	U.Wonder, A.Guarani	14/04/1965, 15/05/1970	PRINCIPAL, COAUTOR	1999	255,00	4	VERDE
BI452	Mamíferos	5, 2	M. Delly, C. Marinho	02/04/1964, 02/07/1965	PRINCIPAL, COAUTOR	2023	302,12	2	AZUL

De maneira semelhante, pode ser observado na Tabela 2.4 que toda vez que CODIGO_EDITORA se repete, o conteúdo do atributo EDITORA é também repetido.

Tabela 2.4 Livros (Dependência Funcional CODIGO_EDITORA).

CÓDIGO_LIVRO	TÍTULO	CÓDIGO_AUTOR	AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR	ANO_PUB	PREÇO	CODIGO_EDITORA	EDITORA
CT123	Grafos	7, 1	T. Lima, A.Guarani	23/02/1972, 15/05/1970	PRINCIPAL, COAUTOR	1995	203,20	2	AZUL
RO487	Despedidas	3	D. Chapman	20/09/1968	PRINCIPAL	1998	197,80	4	VERDE
BI896	Abelhas	5	M. Delly	02/04/1964	PRINCIPAL	2021	180,00	2	AZUL
RO112	Lua Vermelha	3, 4	D. Chapman, E. Roland	20/09/1968, 18/08/1970	PRINCIPAL, COAUTOR	2023	276,00	3	ROXO
CT007	Cálculo	8	U.Wonder	14/04/1965	PRINCIPAL	1998	287,00	3	ROXO
EN369	Mecânica	6	N. Brasilis	19/12/1973	PRINCIPAL	2023	194,43	1	AMARELO
CT995	Álgebra	8, 1	U.Wonder, A.Guarani	14/04/1965, 15/05/1970	PRINCIPAL, COAUTOR	1999	255,00	4	VERDE
BI452	Mamíferos	5, 2	M. Delly, C. Marinho	02/04/1964, 02/07/1965	PRINCIPAL, COAUTOR	2023	302,12	2	AZUL

2.4.2 Atributos Multivalorados

A Tabela 2.5 mostra atributos cujos respectivos valores mudam em relação ao valor da chave candidata **CÓDIGO**, esses atributos são os multivalorados onde o armazenamento em um base de dados pode gerar redundância e dificultar a manipulação das informações [Brown 2011].

Tabela 2.5 Livros (Atributos Multivalorados).

CÓDIGO_LIVRO	TÍTULO	CÓDIGO_AUTOR	AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR	ANO_PUB	PREÇO	CODIGO_EDITORA	EDITORA
CT123	Grafos	7, 1	T. Lima, A.Guarani	23/02/1972, 15/05/1970	PRINCIPAL, COAUTOR	1995	203,20	2	AZUL
RO487	Despedidas	3	D. Chapman	20/09/1968	PRINCIPAL	1998	197,80	4	VERDE
BI896	Abelhas	5	M. Delly	02/04/1964	PRINCIPAL	2021	180,00	2	AZUL
RO112	Lua Vermelha	3, 4	D. Chapman, E.	20/09/1968, 18/08/1970	PRINCIPAL, COAUTOR	2023	276,00	3	ROXO
CT007	Cálculo	8	U.Wonder	14/04/1965	PRINCIPAL	1998	287,00	3	ROXO
EN369	Mecânica	6	N. Brasilis	19/12/1973	PRINCIPAL	2023	194,43	1	AMARELO
CT995	Álgebra	8, 1	U.Wonder, A.Guarani	14/04/1965, 15/05/1970	PRINCIPAL, COAUTOR	1999	255,00	4	VERDE
BI452	Mamíferos	5, 2	M. Delly, C. Marinho	02/04/1964, 02/07/1965	PRINCIPAL, COAUTOR	2023	302,12	2	AZUL

2.4.3 1ª. Forma Normal (1ªFN)

A 1ªFN garante que cada coluna em uma tabela armazena um único valor atômico. Isso significa que os atributos multivalorados de dados devem ser eliminados e normalizados em novas tabelas [Codd 1970]. Pode ser observado que a chave CÓDIGO (Tabela 2.6) passa a fazer parte da nova tabela Livro Autor (Tabela 2.7), participando de uma chave composta (CÓDIGO, CÓDIGO_AUTOR).

Tabela 2.6 Livro (1ªFN).

CÓDIGO	TÍTULO	ANO_PUB	PREÇO	CODIGO_EDITORA	NOME_EDITORA
CT123	Grafos	1995	203,20	2	AZUL
RO487	Despedidas	1998	197,80	4	VERDE
BI896	Abelhas	2021	180,00	2	AZUL
RO112	Lua Vermelha	2023	276,00	3	ROXO
CT007	Cálculo	1988	287,00	3	ROXO
EN369	Mecânica	2023	194,43	1	AMARELO
CT995	Álgebra	1999	255,00	4	VERDE
BI452	Mamíferos	2023	302,12	2	AZUL

Tabela 2.7 LivroAutor (1ªFN).

CÓDIGO	CÓDIGO_AUTOR	NOME_AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR
CT123	**7**	T. Lima	26352	PRINCIPAL
CT123	**1**	A.Guarani	25703	COAUTOR
RO487	**3**	D. Chapman	25101	PRINCIPAL
BI896	**5**	M. Delly	02/04/1964	PRINCIPAL
RO112	**3**	D. Chapman	20/09/1968	PRINCIPAL
RO112	**4**	E. Roland	18/08/1970	COAUTOR
CT007	**8**	U.Wonder	14/04/1965	PRINCIPAL
EN369	**6**	N. Brasilis	19/12/1973	PRINCIPAL
CT995	**8**	U.Wonder	14/04/1965	PRINCIPAL
CT995	**1**	A.Guarani	15/05/1970	COAUTOR
BI452	**5**	M. Delly	02/04/1964	PRINCIPAL
BI452	**2**	C. Marinho	02/07/1965	COAUTOR
BI452	**3**	D. Chapman	20/09/1968	COAUTOR

2.4.4 Dependência Parcial *versus* Dependência Total

A dependência parcial e a dependência total são conceitos importantes na teoria de bancos de dados relacionais. A dependência total ocorre quando um atributo depende de toda a chave composta, a dependência parcial ocorre quando um atributo depende apenas de parte da chave composta.

É observado na Figura 2.1 a chave composta definida pelos atributos A e B. Os atributos que dependem de uma parte da chave, como o caso do atributo D, são caracterizados como parcialmente dependentes. O atributo restante C, é completamente dependente da chave composta.

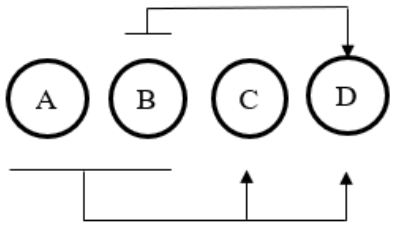

Figura 2.1 Dependência Total e Dependência Parcial.

2.4.5 2ª Forma Normal (2ªFN)

A 2ªFN elimina toda dependência parcial e cria uma nova tabela, representada na Figura 2.2. Sendo assim, uma relação está na 2ªFN, se ela estiver na 1ªFN e nenhum de seus atributos possui dependência parcial [Codd 1970].

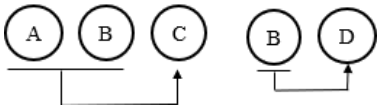

Figura 2.2 Dependência Total.

Observa-se na Tabela 2.9 que os atributos **NOME_AUTOR** e **DT_NASC_AUTOR** dependem parcialmente da chave composta (**CÓDIGO, CÓDIGO_AUTOR**), e o atributo PAPEL_AUTOR depende totalmente da chave.

Tabela 2.8 LivroAutor – Dependência Total *versus* Dependência Parcial.

CÓDIGO	CÓDIGO_AUTOR	NOME_AUTOR	DT_NASC_AUTOR	PAPEL_AUTOR
CT123	7	T. Lima	23/02/1972	PRINCIPAL
CT123	1	A.Guarani	15/05/1970	COAUTOR
RO487	3	D. Chapman	20/09/1968	PRINCIPAL
BI896	5	M. Delly	02/04/1964	PRINCIPAL
RO112	3	D. Chapman	20/09/1968	PRINCIPAL
RO112	4	E. Roland	18/08/1970	COAUTOR
CT007	8	U.Wonder	14/04/1965	PRINCIPAL
EN369	6	N. Brasilis	19/12/1973	PRINCIPAL
CT995	8	U.Wonder	14/04/1965	PRINCIPAL
CT995	1	A.Guarani	15/05/1970	COAUTOR
BI452	5	M. Delly	02/04/1964	PRINCIPAL
BI452	2	C. Marinho	02/07/1965	COAUTOR
BI452	5	M. Delly	02/04/1964	COAUTOR

Observa-se na Tabela 2.9 que foi eliminado a dependência parcial da tabela **Livro Autor**, no qual criou-se a tabela **Autor** evidenciado na Tabela 2.10, deixando assim as duas tabelas na 2ªFN.

Tabela 2.6 Livro (1ªFN).

CÓDIGO	TÍTULO	ANO_PUB	PREÇO	CODIGO_EDITORA	NOME_EDITORA
CT123	Grafos	1995	203,20	2	AZUL
RO487	Despedidas	1998	197,80	4	VERDE
BI896	Abelhas	2021	180,00	2	AZUL
RO112	Lua Vermelha	2023	276,00	3	ROXO
CT007	Cálculo	1988	287,00	3	ROXO
EN369	Mecânica	2023	194,43	1	AMARELO
CT995	Álgebra	1999	255,00	4	VERDE
BI452	Mamíferos	2023	302,12	2	AZUL

Tabela 2.9 LivroAutor (2ªFN).

CÓDIGO	CÓDIGO_AUTOR	PAPEL_AUTOR
CT123	7	PRINCIPAL
CT123	1	COAUTOR
RO487	3	PRINCIPAL
BI896	5	PRINCIPAL
RO112	3	PRINCIPAL
RO112	4	COAUTOR
CT007	8	PRINCIPAL
EN369	6	PRINCIPAL
CT995	8	PRINCIPAL
CT995	1	COAUTOR
BI452	5	PRINCIPAL
BI452	2	COAUTOR
BI452	5	COAUTOR

Tabela 2.10 Autor(2ªFN).

CÓDIGO_AUTOR	NOME_AUTOR	DT_NASC_AUTOR
1	A.Guarani	15/05/1970
2	C. Marinho	02/07/1965
3	D. Chapman	20/09/1968
4	E. Roland	18/08/1970
5	M. Delly	02/04/1964
6	N. Brasilis	19/12/1973
7	T. Lima	23/02/1972
8	U.Wonder	14/04/1965

2.4.6 Dependência Transitiva

A Dependência Transitiva ocorre quando um determinado atributo possuir dependência de outro que não pertence à chave primária [Elmasri *et al.* 2015], situação que pode ser evidenciada na Figura 2.3, na qual o atributo D depende do atributo C, que não faz parte da chave.

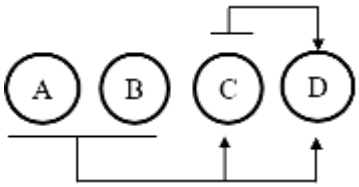

Figura 2.3 Dependência Transitiva.

2.4.7 3ª. Forma Normal (3ª.FN)

Uma relação está na 3ªFN se ela estiver na 2ªFN e suas dependências transitivas tiverem sido eliminadas, garantindo assim que nenhuma coluna não chave dependa de outra coluna não chave [Codd 1970], como pode ser observado na Figura 2.4.

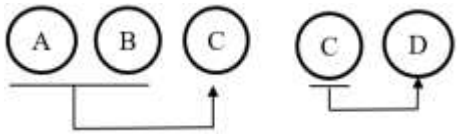

Figura 2.4 Eliminado Dependência Transitiva.

Observa-se na Tabela 2.11 que existe uma dependência transitiva do atributo **EDITORA** com o atributo **CÓDIGO_EDITORA** na tabela **Livro**. Para que a 3ªFN seja obtida, é necessário eliminar a dependência e criar uma nova tabela.

Tabela 2.11 Livro(2ª.FN) – Dependência Transitiva.

CÓDIGO	TÍTULO	ANO_ PUB	PREÇO	CODIGO_ EDITORA	EDITORA
CT123	Grafos	1995	203,20	**2**	AZUL
RO487	Despedidas	1998	197,80	**4**	VERDE
BI896	Abelhas	2021	180,00	**2**	AZUL
RO112	Lua Vermelha	1923	276,00	**3**	ROXO
CT007	Cálculo	1988	287,00	**3**	ROXO
EN369	Mecânica	2023	194,43	**1**	AMARELO
CT995	Álgebra	1999	255,00	**4**	VERDE
BI452	Mamíferos	2023	302,12	**2**	AZUL

Como pode ser observado na Tabela 2.12, o atributo EDITORA foi eliminado da tabela Livro (como mostra a Tabela 2.14), e foi criada a tabela EDITORA mostrada na Tabela 2.13.

Tabela 2.12 Livro (3ªFN).

CÓDIGO	TÍTULO	ANO_PUB	PREÇO	CODIGO_ EDITORA
CT123	Grafos	1995	203,20	2
RO487	Despedidas	1998	197,80	4
BI896	Abelhas	2021	180,00	2
RO112	Lua Vermelha	1923	276,00	3
CT007	Cálculo	1988	287,00	3
EN369	Mecânica	2023	194,43	1
CT995	Álgebra	1999	255,00	4
BI452	Mamíferos	2023	302,12	2

Tabela 2.13 Editora (3ªFN).

CODIGO_ EDITORA	EDITORA
1	AMARELO
2	AZUL
3	ROXO
4	VERDE

Tabela 2.9 LivroAutor (3ªFN).

CÓDIGO	CÓDIGO_ AUTOR	PAPEL_AU TOR
CT123	**7**	PRINCIPAL
CT123	**1**	COAUTOR
RO487	**3**	PRINCIPAL
BI896	**5**	PRINCIPAL
RO112	**3**	PRINCIPAL
RO112	**4**	COAUTOR
CT007	**8**	PRINCIPAL
EN369	**6**	PRINCIPAL
CT995	**8**	PRINCIPAL
CT995	**1**	COAUTOR
BI452	**5**	PRINCIPAL
BI452	**2**	COAUTOR
BI452	**5**	COAUTOR

Tabela 2.10 Autor(3ªFN).

CÓDIGO_AUTOR	NOME_AUTOR	DT_NASC_AUTOR
1	A.Guarani	15/05/1970
2	C. Marinho	02/07/1965
3	D. Chapman	20/09/1968
4	E. Roland	18/08/1970
5	M. Delly	02/04/1964
6	N. Brasilis	19/12/1973
7	T. Lima	23/02/1972
8	U.Wonder	14/04/1965

2.4.8 Outras Formas Normais

(1) a **Forma Normal de Boyce-Codd** (BCNF) foi proposta em [Codd *et al.* 1974] e representa um estágio crucial na normalização de RDBs, por meio do refinamento a Terceira Forma Normal (3ªFN). Ela elimina redundâncias e anomalias de atualização, garantindo a integridade, confiabilidade e eficiência de RDBs. Por meio da decomposição de tabelas complexas, a BCNF otimiza o uso de armazenamento, previne inconsistências e acelera consultas, o que a torna essencial para sistemas de informação robustos. Ao impor restrições mais rigorosas do que as impostas pela 3ªFN, a BCNF elimina as dependências multivaloradas, que causam redundâncias e anomalias.

(2) **4ª Forma Normal (4ªFN):** A 4ªFN, introduzida por [Fagin 1977], representa um estágio avançado na normalização de bases de dados relacionais, focando na eliminação de dependências multivaloradas. Essa forma normal se aplica a situações em que uma coluna não chave depende de partes distintas da chave primária, levando a redundâncias e anomalias de atualização. Para implementar a 4ªFN, a tabela original é dividida em duas ou mais tabelas, de forma que cada coluna não chave dependa apenas de uma única coluna da chave primária. Essa decomposição garante a independência das informações e elimina redundâncias, resultando em um modelo de dados mais eficiente e confiável. Embora a implementação da 4ªFN possa ser complexa e nem sempre necessária, ela é recomendada para casos específicos em que a integridade e a consistência dos dados são cruciais.

(3) 5ª Forma Normal (5ªFN): A 5ªFN proposta em [Fagin 1981], representa o ápice da normalização em RDBs. Ela visa eliminar redundâncias e anomalias remanescentes nas formas normais anteriores, aprimorando a qualidade, confiabilidade e eficiência da base de dados. Ao contrário das formas normais tradicionais que se baseiam em chaves e dependências funcionais, a 5ªFN utiliza domínios e dependências de junção. Mais especificamente, uma dependência de junção ocorre quando uma relação (ou tabela) pode ser decomposta em duas ou mais sub-relações de tal forma que a junção dessas sub-relações (usando uma operação de *JOIN*) reproduz exatamente a relação original, sem perda de informações ou introdução de redundâncias. Essa abordagem permite a decomposição de relações em esquemas irredutíveis, minimizando redundâncias e anomalias. Embora a implementação da 5ªFN possa ser complexa, seus benefícios são significativos. A 5ªFN contribui para a criação de bases de dados mais robustas, confiáveis e eficientes, especialmente em sistemas de informação críticos. Sua aplicação garante a integridade dos dados, facilitando consultas e operações, e otimizando o desempenho da base de dados.

2.5 Álgebra Relacional e SQL

A Álgebra Relacional, definida como uma linguagem formal e procedural para manipulação de dados em RDBs [Date 2010], fornece um conjunto de operações matemáticas que permitem aos usuários definir, de forma precisa e inequívoca, as informações que desejam manipular da base de dados.

De maneira semelhante, a linguagem SQL (*Structured Query Language*) é eficiente e amplamente utilizada em interações com bases de dados relacionais [Date 2010], permitindo aos usuários criar, ler, atualizar e excluir dados armazenados nessas bases de dados, o que a torna uma ferramenta essencial para gerenciamento e análise de informações. Essas duas ferramentas, a Álgebra Relacional e a SQL, formam a base para a manipulação eficiente e precisa de dados em sistemas de gerenciamento de base de dados relacionais. Entre as principais operações derivadas, conforme informado em [Abraham *et al.* 2021], se destacam:

(1) **Seleção (σ)**: Seleciona tuplas de uma relação com base em um predicado (expressão lógica) definido pelo usuário. A seleção filtra tuplas da relação, retornando apenas aquelas que atendem à condição especificada.

Exemplo: Selecionar todos os livros com ano de publicação superior a 2001.

(a) Usando Álgebra Relacional :

σ ano_pub > 201 (Livro)

(b) Usando SQL:

```
SELECT * FROM Livro
WHERE ano_pub > 2001;
```

(2) **Projeção (π):** Projeta um conjunto de atributos de uma relação, descartando os demais. A projeção permite focar em atributos específicos de interesse, ignorando os atributos irrelevantes para a consulta.

Exemplo: Obter o título, número de páginas e ano de publicação de cada livro:

(a) Usando Álgebra Relacional :

π titulo, ano_pub (Livro)

(b) Usando SQL:

```
SELECT titulo, ano_pub FROM Livro;
```

(3) **Junção (⋈)**: A junção na álgebra relacional combina tuplas de diferentes tabelas com base em valores comuns em atributos específicos, permitindo a integração de dados e a obtenção de informações mais abrangentes.

Exemplo: Obter a relação entre Livros e Editora:

(a) Usando Álgebra Relacional:

πLivro.titulo, Editora.editora
(Livro⋈Livro.codigo_editora=Editora.codigo_editoraEditora)

(b) Usando SQL:

```
SELECT Livro.titulo, Editora.editora
FROM Livro JOIN Editora ON Livro.codigo_editora = Editora.codigo_editora;
```

2.6 Tipos de Relacionamento

No contexto de RDBs, os relacionamentos servem como pontes que conectam diferentes tabelas, organizando os dados de forma eficiente. Compreender os diversos tipos de relacionamentos é crucial para projetar uma base de dados robusta e gerenciar informações com precisão [Teorey 2014].

(1) Relacionamento Um para Muitos (1:N): Aqui, uma instância em uma tabela está relacionada a várias instâncias em outra tabela. Um exemplo clássico é a tabela Editora com um campo "CODIGO_EDITORA" e a tabela Livro, em que cada editora pode ter vários livros, e cada livro foi impresso por uma única.

(2) Relacionamento Muitos para Muitos (N:M): Neste caso, várias instâncias de uma tabela estão associadas a várias instâncias de outra tabela. Um exemplo comum é a tabela Livro com um campo "CODIGO_LIVRO" e a tabela Autor, onde cada livro pode ter vários autores, e cada autor pode ter vários livros.

2.7 Tipos de Dados Primitivos

Os tipos de dados primitivos em bases de dados podem variar de acordo com o sistema de gerenciamento de bases de dados (RBDS) utilizado, Esses tipos de dados proporcionam a base para a representação de informações em uma base de dados, permitindo a manipulação eficiente e organizada dos dados armazenados [Elmasri *et al.* 2015]. Seguem alguns tipos de dados comuns que são suportados pela maioria dos RDBMS:

(1) *Integer* (Inteiro): Representa números inteiros, geralmente com tamanho fixo, como INT, INTEGER, SMALLINT, BIGINT, entre outros.

(2) *Decimal*/*Float* (Decimal/Ponto Flutuante): Representa números decimais ou de ponto flutuante, geralmente com precisão fixa ou variável, como DECIMAL, FLOAT, REAL, DOUBLE, entre outros.

(3) *Character/String* (Caractere/Cadeia de Caracteres): Representa sequências de caracteres alfanuméricos, como *CHAR*, *VARCHAR*, *TEXT*, entre outros.

(4) *Boolean (Booleano):* Representa valores lógicos verdadeiro ou falso, como BOOLEAN.

(5) *Date/Time (Data/Hora):* Representa datas ou horas, como *DATE, TIME, DATETIME, TIMESTAMP*, entre outros.

(6) *Binary (Binário):* Representa dados binários, como *BLOB (Binary Large Object), BYTE, VARBINARY*, entre outros.

2.8 Modelo de Dados

A modelagem de dados, como definida por Brown em [Brown 2011], é um processo fundamental para projetar e estruturar bancos de dados relacionais. Ela consiste na criação de uma representação abstrata dos dados que serão armazenados no base de dados, definindo as tabelas, seus atributos e suas relações entre si.

2.8.1 Modelo Conceitual de DB

O modelo conceitual de base de dados, representado na Figura 2.5, é uma representação abstrata e de alto nível que descreve as tabelas, atributos e relacionamentos essenciais do sistema sem se preocupar com detalhes técnicos de implementação. Ele serve como uma ferramenta de comunicação entre os stakeholders e é utilizado para capturar os requisitos de dados de forma clara e compreensível. Este modelo é frequentemente representado por diagramas de Tabela-Relacionamento (ER), onde são identificadas as principais tabelas e os relacionamentos entre elas. De acordo com [Elmasri *et al*. 2015], o modelo conceitual é crucial para garantir que todos os envolvidos tenham uma compreensão comum do que o sistema deve suportar.

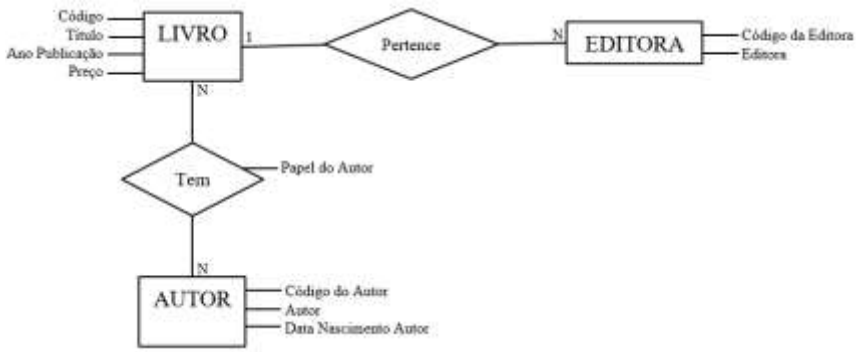

Figura 2.5 Modelo Conceitual do Processo de Controle de Livros.

2.8.2 Modelo Lógico

O modelo lógico de base de dados, representado na Figura 2.6, é uma tradução mais detalhada do modelo conceitual, estruturado de acordo com as regras de um base de

dados relacional, mas ainda independente de um Sistema de Gerenciamento de Base de dados (SGBD) específico. Ele inclui detalhes técnicos como tipos de dados, chaves primárias e estrangeiras, e segue os princípios de normalização para eliminar redundâncias e garantir a integridade dos dados. Este modelo serve como uma ponte entre o design conceitual e a implementação física. Segundo [Date 2010], o modelo lógico é fundamental para preparar a base para a implementação do base de dados, assegurando que a estrutura lógica atenda aos requisitos funcionais e não funcionais do sistema.

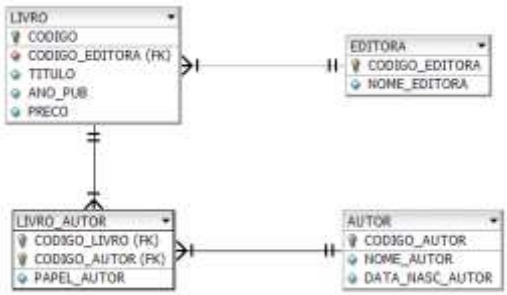

Figura 2.6 Modelo Lógico do Processo de Controle de Livros.

2.8.3 Modelo Físico

O modelo físico de base de dados é a representação detalhada de como os dados serão armazenados no sistema de gerenciamento de base de dados (SGBD) específico. Ele traduz o modelo lógico em um esquema físico, especificando tabelas, colunas, tipos de dados e outros detalhes técnicos de implementação. Este modelo é crítico para a performance, segurança e integridade dos dados no ambiente de produção. Segundo [Elmasri *et al*. 2015], o modelo físico deve considerar fatores como otimização de consultas e eficiência de armazenamento para assegurar que o sistema atenda aos requisitos de desempenho e escalabilidade.

Após a análise detalhada dos conceitos de normalização, relacionamento e tipos de dados primitivos pertinentes à definição de bases de dados, tais fundamentos estão consolidados na construção do modelo físico, tal como manifestado na representação visual da Diagrama de Tabela-Relacionamento (DER), especificamente identificada como a Figura 2.7 ilustra o diagrama de tabela e relacionamento (DER) da tabela **Livros** normalizada na Terceira Forma Normal (3ª FN).

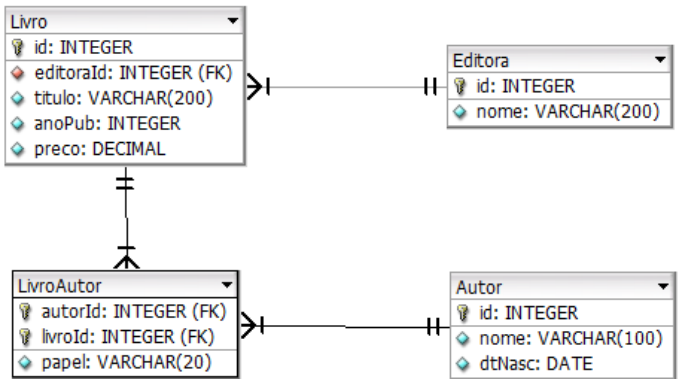

Figura 2.7 Modelo Físico Normalizado na 3ªFN envolvendo as 4 tabelas (Livro, Editora, Autor, LivroAutor).

A Linguagem de Definição de Dados (DDL) é parte fundamental da SQL, e é usada para criar, modificar e remover objetos em uma base de dados [Silberschatz *et al.* 2021]. Seus comandos principais, tais como CREATE, ALTER e DROP, permitem a criação de tabelas, índices e outros objetos, e a DDL também pode ser usada na manipulação de estruturas e restrições [Elmasri *et al.* 2015]. Por meio da DDL, a integridade e a consistência dos dados são garantidas, o que estabelece a robustez e confiabilidade da base de dados [Date 2010].

2.8.4 Notações Utilizadas

A notação clássica para Diagrama de Tabela e Relacionamento (DER) [Chen 1976] é a mais utilizada e compreensível para representar os elementos de um modelo de dados conceitual. Essa notação utiliza símbolos específicos para representar:

Retângulos: Representam as tabelas do modelo, que são os objetos principais sobre os quais se armazenam dados. Cada tabela possui um nome singular maiúsculo no interior do retângulo.

Linha simples: Indica um relacionamento entre duas tabelas, onde a chave estrangeira será parte da chave da primária.

Linha tracejada: Indica um relacionamento entre duas tabelas, onde a chave estrangeira não fará parte da chave primária.

Dois traços paralelos na vertical: Indica um relacionamento para 1 (um)

Pé de corvo: Indica um relacionamento para N (muitos).

Chave: Indica a chave primária

FK: Indica Chave Estrangeira

Losango: Indica um atributo

Capítulo 3
Mapeamento Objeto-Relational (ORM)

3.1 Sistemas OLTP e OLAP

Os autores em [Kojic *et al.* 2015] abordam sistemas computacionais divididos em duas grandes classes: aqueles caracterizados como transacional (*Online Transaction Processing* – OLTP) e os caracterizados como analíticos (*Online Analytical Processing* – OLAP).

Sistemas da classe OLTP têm como foco o processamento em tempo real de transações online e têm como principais características: (a) o armazenamento de dados existentes e ativos, (b) transações on-line curtas e intensivas, (c) processamento rápido das consultas e (d) manutenção rígida de consistência de dados em um ambiente concorrente. Devido às suas características, sistemas OLTP são usados para a execução online de muitas tarefas, tais como: transações geradas por atendentes em caixas bancários físicos ou operações realizadas em máquinas ATMs, aplicações de autoatendimento, tais como online banking, compra de passagens e e-commerce que, também geram transações em DBs. Sistemas OLTP podem, pois, ser uma fonte de dados para sistemas OLAP.

Sistemas da classe OLAP via de regra produzem e armazenam dados persistentes em RDBs, particularmente devido ao fato que tais DBs têm grande credibilidade e adquiriram, ao longo do tempo, confiabilidade junto ao mercado. A otimização de desempenho e o tratamento eficiente de dados estão fortemente acoplados ao modelo de dados oferecido em RDBs. Um sistema OLAP, por exemplo, pode ser usado para analisar dados relacionados a um determinado setor do mercado, sob diferentes perspectivas.

3.2 A Gênesis do Mapeamento Objeto-Relacional

A estrutura nomeada como Mapeamento Objeto-Relacional (ORM) teve origem no contexto de duas tendências significativas na área de desenvolvimento de software: programação orientada a objetos e base de dados relacional.

À medida que o custo do hardware comum começou a declinar a partir de meados da década de 1980, a indústria começou a abandonar os computadores conhecidos como *mainframes* e passou a adotar a implantação em massa de aplicativos GUI (*Graphical User Interface*), termo utilizado para descrever a interface gráfica utilizada por um software ou aplicativo, que permite que usuários possam interagir com um sistema computacional via computador, *tablet*, celular ou qualquer outro dispositivo eletrônico baseado em um modelo cliente-servidor.

O mundo dos negócios à época, entretanto, ainda dependia de *mainframes* e das bases de dados relacionais que surgiram nas décadas de 1970 e 1980, capazes de armazenar enormes volumes de informações persistentes e que já tinham alcançado estabilidade, bem como angariado um alto grau de confiabilidade por parte da comunidade de usuários.

Como Rickerby comenta em [Rickerby 2015], um dos principais objetivos desta classe emergente de "aplicações empresariais" era o de se conectar a essas bases de dados, liberando o potencial do software para automatizar fluxos de trabalho tediosos e, com isso, permitir que utilizadores não técnicos gerenciassem e manipulassem dados. Associada a esta transição, a programação orientada a objetos se estabeleceu pesadamente tanto na indústria quanto em pesquisas acadêmicas.

À medida que os ideais altamente divulgados da programação orientada a objetos foram entrando em conflito com a bem estabelecida e comprovada tecnologia de base de dados relacional, um problema, caracterizado como "*incompatibilidade da impedância objeto-relacional*", antes inexistente, emergiu e se tornou cada vez mais presente. O *conceito de impedância*, importado da área de engenharia elétrica, foi usado para caracterizar incompatibilidades quando códigos orientados a objetos trocam dados com uma base de dados relacional, problema que foi tratado com a introdução de uma camada de mapeamento entre programação orientada a objetos e base de dados relacional.

Teixeira, em [Teixeira 2017], informa que a opção mais usada para a redução do atrito entre modelos relacionais e código orientado a objetos é aquela feita por meio do uso de *frameworks* j Alguns desses *frameworks* são abordados nas seções 3.5 e 3.6.

3.3 Caracterização do Mapeamento Objeto-Relacional

No cenário atual de desenvolvimento de software, o uso de sistemas de gerenciamento de base de dados relacional para persistir objetos de um modelo de domínio orientado a objetos é uma abordagem comum, que é adotada por um vasto número de organizações.

Como informado em [Lorenz *et al.*, 2016] e brevemente comentado anteriormente, o mapeamento objeto-relacional (ORM) emula um mecanismo para preencher a lacuna semântica entre linguagens de programação orientada a objetos (OOP) e RDBMS.

No contexto de muitos dos ambientes organizacionais de desenvolvimento de software atuais, as chamadas estruturas de Mapeamento Objeto-Relacional (Object-Relational Mapping-ORM) estão sendo frequentemente projetadas, desenvolvidas e disponibilizadas, com o objetivo de facilitar a acomodação de programação baseada em objetos a um ambiente computacional que faz uso de base de dados relacional (RDB).

Devido aos conceitos fundamentalmente diferentes empregados por ambos, a saber, álgebra relacional (empregada em DBRs) e o modelo orientado a objetos incorporado às linguagens de programação (OOP), um conjunto de desafios afloram quando do mapeamento de objetos para tabelas (relações).

Apesar das diferenças entre as estruturas existentes serem muitas, todas elas, de certa forma, traduzem a mesma ideia, que é a de fornecer um mapeamento entre a camada de objetos (classe object) e relações (tabelas da BDR), dois conceitos que diferem substancialmente entre si. Objetos são temporários enquanto dados no modelo relacional são persistentes. Objetos não obedecem a um esquema restrito, enquanto a SQL de BDR segue um esquema restrito.

O autor Barnes em [Barnes 2007] investiga o uso do Mapeamento Objeto-Relacional para persistência em aplicações orientadas a objetos. O autor identifica desafios significativos, como aquele caracterizado e brevemente definido como

impedance mismatch entre os paradigmas orientado a objetos e relacional, que incluem (a) dificuldades em mapear herança, (b) polimorfismo, (c) tipos de dados, e (d) associações complexas entre objetos. Questões de transparência também são críticas, pois muitos sistemas ORM não conseguem fornecer uma integração completamente invisível para os desenvolvedores, o que limita sua adoção ampla. Barnes especula que o futuro do ORM pode envolver melhorias na transparência e na automatização do mapeamento, além de uma maior integração com *frameworks* e linguagens de programação modernas, o que pode finalmente levar a uma adoção mais generalizada dessa tecnologia.

Como comentado anteriormente o problema de *impedance mismatch* é um dos desafios centrais abordados pelas soluções ORM que surge devido às diferenças fundamentais entre a forma como os dados são representados em bases de dados relacionais (tabelas e colunas) e em linguagens de programação orientadas a objetos (objetos e classes) [Teixeira 2017]. As soluções de ORM fornecem ferramentas para diluir este problema, mas não eliminam completamente a complexidade envolvida. Um dos principais desafios no ORM é a *impedance mismatch*, que ocorre devido às diferenças fundamentais entre os paradigmas orientado a objetos e relacional. Em [Ambler 1997] o autor reforça que essa discrepância surge porque o paradigma orientado a objetos se baseia em objetos com comportamento e dados, enquanto o paradigma relacional se baseia em dados armazenados em tabelas. Navegar entre objetos por suas relações contrasta com a junção de linhas em tabelas no modelo relacional. Outro desafio é a utilização de identificadores de objetos (OIDs), essenciais para distinguir objetos em uma base de dados relacional. O autor ressalta que os OIDs simplificam a estratégia de chaves na base de dados, facilitando a manutenção de relacionamentos entre objetos.

Apesar das vantagens oferecidas pelas ferramentas de ORM, existem desafios e limitações inerentes. Um dos principais desafios é a necessidade de adaptação dos modelos de análise para lidar com as peculiaridades de diferentes soluções que empregam ORM. Além disso, a suposição de que essas ferramentas podem resolver automaticamente todos os problemas de persistência pode levar a mal-entendidos e implementações ineficazes, com discutido em [Ogheneovo *et al*. 2013].

3.4 Foco em Padrões e Práticas de Design

O uso de padrões de design é crucial para a implementação eficaz de ORM. Os autores em [Torres *et al.* 2017] identificam e discutem diversos padrões arquiteturais e estruturais utilizados por diferentes soluções de ORM. Apesar das diferenças terminológicas, muitas dessas soluções compartilham padrões comuns que influenciam diretamente a forma como os modelos de classe devem ser adaptados para o mapeamento prático à base de dados.

3.5 Foco em Comparações e Avaliações de Desempenho

Durante o levantamento bibliográfico conduzido e identificado como LB2 (ver Seção 4), diversos *frameworks* ORM que são amplamente utilizados na indústria de software foram identificados, tais como Hibernate, JPA (Java Persistence API) e Spring Data JPA. Essas ferramentas não apenas facilitam o mapeamento de classes Java para tabelas da base de dados, como também oferecem funcionalidades avançadas, tais como consultas e recuperação de dados, o que pode reduzir significativamente o tempo de desenvolvimento.

Em [Tudose *et al.* 2021] os autores destacam o uso do Java Persistence API (JPA) e do Hibernate como ferramentas populares para implementar ORM em aplicações Java. Segundo os autores, estas ferramentas abstraem grande parte da complexidade do mapeamento objeto-relacional, fornecendo uma interface simples para operações de persistência.

Em [Ghandeharizadeh *et al.* 2014] os autores avaliam o desempenho do Hibernate comparando-o com o desempenho de uma implementação nativa usando JDBC (Java Database Connectivity). A pesquisa realizada evidencia que o Hibernate tende a emitir mais consultas SQL do que o JDBC, resultando em tempos de resposta mais lentos. No entanto, os autores sugerem que o uso da Hibernate Query Language (HQL) pode otimizar o desempenho do Hibernate, aproximando-o ao do JDBC.

Em [Kisman *et al.* 2016] os autores propõem a extensão Hibernate Criteria Extension (HCE) como uma solução para simplificar a criação de consultas complexas quando do uso do Hibernate. O HCE oferece uma API que permite definir associações de consultas utilizando uma nova notação que simplifica a configuração de junções e

restrições. A principal vantagem do HCE é a da redução da complexidade na elaboração de consultas elaboradas, especialmente quando existe a necessidade de lidar com projeções e associações de tabelas, algo que o Hibernate tradicional tem dificuldades.

Em [Keller 1997] o autor aborda a aplicação do Hibernate em um contexto específico, destacando as melhores práticas para otimização de desempenho e gerenciamento eficiente de transações. O autor enfatiza a importância de utilizar técnicas como *caching* e *lazy loading* para melhorar a eficiência e a escalabilidade das aplicações que utilizam o Hibernate.

No artigo [Lorenz *et al.* 2016], os autores revisitam os conceitos fundamentais do Mapeamento Objeto-Relacional, com um enfoque detalhado em como as tecnologias atuais abordam os desafios tradicionais de mapeamento de objetos para tabelas relacionais. Os autores discutem as melhorias de desempenho e a capacidade de escalabilidade das ferramentas ORM modernas, que se adaptam às necessidades de grandes sistemas distribuídos e de aplicações Web dinâmicas. Além disso, abordam técnicas avançadas de *caching*, gerenciamento de transações e otimização de consultas SQL geradas automaticamente, o que permite evidenciar uma visão crítica das práticas e inovações recentes na área de Mapeamento Objeto-Relacional.

3.6 Ferramentas ORM para Linguagem Java

No universo do desenvolvimento de software em Java, a persistência de dados se configura como um aspecto crucial para a construção de aplicações robustas e escaláveis que envolvem base de dados relacionais. Nesse contexto, dois *frameworks* se destacam como ferramentas essenciais para gerenciar a interação entre objetos Java e bancos de dados relacionais: Hibernate e Spring Data JPA, que são abordados respectivamente nas duas subseções que seguem.

3.6.1 Hibernate: Relacionamento Objeto-Relacional de Alto Desempenho

O Hibernate, diagramado na Figura 3.1, se apresenta como um *framework* que implementa um Relacionamento Objeto-Relacional (ORM) de código aberto, que é

caracterizado na literatura como robusto e de alto desempenho [Hibernate ORM 2024] [Bauer *et al*. 2005].

Sua principal função está relacionada ao mapeamento entre modelos de domínios orientados a objetos descritos em linguagens orientadas a objetos (nesta pesquisa, a linguagem Java) e tabelas de base de dados relacional, automatizando assim a tarefa de composição de código SQL explícito. Por meio de anotações e configurações, o Hibernate se encarrega de traduzir as propriedades e relações entre os objetos descritos em Java em instruções SQL adequadas, otimizando assim o acesso aos dados.

A abstração disponibilizada pelo Hibernate simplifica significativamente o desenvolvimento, a manutenção e a portabilidade de aplicações, uma vez que o código se torna menos dependente das particularidades de cada banco de dados relacional considerado.

Figura 3.1 – Hibernate *Framework*.

O Hibernate oferece uma série de recursos que o tornam uma ferramenta de interesse para ser utilizada em ambientes computacionais voltados ao desenvolvimento de código escrito em Java. Dentre os mais relevantes estão:

(1) Mapeamento Objeto-Relacional: O Hibernate mapeia classes Java para tabelas de base de dados relacional, atributos de classe para colunas de tabela e relacionamentos entre objetos e chaves estrangeiras, proporcionando uma representação transparente dos dados no código Java.

(2) Linguagem de Consulta HQL: A chamada Hibernate Query Language (HQL) é uma linguagem de consulta orientada a objetos que permite realizar consultas complexas na base de dados relacional, de forma intuitiva e independente do banco de dados relacional considerado.

(3) Gerenciamento de Transações: O Hibernate oferece suporte a transações, garantindo a consistência dos dados em operações que envolvem múltiplas alterações na base de dados relacional considerado.

(4) Cache de Segundo Nível: O cache de segundo nível do Hibernate armazena dados em memória, reduzindo assim o número de acessos a base de dados relacional sendo considerado e, com isso, como consequência, melhora o desempenho da aplicação.

(5) Suporte à Herança: O Hibernate oferece diferentes estratégias para mapear hierarquias de classes Java para o modelo relacional, permitindo representar relacionamentos de herança de forma flexível.

Dentre as principais vantagens obtidas com o uso do Hibernate estão:

(1) Redução do volume de código SQL: O Hibernate elimina a necessidade de desenvolvedores terem que compor manualmente código SQL, o que promove e simplifica o desenvolvimento e aumenta a legibilidade do código.

(2) Gerenciamento de transações robusto: O Hibernate oferece recursos avançados para gerenciamento de transações, garantindo a integridade dos dados em cenários complexos.

(3) Suporte a diversas bases de dados: O Hibernate é compatível com uma ampla variedade de bases de dados relacionais, proporcionando flexibilidade e portabilidade.

(4) Cache de consultas: O Hibernate implementa um mecanismo de cache eficiente para consultas SQL, que otimiza o desempenho e reduz o tempo de acesso aos dados.

Resumindo, o *framework* Hibernate se qualifica como uma ferramenta ORM fundamental em ambientes de programação Java com interações com base de dados relacional. O Hibernate oferece recursos poderosos bem como benefícios tangíveis para o desenvolvimento de aplicações que interagem com base de dados relacional. As capacidades de abstrair a complexidade do acesso a dados, de aumentar a produtividade, de melhorar a manutenabilidade e de garantir a portabilidade tornam o Hibernate uma escolha estratégica para projetos desenvolvidos em Java de diversos portes e complexidades.

3.6.2 Spring Data JPA: Abstraindo a Complexidade

O Spring Data JPA, diagramado na Figura 3.2, é uma camada de abstração que simplifica o uso do **JPA**, oferecendo ferramentas como repositórios automáticos e consultas baseadas em convenções, enquanto o **Hibernate** é uma implementação da

especificação JPA, responsável pelo mapeamento objeto-relacional e execução das operações de persistência. O Spring Data JPA utiliza o Hibernate internamente para realizar o gerenciamento das entidades, execução de consultas e controle do ciclo de vida dos objetos. Essa integração combina a produtividade do Spring Data JPA com o poder e a flexibilidade do Hibernate, permitindo desenvolver aplicações com menos código e maior eficiência.

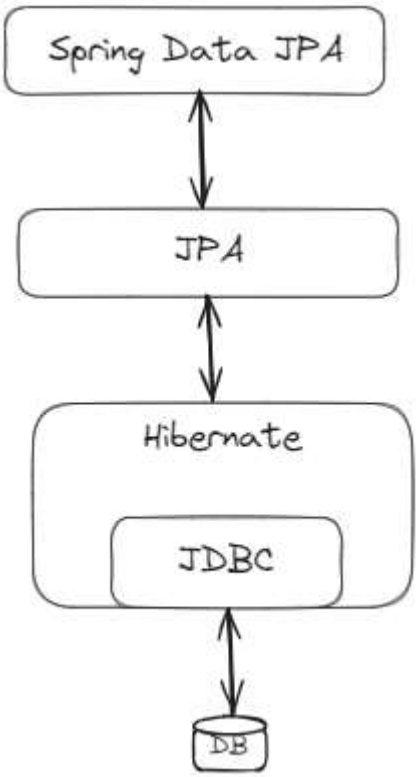

Figura 3.2 – Spring Data JPA.

O Spring Data JPA oferece uma série de recursos que o tornam uma escolha atraente para o desenvolvimento de programação Java:

(1) Criação Automática de Consultas: O Spring Data JPA permite criar consultas dinâmicas com base em nomes de métodos de interface, eliminando com isso, a necessidade de escrever consultas SQL complexas.

(2) Suporte a Paginação e Ordenação: O Spring Data JPA facilita a implementação de paginação e a ordenação de resultados de consultas, tornando a manipulação de grandes volumes de dados mais eficiente.

(3) Integração com o Spring Framework: O Spring Data JPA se integra perfeitamente ao Spring, permitindo o uso de recursos como injeção de dependências, gerenciamento de transações e suporte a testes de forma transparente.

(4) Customização de Consultas: O Spring Data JPA permite a criação de consultas personalizadas usando SQL nativo, oferecendo flexibilidade para lidar com cenários complexos.

(5) Auditoria Automática: O Spring Data JPA oferece suporte à auditoria automática de tabelas, registrando informações como data de criação, data de modificação, usuário responsável.

Dentre as principais vantagens no uso do Spring Data JPA podem ser elencadas:

(1) Aumento da produtividade: reduz o volume de código *boilerplate* necessário para acessar dados, permitindo que desenvolvedores se concentrem na lógica de negócio da aplicação.

(2) Melhoria da testabilidade: facilita a criação de testes unitários e de integração para o código de acesso a dados.

(3) Reutilização de código: oferece diversos recursos prontos para uso, como paginação, ordenação e filtragem de dados, reduzindo a duplicação de código.

(4) Integração com o Spring Framework: se integra perfeitamente com o Spring Framework, facilitando a construção de aplicações Spring robustas e escaláveis.

Resumindo, o Spring Data JPA se destaca como uma solução moderna e eficiente para o acesso a dados em aplicações Java, combinando a simplicidade da JPA com a robustez do *framework* Spring [Walls 2016]. Sua capacidade de aumentar a produtividade, melhorar a mantenabilidade e reduzir erros o torna uma escolha valiosa para projetos Java que buscam otimizar o desenvolvimento e a interação com bancos de dados relacionais.

Capítulo 4
Desenvolvimento do Padrão ORM-DAO

4.1 Considerações Iniciais

A pesquisa desenvolvida e descrita nesta dissertação propõe e explora a implementação de um padrão ORM (Object-Relational Mapping) utilizando DAO em aplicações Java, nomeado de ORM-DAO. O objetivo da pesquisa foi o de criar um padrão ORM que promovesse a compatibilidade entre o modelo de classes de tabela e o modelo relacional de base de dados.

Descrito de uma forma simplificada, o padrão ORM-DAO desenvolvido insere uma chave primária, representada por um único atributo, em cada tabela de determinada base de dados relacional, de uma maneira que cada t-upla, em cada uma das tabelas consideradas, tenha um identificador único. O padrão ORM-DAO também agrega a implementação de uma álgebra relacional, que tem por objetivo tornar as buscas mais flexíveis e eficientes em consultas SQL, visando aumentar a versatilidade na manipulação de condições complexas.

O padrão ORM-DAO desenvolvido também faz uso de uma estrutura de dados em ambiente Java chamada HashMap, definida como uma estrutura de dados na qual podem ser armazenados pares chave-valor que promovem o estabelecimento de uma associação entre dados (valores) e identificadores (chaves) de maneira eficiente, tornando assim a recuperação rápida de valores com base nas chaves. Esse procedimento tende não apenas a diminuir o consumo de memória, mas também aumentar a eficiência do sistema, proporcionando acesso rápido às instâncias previamente carregadas.

4.2 Modelo de Dados Utilizado como Exemplo

Para descrever o padrão ORM-DAO proposto, bem como o seu funcionamento, será usado o exemplo de um modelo de dados físico representado na Figura 4.1, que agrega 4 tabelas associadas, em que cada uma das tabelas é projetada com um único identificador (**ID**). Pode ser observado na figura que a relação entre as tabelas **Livro** e **Autor** é de muitos para muitos (N para N), o que implica a criação de uma terceira tabela intermediária identificada como **LivroAutor**.

A tabela **LivroAutor**, mostrada na Figura 4.1, possui um identificador único, o qual será utilizado como identificador de classe da tabela correspondente. Tanto o atributo **livroId** quanto o **autorId** são chaves estrangeiras na tabela **LivroAutor**, mas não compõem a chave primária, que é definida pelo **ID** para que a tabela tenha um identificador único (apenas um atributo). Para assegurar que não haja duplicidade de combinações de **Livro** e **Autor** na tabela **LivroAutor**, os campos **livroId** e **autorId** devem possuir uma chave alternativa (*alternate key*).

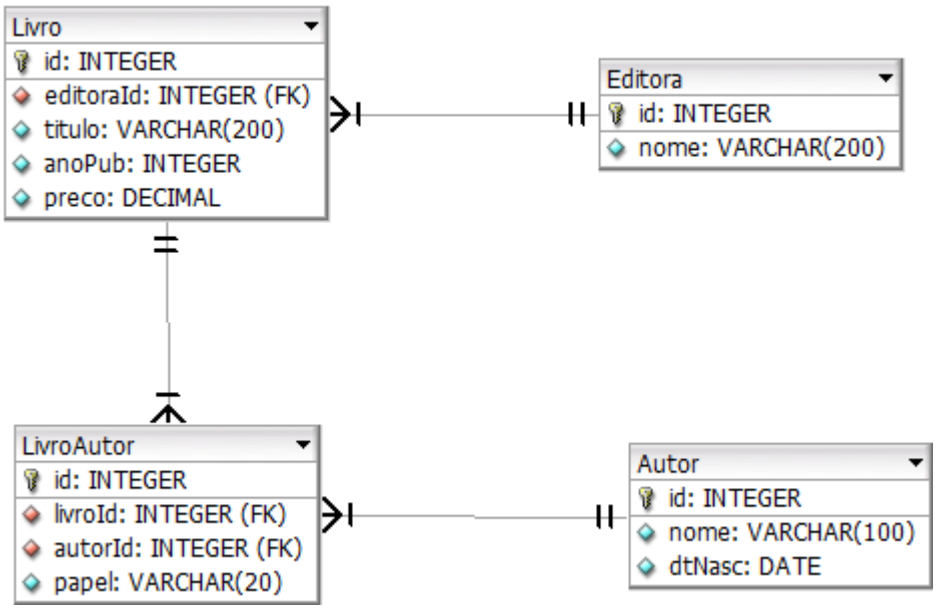

Figura 4.1 – Modelo Físico de Dados.

Conforme pode ser observado no script DDL a seguir, a criação de uma Chave Alternativa/Única, identificada em negrito, que envolve os campos **livroId** e **autorId**, garante a não ocorrência de duplicidade entre a tabela **Livro** e a tabela **Autor**, certificando que não exista mais de um registro em **Livro** que tenha o mesmo **Autor**.

Script DDL:

```
CREATE TABLE IF NOT EXISTS `LivroAutor` (
  `id` INT NOT NULL,
  `papel` VARCHAR(20) NOT NULL,
  `livroId` INT NOT NULL,
  `autorId` INT NOT NULL,
  INDEX `fk_LivroAutor_Autor_idx` (`autorId` ASC) VISIBLE,
  INDEX `fk_LivroAutor_Livro_idx` (`livroId` ASC) VISIBLE,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `LivroAutor_UNIQUE` (`livroId` ASC, `autorId` ASC) VISIBLE,
  CONSTRAINT `fk_LivroAutor_has_Livro`
    FOREIGN KEY (`livroId`)
    REFERENCES `Livro` (`id`)
    ON DELETE RESTRICT
    ON UPDATE RESTRICT,
  CONSTRAINT `fk_LivroAutor_has_Autor`
    FOREIGN KEY (`autorId`)
    REFERENCES `Autor` (`id`)
    ON DELETE RESTRICT
    ON UPDATE RESTRICT)
ENGINE = InnoDB;
```

4.3 MODEL do Padrão MVC (**M**odel **V**iew **C**ontrol**)**

No exemplo do padrão ORM-DAO é utilizado o padrão MVC (ver Anexo 3), porém será desenvolvido apenas o Model, mostrado na Figura 4.2. Para isso, foi implementado a seguinte estrutura de pacotes: model.dao, model.dao.interfaces e model.dao.entites. Essa estrutura permite uma clara separação das responsabilidades, em que model.entities contém as classes de tabela que representam as tabelas da base de dados, model.dao contém as classes de acesso aos dados, e model.dao.interfaces define as interfaces para as classes DAO, facilitando a manutenção e a escalabilidade do código.

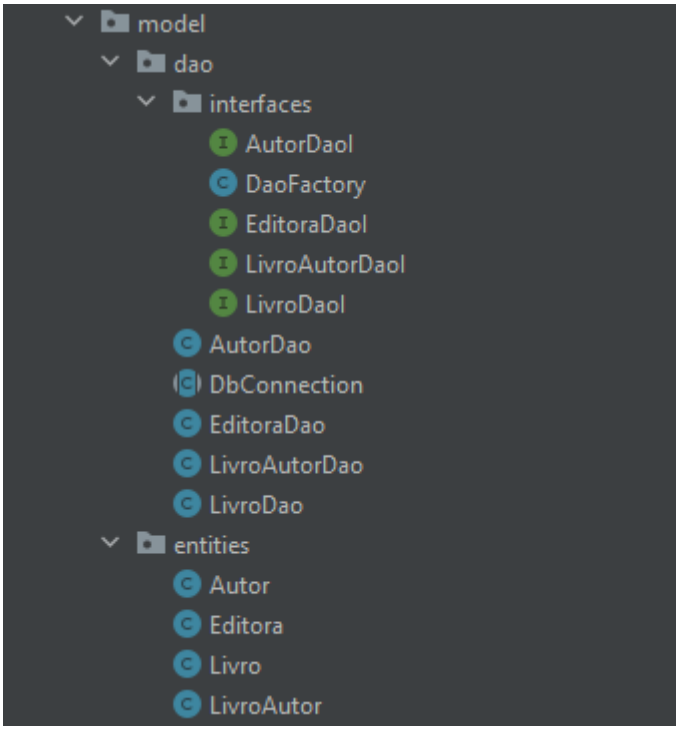

Figura 4.2 – Model do Padrão MVC.

4.4 Diagrama de Classes das Tabelas

O trabalho investiu na construção de um diagrama de classes das tabelas envolvidas no model.dao.entites (ver Figura 4.2), de maneira a poder evidenciar que cada tabela da base de dados é representada por uma classe de tabela correspondente, como mostrado na Figura 4.3.

As relações entre essas classes são modeladas utilizando **associação por composição**, que implica a existência de um objeto depender da existência do outro. Por exemplo, um **Livro** possui uma **Editora** e uma **Editora** pode possuir N **Livro**, refletindo a relação de um para muitos (1:N) entre essas tabelas no contexto do exemplo considerado. Entende-se, pois, que **Livro** é composto por uma **Editora**, ou seja, a existência da tabela **Livro** depende da existência da tabela **Editora**.

A abordagem de composição proposta no trabalho, e diagramada no exemplo mostrado na Figura 4.3, pretende garantir que todas as relações entre as tabelas sejam representadas de forma direta, e que a integridade das associações seja mantida ao longo das operações de persistência e manipulação de dados.

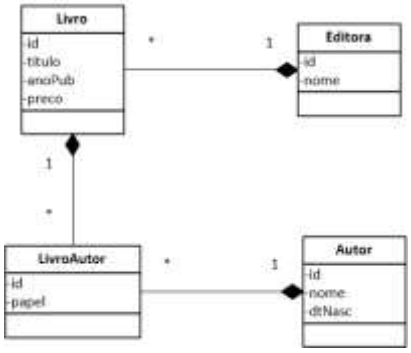

Figura 4.3 – Diagrama de Classes das Tabelas.

No código JAVA desenvolvido, a tabela **Livro** recebe a tabela **Editora** por meio de composição. Isso é evidenciado pelo atributo **editora** na classe **Livro**, que é do tipo **Editora**. A classe **Livro** possui um construtor que aceita uma instância de **Editora** como um dos seus parâmetros, permitindo assim a associação de um objeto **Editora** a um objeto **Livro**. Além disso, os métodos **getEditora** e **setEditora** são utilizados para acessar e modificar essa associação, garantindo que cada instância de **Livro** possa ter uma instância de **Editora** associada a ela.

Código JAVA:

```
package model.entities;

import java.io.Serializable;

public class Livro implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private String titulo;
    private Integer anoPub;
    private Double preco;
    private Editora editora;

    public Livro() {
    }
    public Livro(Integer id, String titulo, Integer anoPub, Double preco, Editora
editora) {
        this.id = id;
        this.titulo = titulo;
        this.anoPub = anoPub;
        this.preco = preco;
        this.editora = editora;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public Integer getAnoPub() {
        return anoPub;
```

```
    }
    public void setAnoPub(Integer email) {
       this.anoPub = anoPub;
    }
    public Double getPreco() {
       return preco;
    }
    public void setPreco(Double preco) {
       this.preco = preco;
    }

    public Editora getEditora() {

       return editora;
    }

    public void setEditora(Editora editora) {
       this.editora = editora;
    }

    @Override
    public String toString() {
       return "Seller [id=" + id + ", titulo=" + titulo + ", anoPub=" + anoPub + ",
preco="
             + preco + " HashCode="  + hashCode() + ", editora=" + editora + "]";
    }
}
```

4.5 Diagramas de Classes do DAO

O diagrama de classes apresentado no model.dao (ver Figura 4.2), ilustra a arquitetura do componente DAO, elemento fundamental na gestão da persistência de dados do sistema representado na Figura 4.4. A classe abstrata **DbConnection** desempenha um papel central ao fornecer métodos utilitários para o estabelecimento e gerenciamento da conexão com a base de dados, além da execução de consultas SQL.

As classes concretas AutorDao, EditoraDao, LivroDao e LivroAutorDao, por sua vez, herdam a funcionalidade de **DbConnection** e implementam as operações **CRUD** (criar, ler, atualizar e deletar) específicas para cada tabela do modelo de domínio. Essa abordagem orientada a objetos encapsula o acesso aos dados e abstrai a complexidade inerente às operações **SQL**, resultando em um código mais modular, reutilizável e de fácil manutenção.

Adicionalmente, o emprego do padrão de projeto **Factory**, através da classe **DaoFactory**, contribui para a flexibilidade do sistema, permitindo a criação dinâmica de instâncias de **DAOs** e facilitando a adaptação a eventuais mudanças na implementação da persistência. Em suma, a arquitetura do componente **DAO**, com sua estrutura em camadas e uso estratégico de padrões de projeto, otimiza a gestão da persistência de dados, promovendo a escalabilidade e a robustez da aplicação.

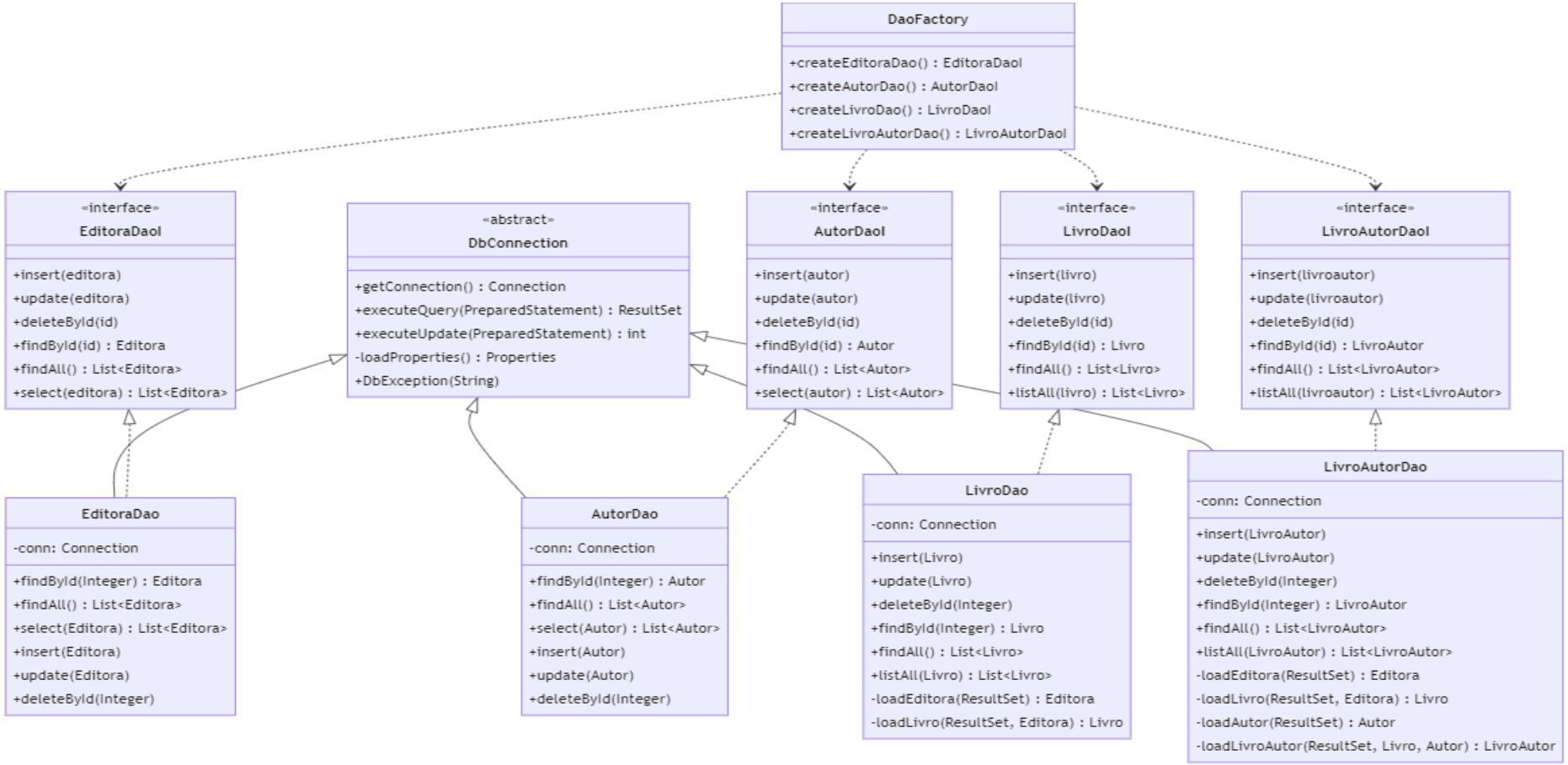

Figura 4.4 – Diagrama de Classes do DAO.

4.6 Diagrama de Sequência

O diagrama de sequência apresentado na Figura 4.5 descreve o fluxo de interações entre o cliente, a fábrica de DAOs (DaoFactory) e um DAO concreto (EditoraDao) durante uma operação de consulta. Inicialmente, o cliente solicita a criação de um DAO específico à fábrica, que instancia o objeto concreto correspondente e retorna uma referência à sua interface. Em seguida, o cliente utiliza essa referência para invocar um método do DAO, tal como o findById. O DAO, por sua vez, estabelece uma conexão com a base de dados por meio da classe DbConnection, prepara e executa a consulta SQL parametrizada, e mapeia o resultado para um objeto de domínio, que é finalmente retornado ao cliente por meio da interface. Essa sequência de ações exemplifica a aplicação do padrão de projeto Factory, promovendo o baixo acoplamento e facilitando a manutenção da camada de persistência de dados.

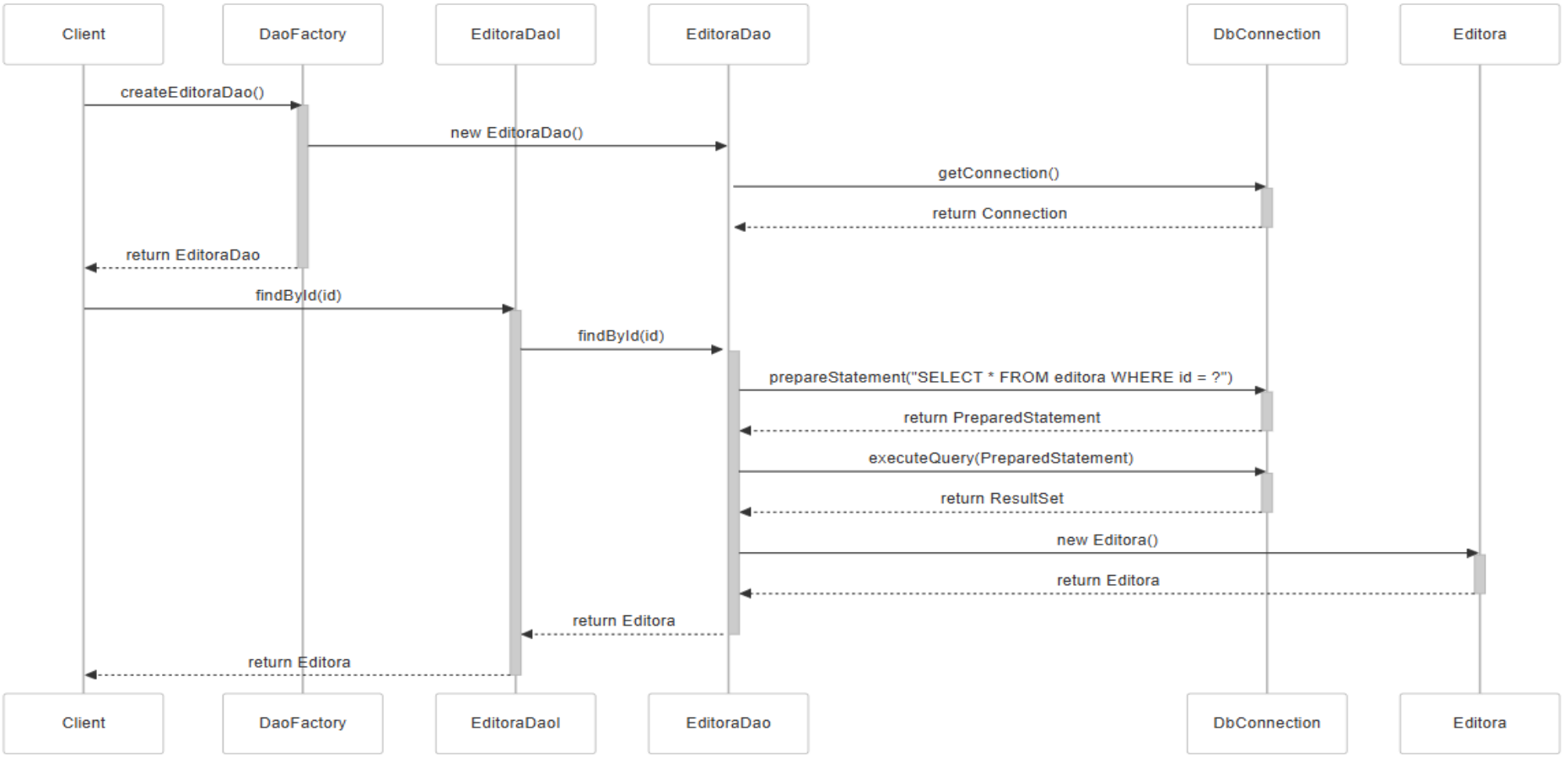

Figura 4.5 – Diagrama de Sequência.

4.7 Padrão ORM-DAO

Nas subseções seguintes, serão abordadas as operações fundamentais de inserção, atualização, deleção e seleção de dados, que constituem o núcleo das interações entre a aplicação e a base de dados. Utilizando o padrão ORM-DAO (Object-Relational Mapping e Data Access Object), a aplicação abstrai as complexidades do gerenciamento de persistência e consultas SQL, mantendo a separação de responsabilidades entre as camadas de negócio e de acesso a dados. Esse padrão permite maior modularidade e flexibilidade no desenvolvimento, garantindo que as operações com a base de dados sejam realizadas de maneira eficiente e segura, facilitando a manutenção e evolução do sistema.

4.7.1 Operações de Inserção e de Atualização de Dados

A Figura 4.6 exibe as operações de inserção e de atualização de dados, apresentado no Código JAVA 1 a seguir, refletindo a interação com a base de dados. No método insert, uma nova Tabela "Editora" é persistida na RDB por meio de uma instrução SQL preparada, encapsulada pelo objeto DAO e executada via Conexão. Já o método update modifica um registro existente na tabela "Editora", alterando o valor do campo "nome" com base no "id" fornecido. Em ambos os casos, o tratamento de erros garante a robustez das duas operações, enquanto a manipulação da Tabela ocorre tanto no Front-End quanto no Back-End, evidenciando o fluxo de dados na aplicação. Em suma, o código materializa as ações de criação e modificação de registros, de maneira a representar a dinâmica de interação com a base de dados visualizada na imagem.

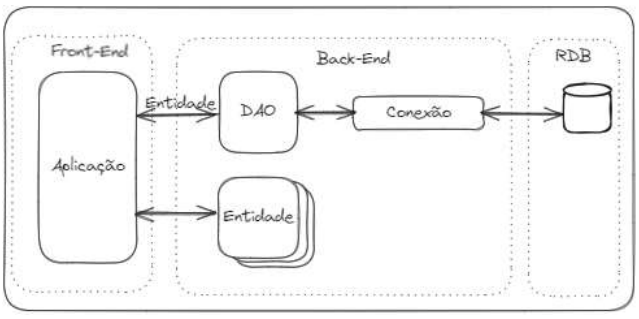

Figura 4.6 – ORM-DAO Insert/Update.

Código JAVA 1:

```
public void insert(Editora editora) {
    PreparedStatement st = null;

    try {
       st = conn.prepareStatement(
             "INSERT INTO editora " +
                   "(nome) " +
                   "VALUES " +
                   "(?)",
             Statement.RETURN_GENERATED_KEYS);

       st.setString(1, editora.getNome());

       int rowsAffected = DbConnection.executeUpdate(st);

       if (rowsAffected > 0) {
          ResultSet rs = st.getGeneratedKeys();
          if (rs.next()) {
             int id = rs.getInt(1);
             editora.setId(id); }
       }
       else {
          DbException("Unexpected error! No rows affected!");
       }
    }
    catch (SQLException e) {
       DbException(e.getMessage());
    }
    finally {
       st=null;
    }
}

public void update(Editora editora) {
    PreparedStatement st = null;
    try {

       st = conn.prepareStatement(
             "UPDATE Editora " +
                   "SET nome = ? " +
                   "WHERE id = ?");

       st.setString(1, editora.getNome());
       st.setInt(2, editora.getId());

       DbConnection.executeUpdate(st);
    }
    catch (SQLException e) {
       DbException(e.getMessage());
    }
    finally {
       st=null;
    }
}
```

4.7.2 Operação de Deleção de Dados

O método delete, apresentado no Código JAVA 2 a seguir, implementa a remoção de dados na arquitetura ilustrada na Figura 4.7. O método recebe o id da Tabela e, por meio de uma instrução SQL preparada no DAO, realiza a exclusão do Registro no RDB via Conexão. O tratamento de erros assegura a robustez da operação, enquanto a

interação com a base de dados ocorre no Back-End, evidenciando a separação de responsabilidades entre as camadas da aplicação.

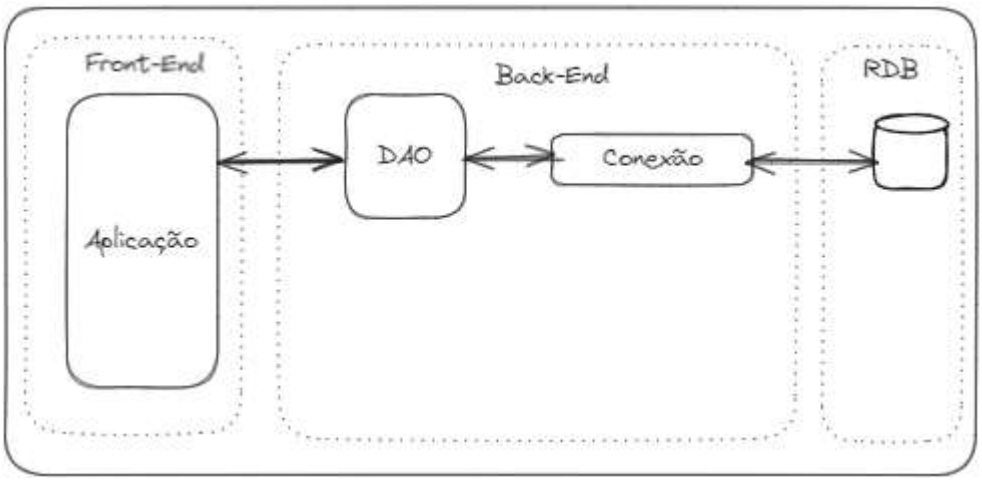

Figura 4.7 – ORM Delete.

Código JAVA 2:

```
public void deleteById(Integer id) {
    PreparedStatement st = null;
    try {

       st = conn.prepareStatement(
             "DELETE FROM Editora WHERE id = ?");

       st.setInt(1, id);

       DbConnection.executeUpdate(st);
    }
    catch (SQLException e) {
       DbException(e.getMessage());
    }
    finally {
       st=null;
    }
}
```

4.7.3 Operação de Seleção de Dados

O método select, apresentado no Código JAVA 3 a seguir, implementa a recuperação de dados na arquitetura ilustrada na Figura 4.8. Ele utiliza um objeto DAO para construir e executar uma consulta SQL parametrizada. A consulta seleciona registros da tabela Editora com base nos atributos id e nome do objeto Editora fornecido. A lógica da consulta utiliza a álgebra relacional para construir condições flexíveis na cláusula WHERE.

Condição para o **id**: A condição **(id=? OR Optional.ofNullable(editora.getId()).isEmpty())** traduz-se para a álgebra relacional como **σ ((id = <valor_fornecido>) ∨ (id = null))**. Isso significa que a consulta selecionará registros em que o **id** é igual ao valor fornecido OU onde o **id** é nulo, permitindo buscas mesmo quando o **id** não é especificado.

Condição para o **nome**: Similarmente, a condição para o atributo **nome** segue a mesma lógica, selecionando registros em que o **nome** corresponde ao valor fornecido ou é nulo.

Após a execução da consulta, os dados recuperados são transformados em uma lista de objetos Editora e retornados. O tratamento de erros garante a robustez do método, enquanto a interação com a base de dados ocorre no Back-End, refletindo a separação de responsabilidades dentro da aplicação.

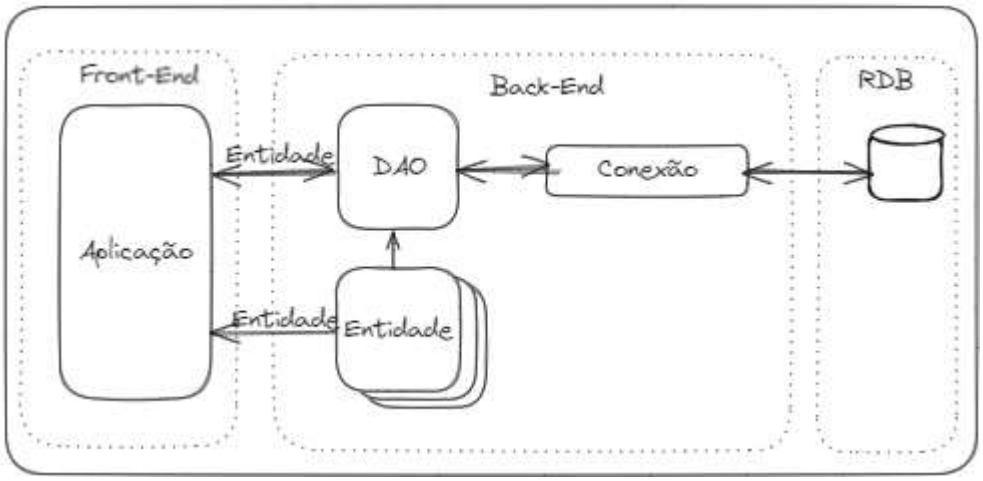

Figura 4.8 – ORM Select.

Código JAVA 3:

```
public List<Editora> select(Editora editora) {
    PreparedStatement st = null;
    ResultSet rs = null;

    try {
       st = conn.prepareStatement(
             "SELECT * FROM editora " +
                   "WHERE (id = ? OR " +
Optional.ofNullable(editora.getId()).isEmpty() + ")" +
                   "AND (nome = ? OR " +
Optional.ofNullable(editora.getNome()).isEmpty() + ")" );

       st.setInt(1, Optional.ofNullable(editora.getId()).orElse(0));
       st.setString(2, Optional.ofNullable(editora.getNome()).orElse(""));
```

```
            rs = DbConnection.executeQuery(st);

            List<Editora> list = new ArrayList<>();

            while (rs.next()) {
                Editora obj = new Editora();
                obj.setId(rs.getInt("id"));
                obj.setNome(rs.getString("nome"));
                list.add(obj);
            }
            return list;
        }
        catch (SQLException e) {
            DbException(e.getMessage());
        }
        finally {
            st=null;
            rs=null;

        }
        return null;
    }
```

4.8 Buscas Flexíveis em Consultas SQL

Com relação a buscas flexíveis em consultas SQL (ver **Código JAVA de Busca:**), o padrão proposto utiliza a expressão da álgebra relacional **σ ((<condição1> = <condição2>) ∨ (<condição2> = null))**. Espera-se com isso obter ganhos significativos no tempo de desenvolvimento bem como um aumento de flexibilidade adicional, sem a necessidade de criar novos métodos de consulta para cada tipo específico de busca solicitada. A utilização da expressão é vantajosa pelos motivos elencados abaixo:

- **Flexibilidade Dinâmica**: A flexibilidade da cláusula WHERE permite que a consulta se adapte dinamicamente com base nos parâmetros fornecidos. Com isso, é possível realizar consultas variadas sem alterar a estrutura fundamental da consulta SQL.
- **Redução de Código Redundante**: Evita a redundância de criar múltiplos métodos de consulta para diferentes combinações de critérios de busca. Ao invéz de criar métodos separados para cada combinação possível, a lógica de filtragem é encapsulada na própria consulta SQL.
- **Manutenção Simplificada**: Com um número menor de métodos específicos para gerenciar, a manutenção do código se torna mais simples. Alterações nos critérios de busca podem ser feitas diretamente na construção da consulta SQL sem afetar a estrutura geral do código da aplicação.

- **Performance e Eficiência**: Com a utilização de uma única consulta flexível, otimizada e parametrizada (como um PreparedStatement), a base de dados pode executar a consulta de maneira eficiente, aproveitando índices e outras otimizações disponíveis.
- **Segurança contra Injeção SQL**: O uso de PreparedStatement ajuda a prevenir ataques de injeção SQL, garantindo que os parâmetros sejam tratados de maneira segura pela API JDBC, sem comprometer a integridade ou segurança dos dados.

Ao empregar essa abordagem, não apenas ganha flexibilidade e eficiência na construção de consultas SQL dinâmicas, mas também melhora a manutenção do código e a segurança da aplicação, ao mesmo tempo em que mantém a performance da consulta na base de dados.

Código Java de Busca:

```
st=conn.prepareStatement(
SELECT * FROM editora
WHERE (id = ? OR  Optional.ofNullable(editora.getId()).isEmpty())
AND (nome = ? OR  Optional.ofNullable(editora.getNome()).isEmpty()));
```

4.9 Otimização de Memória

Conforme comentado anteriormente com relação à otimização de memória, a pesquisa realizada investiu na implementação de um mecanismo de *caching* de objetos utilizando HashMap, como ilustrado na Figura 4.9. Esta estratégia visa otimizar o consumo de memória e aprimorar a eficiência do sistema, garantindo acesso rápido a instâncias de objetos Editora previamente carregadas. Ao associar cada objeto Livro a uma única instância de Editora em memória, o HashMap elimina a redundância de dados e agiliza o processo de recuperação de informações relacionadas às editoras.

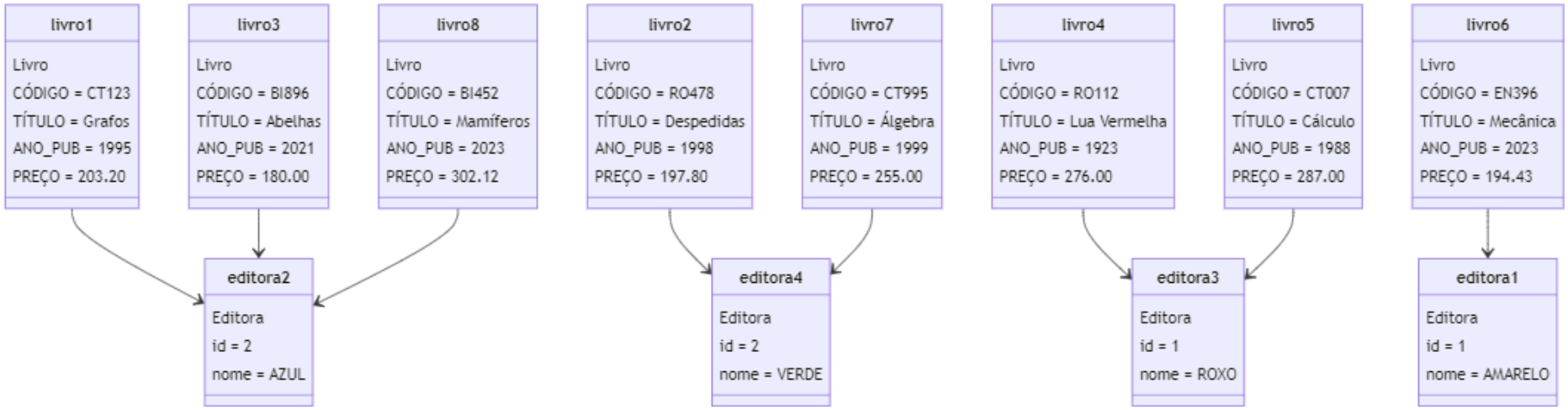

Figura 4.9 – Diagrama de Objeto Livro e de Objeto Editora.

No **Código JAVA da Otimização de Memória** escrito no final da página, o uso do **HashMap** para realizar o *caching* dos objetos **Editora** traz benefícios significativos em termos de consumo de memória, descritos na sequência:

- **Redução de *Overhead* de Memória:** Utilizando o **HashMap**, é possível armazenar instâncias únicas de objetos **Editora** com base no atributo **editoraId**. Isso significa que, se várias instâncias de **Livro** compartilham a mesma **Editora**, apenas uma instância dessa **Editora** é mantida em memória. Isso reduz significativamente o consumo de memória, pois evita a duplicação de objetos.
- **Eficiência na Utilização de Recursos:** Ao evitar a duplicação de objetos **Editora**, o código se torna mais eficiente na utilização dos recursos de memória. Isso é especialmente importante em aplicações que lidam com grandes volumes de dados ou então são processadas em ambientes onde a eficiência de memória é crucial para o desempenho geral da aplicação.
- **Melhoria na Performance:** Além de economizar memória, o *caching* com **HashMap** também pode melhorar a performance da aplicação. Acesso rápido aos objetos **Editora** já carregados em memória significa que as operações que requerem referências frequentes a esses objetos são executadas mais rapidamente, sem a necessidade de criar mais um objeto **Editora**.

Portanto, ao utilizar um **HashMap** para fazer *caching* dos objetos **Editora**, o código não apenas otimiza o uso de memória, mas também contribui para um desempenho mais eficiente da aplicação, garantindo que as operações de carregamento e referência aos objetos sejam realizadas de maneira rápida e econômica em termos de recursos computacionais.

Código JAVA da Otimização de Memória:

```
List<Livro> list = new ArrayList<>();
Map<Integer, Editora> map = new HashMap<>();
while (rs.next()) {
    Editora editora = map.get(rs.getInt("editoraId"));
    if (editora == null) {
       editora = loadEditora(rs);
       map.put(rs.getInt("editoraId"), editora);
    }
    Livro obj = loadLivro(rs, editora);
    list.add(obj);
}
return list;
```

4.10 Utilização de Padrões

No padrão ORM-DAO proposto foram utilizados diversos conceitos e padrões de design, incluindo alguns padrões GoF (*Gang of Four*), alguns princípios do SOLID e algumas práticas de Clean Code para garantir a robustez e a escalabilidade da aplicação.

Os autores do (GOF) [Gamma et al. 1994] não mencionam explicitamente o padrão DAO (Data Access Object) em seu livro. O padrão DAO surgiu mais tarde, influenciado pelos conceitos de encapsulamento e separação de responsabilidades já abordados no trabalho dos GOF, mas voltado especificamente para o acesso a dados persistentes.

No livro [Gamma et al. 1994], os GOF descrevem padrões de projeto mais gerais e reutilizáveis, como Factory, Singleton, Adapter, Strategy, entre outros. O DAO, por outro lado, é considerado um padrão arquitetural mais específico, que se tornou popular com o crescimento de frameworks e tecnologias para persistência de dados, como o J2EE (Java EE).

Os princípios que fundamentam o DAO, como a separação de responsabilidades (isolar a lógica de acesso a dados do restante da aplicação), estão alinhados com os conceitos do GOF, mas o padrão em si não é mencionado.

Padrão DAO:

1. **DAO (Data Access Object)**:
    - O padrão DAO é implementado para fornecer uma abstração da camada de persistência e facilitar o gerenciamento das operações de base de dados para cada tabela (**Autor, Editora, Livro, LivroAutor).**
    - O padrão Data Access Object (DAO) é utilizado para abstrair e encapsular o acesso á base de dados, mitigando o problema de **impedance mismatch**. Isso é alcançado por meio da definição clara de métodos CRUD e de buscas nas interfaces DAO, que simplificam a interação com a camada de persistência e isolam a lógica de acesso a dados do restante da aplicação.

- O padrão DAO utiliza SQL nativo para consultas, mitigando o problema de **lazy loading**, comum em ORMs, com o uso criterioso de consultas controladas e carregamento seletivo dos dados. Nos métodos de buscas, apenas os dados necessários são carregados.

Padrão GoF:

1. **Factory:**
   - A DaoFactory segue o padrão Factory, permitindo a criação de objetos sem expor a lógica de instanciamento, promovendo a reutilização e a manutenção do código. Métodos como createEditoraDao, createAutorDao, createLivroDao, e createLivroAutorDao exemplificam isso (ver Anexo 3).

Princípios SOLID:

1. **Single Responsibility Principle (SRP)**:
   - Cada classe tem uma única responsabilidade. As classes de tabela (**Autor**, **Editora**, **Livro**, **LivroAutor**) representam tabelas do domínio, enquanto os DAOs (**AutorDao**, **EditoraDao**, **LivroDao**, **LivroAutorDao**) são responsáveis pelas operações de base de dados dessas tabelas.
2. **Open/Closed Principle (OCP)**:
   - As interfaces DAO (por exemplo, **LivroDaoI**, **AutorDaoI**) permitem a extensão das funcionalidades sem modificar o código existente. Novas implementações de DAOs sem que a lógica das interfaces sejam alteradas.
3. **Liskov Substitution Principle (LSP)**:
   - As implementações concretas dos DAOs (**AutorDao**, **EditoraDao**, etc.) podem ser substituídas por qualquer outra implementação que siga a mesma interface (por exemplo, **AutorDaoI**), sem quebrar o comportamento do sistema.

4. **Interface Segregation Principle (ISP)**:
    - Interfaces específicas (por exemplo, **LivroDaoI**, **AutorDaoI**) são definidas para cada DAO, evitando interfaces genéricas e complexas, facilitando assim a implementação e uso.
5. **Dependency Inversion Principle (DIP)**:
    - A aplicação depende de abstrações (interfaces) em vez de implementações concretas. A DaoFactory cria instâncias concretas dos DAOs, mas o código que utiliza essas instâncias depende das interfaces.

Práticas de Clean Code:

1. **Nomes Significativos**:
    - Classes, métodos e variáveis devem possuir nomes claros e descritivos, como **Editora**, **Livro**, **AutorDao**, **findById**, **insert**, o que facilita a compreensão do código.
2. **Pequenos Métodos**:
    - Os métodos devem ser concisos e limitados a apenas uma funcionalidade de execução, o que promove a legibilidade e a manutenibilidade do código.
3. **Tratamento de Exceções e Erros**:
    - O tratamento de exceções e erros é realizado de maneira consistente, e é encapsulado no método **DbException**, que propaga mensagens claras.
4. **Comentário Relevante**:
    - O código do ORM-DAO é quase sempre autoexplicativo, evitando comentários redundantes. Quando necessário, comentários são usados para explicar partes críticas do código.

4.11 Análise de Complexidade

Analisando a complexidade da Classe **LivroAutorDAO** em 4 passos:

1. **Preparação da Declaração SQL:**

```
st = conn.prepareStatement(
  "SELECT ... "
      + "FROM ... "
      + "WHERE ... "
      + "ORDER BY ...");
```

A complexidade desta operação é constante, ou seja, **O(1)**, pois preparar uma declaração SQL não depende do volume de dados.

2. **Configuração dos Parâmetros:**

```
st.setInt(1, Optional.ofNullable(livroautor.getId()).orElse(0));
st.setString(2, Optional.ofNullable(livroautor.getPapel()).map(p -> "%" + p + "%").orElse(""));
```

A complexidade também é **O(1)**, uma vez que a configuração dos parâmetros não depende da quantidade de parâmetros envolvidos.

3. **Execução da Consulta**

```
rs = DbConnection.executeQuery(st);
while (rs.next()) { ... }
```

A execução da consulta SQL depende da implementação da base de dados e do volume de dados. Em geral, a complexidade pode variar, mas objetivando exemplificação, é assumido que a execução da consulta, dentro do laço de repetição é proporcional ao número de registros (n) retornados, **O(n)**.

4. **Total:**

Preparação da declaração SQL: **O(1)**

Configuração dos parâmetros: **O(1)**

Processamento dos resultados: **O(n)**

Portanto, a complexidade total do código ORM-DAO é dominada pela execução da consulta e pelo processamento dos resultados, resultando em uma complexidade de **O(n)**, onde n é o número de registros retornados pela consulta SQL.

Capítulo 5
Experimentos Usando ORM-DAO e Spring Data JPA

5.1 Análise Comparativa do Padrão Proposto

Com o intuito de avaliar a eficácia do padrão proposto nessa dissertação, foram conduzidos experimentos para avaliação de desempenho (consumo de memória e tempo de execução) entre o ORM-DAO proposto nesse trabalho e o Spring Data JPA tradicional. Para tanto, foram utilizados dados que estão organizados em quatro tabelas que se diferenciam entre si em volumes de registros, permitindo uma análise abrangente em cenários e de cardinalidades distintas.

Uma breve descrição das tabelas é apresentada na sequência. A Tabela Autor (Tabela 5.1) contém 1.000 registros e a Tabela Editora (Tabela 5.2) também tem 1.000 registros. A Tabela Livro (Tabela 5.3) contém 1.000.000 de registros, representando um conjunto de dados considerável. Já a Tabela LivroAutor (Tabela 5.4), que estabelece a relação muitos-para-muitos entre Livros e Autores, possui 2.499.169 registros, evidenciando a cardinalidade complexa desse relacionamento.

Utilizando a mesma base de dados que é composta pelas 4 tabelas mencionadas no parágrafo anterior, foram realizados experimentos envolvendo vinte execuções de consulta (utilizando o operador *select*) em ambos os padrões. Muito embora os dados utilizados em cada uma das execuções foram os mesmos, as 20 execuções foram realizadas com objetivo de equalizar os resultados obtidos por meio de um valor médio. Essa estratégia foi adotada pelo motivo do ambiente computacional em que os experimentos foram conduzidos sofrer influência do sistema operacional.

O experimento colecionou o tempo de execução bem como o consumo de memória, aspectos cruciais para evidencia eficiência e escalabilidade de aplicações, especialmente em cenários com grande volume de dados.

5.2 Experimentos Envolvendo Dados da Tabela Autor e da Tabela Editora

A Tabela 5.1 apresenta os resultados dos experimentos realizados com os dados da Tabela Autor, que possui 1.000 registros. A Tabela 5.2 apresenta os resultados dos experimentos realizados com os dados da Tabela Editora, que também possui 1.000 registros.

Os valores apresentados em ambas as tabelas, exibem um comportamento similar. O ORM-DAO apresentou um tempo de execução e consumo de memória consistentemente menores em comparação àqueles apresentados pelo Spring JPA tradicional. Nas tabelas com menor volume de dados, a diferença de performance entre os dois padrões é menos expressiva, o que era esperado, visto que o impacto do ORM-DAO se torna mais evidente em aplicações com maior complexidade e maior volume de dados.

Com relação ao ORM-DAO para tabela Autor, foi utilizado a seguinte *query*: "SELECT * FROM autor;"

Tabela 5.1 Resultados dos Experimentos (ORM-DAO vs Spring JPA) com Dados da Tabela Autor.

	Tempo Execução(ms)		Consumo Memória (Bytes)	
Execuções	ORM-DAO	Spring JPA	ORM-DAO	Spring JPA
Execução 01	2.600	7.948	453.991.448	641.206.272
Execução 02	1.627	7.588	486.786.712	619.675.736
Execução 03	1.474	10.795	524.377.624	649.574.312
Execução 04	1.494	9.674	535.991.376	663.214.344
Execução 05	1.484	8.894	523.940.760	658.533.848
Execução 06	1.576	9.899	521.501.944	703.064.560
Execução 07	1.500	9.268	473.750.264	727.719.128
Execução 08	1.772	10.613	455.922.360	705.181.032
Execução 09	1.463	9.774	517.251.400	726.148.488
Execução 10	1.534	8.792	472.985.784	726.164.440
Execução 11	1.525	4.497	513.594.040	692.069.248
Execução 12	1.590	5.395	495.885.496	721.792.952
Execução 13	1.534	8.292	542.643.512	729.303.952
Execução 14	1.635	5.520	544.002.664	736.989.112
Execução 15	1.602	4.459	469.029.424	721.810.544
Execução 16	1.569	4.835	474.957.824	748.536.576
Execução 17	1.562	5.030	473.055.272	743.816.768
Execução 18	1.540	5.842	497.025.024	739.633.432
Execução 19	2.074	4.426	497.551.360	727.658.408
Execução 20	2.679	4.244	501.336.576	720.851.168
MÉDIA	**1.692**	**7.289**	**498.779.043**	**705.147.216**

Com relação ao ORM-DAO para tabela Editora, foi utilizada a seguinte *query*: "SELECT * FROM editora;"

Tabela 5.2 Resultados dos Experimentos (ORM-DAO vs Spring JPA) com Dados da Tabela Editora.

	Tempo Execução(ms)		Consumo Memória (bytes)	
Execuções	ORM-DAO	Spring JPA	ORM-DAO	Spring JPA
Execução 01	499	295	6.471.400	8.912.896
Execução 02	71	24	555.520	1.048.576
Execução 03	73	14	647.800	1.048.576
Execução 04	73	16	1.111.368	1.048.576
Execução 05	70	36	575.984	1.048.576
Execução 06	74	36	576.104	1.048.576
Execução 07	68	8	576.112	1.048.576
Execução 08	65	11	576.120	1.048.576
Execução 09	97	18	576.104	1.048.576
Execução 10	64	13	576.104	1.048.576
Execução 11	63	9	576.104	1.048.576
Execução 12	72	7	574.432	1.048.576
Execução 13	71	6	579.880	1.048.576
Execução 14	60	17	579.720	1.048.576
Execução 15	70	33	578.224	1.586.872
Execução 16	61	12	601.264	1.048.576
Execução 17	56	8	579.880	1.048.576
Execução 18	60	65	579.880	1.048.576
Execução 19	74	8	579.880	1.048.576
Execução 20	55	7	579.880	1.048.576
MÉDIA	**90**	**32**	**902.588**	**1.468.707**

5.3 Experimentos Envolvendo Dados da Tabela Livro

A Tabela 5.3 exibe os resultados dos experimentos com registros da tabela Livro. Pode ser evidenciado na tabela que o padrão ORM-DAO apresenta melhores resultados quando comparados aos resultados obtidos pelo Spring JPA tradicional.

O tempo médio de execução para o ORM-DAO foi de 1.692 ms e para o Spring JPA tradicional foi de 7.269 ms.

O consumo de memória foi inferior na execução com o ORM-DAO, com uma média de 498.779.043 bytes quando comparado com a média de 700.147.216 bytes obtidos com o uso do Spring JPA tradicional.

Com relação ao ORM-DAO para tabela Livro, foi utilizada a seguinte *query*: "SELECT livro.*,editora.id as editoraId,editora.nome as editoraNome FROM livro INNER JOIN editora ON livro.editora_id = editora.id;"

Tabela 5.3 Resultados dos Experimentos (ORM-DAO vs Spring JPA) com Dados da Tabela Livro.

	Tempo Execução(ms)		Consumo Memória (Bytes)	
Execuções	ORM-DAO	Spring JPA	ORM-DAO	Spring JPA
Execução 01	2.600	7.948	453.991.448	641.206.272
Execução 02	1.627	7.588	486.786.712	619.675.736
Execução 03	1.474	10.795	524.377.624	649.574.312
Execução 04	1.494	9.674	535.991.376	663.214.344
Execução 05	1.484	8.894	523.940.760	658.533.848
Execução 06	1.576	9.899	521.501.944	703.064.560
Execução 07	1.500	9.268	473.750.264	727.719.128
Execução 08	1.772	10.613	455.922.360	705.181.032
Execução 09	1.463	9.774	517.251.400	726.148.488
Execução 10	1.534	8.792	472.985.784	726.164.440
Execução 11	1.525	4.497	513.594.040	692.069.248
Execução 12	1.590	5.395	495.885.496	721.792.952
Execução 13	1.534	8.292	542.643.512	729.303.952
Execução 14	1.635	5.520	544.002.664	736.989.112
Execução 15	1.602	4.459	469.029.424	721.810.544
Execução 16	1.569	4.835	474.957.824	748.536.576
Execução 17	1.562	5.030	473.055.272	743.816.768
Execução 18	1.540	5.842	497.025.024	739.633.432
Execução 19	2.074	4.426	497.551.360	727.658.408
Execução 20	2.679	4.244	501.336.576	720.851.168
MÉDIA	**1.692**	**7.289**	**498.779.043**	**705.147.216**

5.4 Experimentos envolvendo tabela LivroAutor

A Tabela 5.4 apresenta os resultados dos experimentos realizados com os dados da Tabela LivroAutor, que possui o maior volume de dados entre as 4 tabelas consideradas. Observa-se uma diferença significativa nos valores do tempo de execução entre o padrão proposto (ORM-DAO) e o Spring JPA tradicional. O padrão ORM-DAO apresentou um tempo médio de execução de 12.553 ms, enquanto o Spring JPA tradicional apresentou um tempo médio de 73.712 ms. Essa diferença expressiva indica um ganho considerável em performance com a utilização do ORM-DAO, especialmente em cenários com grande volume de dados e relacionamentos complexos. Em relação ao consumo de memória, o ORM-DAO também se mostrou mais eficiente, com um consumo médio de 1.318.437.485 bytes, contra 1.981.786.002 bytes do Spring JPA tradicional, o que corrobora a hipótese de que o ORM-DAO opera com uma menor quantidade de abstrações e demanda menos recursos do sistema.

Com relação ao ORM-DAO para tabela LivroAutor, foi utilizada a seguinte *query*:

"SELECT livro_autor.id, livro_autor.papel, livro.id as livroId, livro.titulo as livroTitulo, livro.ano_pub as livroAnoPub, livro.preco as livroPreco, editora.id as editoraId, editora.nome as editoraNome, autor.id as autorId, autor.nome as autorNome, autor.dt_nasc as autorDtNasc FROM livro_autor, livro, autor, editora WHERE livro_autor.livro_id = livro.id AND livro.editora_id = editora.id AND livro_autor.autor_id = autor.id ORDER BY livro_autor.id;"

Tabela 5.4 Resultados dos Experimentos (ORM-DAO vs Spring JPA) com Dados da Tabela LivroAutor.

	Tempo Execução(ms)		**Consumo Memória (Bytes)**	
Execuções	ORM-DAO	Spring JPA	ORM-DAO	Spring JPA
Execução 01	13.305	80.559	1.344.234.520	2.041.878.800
Execução 02	12.599	77.432	1.324.351.488	2.147.292.640
Execução 03	12.180	73.328	1.305.242.192	1.968.107.728
Execução 04	12.379	73.769	1.318.211.776	1.935.806.088
Execução 05	12.432	73.153	1.326.204.560	1.945.273.144
Execução 06	12.462	75.849	1.315.829.848	1.980.964.080
Execução 07	12.569	79.644	1.312.572.768	1.980.082.912
Execução 08	12.160	75.410	1.317.531.552	1.998.317.840
Execução 09	13.033	74.493	1.290.554.256	1.947.985.760
Execução 10	13.624	73.638	1.331.189.136	2.031.867.296
Execução 11	12.444	74.002	1.319.914.416	1.957.421.336
Execução 12	12.268	72.663	1.338.787.552	2.006.892.504
Execução 13	12.749	74.347	1.289.504.480	1.912.529.416
Execução 14	12.247	70.682	1.312.275.120	2.227.123.880
Execução 15	12.533	70.600	1.309.425.216	2.036.280.432
Execução 16	12.481	69.819	1.320.378.944	1.937.713.856
Execução 17	12.352	73.503	1.314.665.912	1.930.374.944
Execução 18	12.322	71.003	1.345.074.248	1.916.747.360
Execução 19	12.527	70.016	1.329.347.560	1.954.482.736
Execução 20	12.390	70.339	1.323.454.360	1.978.596.888
MÉDIA	**12.553**	**73.712**	**1.319.437.495**	**1.991.786.982**

Capítulo 6
Conclusão & Trabalhos Futuros

6.1 Conclusão

Os resultados apresentados nas seções anteriores mostram que o ORM-DAO oferece um ganho significativo em performance, especialmente em cenários com grande volume de dados e relacionamentos complexos. A redução no tempo de execução e no consumo de memória evidencia a eficiência do ORM-DAO tanto na otimização de consultas quanto no gerenciamento de recursos. O uso do ORM-DAO em aplicações com menor volume de dados também apresenta vantagens com ralação ao Spring JPA, muito embora sejam menos expressivas.

Em relação ao tempo de execução, observou-se maior estabilidade entre as execuções com o uso do ORM-DAO, enquanto que com o Spring JPA uma certa variação foi notada. Essa constatação sugere que o ORM-DAO oferece um desempenho mais previsível, o que pode ser determinante em aplicações que exigem estabilidade e tempos de resposta previsíveis.

A escolha ideal entre as duas abordagens dependerá das necessidades e prioridades de cada projeto que estiverem envolvidas. O ORM-DAO se destaca pela eficiência no consumo de memória e consistência no tempo de execução, enquanto o Spring Data JPA prioriza a facilidade de uso e a flexibilidade. A decisão final sobre qual padrão adotar deve levar em consideração as particularidades da aplicação, os requisitos de desempenho e os recursos computacionais disponíveis.

6.6 Trabalhos Futuros

O trabalho realizado e descrito nesta dissertação de mestrado teve como objetivo promover, em um ambiente computacional orientado a objeto, a compatibilidade entre

o modelo de classes de tabelas e o modelo relacional de bases de dados, por meio da proposta e desenvolvimento de um padrão, nomeado de ORM-DAO.

Como descrito em detalhes na dissertação, o desenvolvimento e implementação de um padrão com as características do ORM-DAO pode, de certa forma, permitir e sustentar possiblidades de extensões das implementações desenvolvidas, bem como colaborar na exploração de aplicações mais variadas, com o propósito de expandi-lo.

Dentre as possibilidades de expansão do padrão ORM-DAO criado, foram cogitados, ao longo de seu desenvolvimento e, posteriormente, de sua operacionalidade, as seguintes linhas de investigação:

- Desenvolvimento de uma ferramenta CASE (*Computer-Aided Software Engineering*) que possa ser integrada diretamente com bases de dados relacionais. O objetivo central dessa ferramenta seria automatizar a geração da camada Model (ver Seção 4.3), crucial para a implementação do padrão ORM-DAO. A ferramenta CASE sendo cogitada a ser desenvolvida teria a capacidade de se conectar à base de dados relacional, analisar suas tabelas e seus relacionamentos e, em seguida, gerar automaticamente as classes de tabela e os DAOs (*Data Access Objects*) correspondentes. A ferramenta traria diversos benefícios, incluindo:

  (a) aumento da produtividade, uma vez que a geração automática de código reduziria o tempo e o esforço necessários para a criação da camada Model, permitindo que desenvolvedores se concentrem em outras partes da aplicação;
  (b) redução de erros: A ferramenta CASE eliminaria a possibilidade de erros manuais na criação das classes e DAOs, garantindo a consistência e a integridade do código gerado;
  (c) facilidade de manutenção: A ferramenta CASE facilitaria a manutenção da camada Model, já que qualquer alteração na base de dados poderia ser refletida automaticamente no código da aplicação.

- O padrão ORM-DAO pode trazer avanços significativos para a área de Engenharia de Software em ambientes computacionais voltados à programação orientada a objetos, impulsionando a adoção de práticas e padrões que aprimorem a qualidade e a eficiência do desenvolvimento de aplicações.

Referências & Bibliografia

[Abraham et al. 2021] Silberschatz A, Korth H, Sudarshan S (2021). Database System Concepts (17th Ed.), Ed. McGraw-Hill.

[Ambler 1997] Ambler S. W. (1997) Mapping objects to relational databases, An AmbySoft Inc. White Paper, http://www.AmbySoft.com/mappingObjects.pdf

[Asenjo et al. 2023] Daniel Asenjo, Nicolás Barroso, Laura Jalón, and Carlos Manso. 2023. Object Oriented Database Interface for a Relational Database: A Didactic-Pedagogical Strategy for Teaching Programming Techniques. In Proceedings of the 3rd International Conference on Advanced Research in Education. Association for Computing Machinery, 15–22.

[Barnes 2007] Barnes J M (2007). Object-relational mapping as a persistence mechanism for object-oriented applications, Mathematics, Statistics, and Computer Science Honors Projects 6, https://digitalcommons.macalester.edu/mathcs_honors/6

[Bauer et al. 2005] Bauer, H. H., Barnes, S. J., Reichardt, T., & Neumann, M. M. (2005). Driving consumer acceptance of mobile marketing: A theoretical framework and empirical study.1 International Journal of Electronic Commerce,2 9(3), 181-202.

[Bloch 2017] Bloch, J (2017). Effective Java (3rd ed.), Ed. Addison-Wesley Professional.

[Booch 2010] Booch G (2010). Análise e Projeto com Orientação a Objetos: Usando UML e Padrões de Design, Ed. Elsevier Brasil, Rio de Janeiro.

[Brown 2011] Brown K D (2011). Modelagem Conceitual de Dados: um Foco em Bancos de Dados Relacionais (2ª ed.), Ed. Elsevier Brasil, Rio de Janeiro.

[Byrne et al. 2033] Eileen Byrne, Gillian Lyons, and Paul Rigby. 2003. Teaching object-role modeling and relational database design: a pedagogical and cognitive perspective. In Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education. ACM, 213–217.

[Chen 1976] Chen, P P (1976). The entity-relationship model: Toward a Unified View of Data, Ed. Hardpress Publishing.

[Codd 1970] Codd E F (1970). A relational model of data for large shared data banks. Communications of the ACM, v. 13, no. 5, pp. 377-387.

[Codd 1985a] Codd E F (1985). Is your DBMS really relational?, Computerworld, October 14th, v. 19, pID1, consultado em 07-02-2024.

[Codd 1985b] Codd E F (1985). Does your DBMS run by the rules? Computerworld, October 21st, v. 19, p. 49, consultado em 27-03-2024.

[Codd et al. 1974] Codd, E. F., & Boyce, R. F. (1974). Normalization of Data: The Boyce-Codd Normal Form (BCNF). IBM Research Report, RJ 1334.

[Date 2010] Date C J (2010). An Introduction to Database Systems, Ed. Pearson Education.

[Elmasri et al. 2015] Elmasri R, Navathe S B (2015). Fundamentals of Database Systems (7th ed.), Ed. Pearson.

[Fagin 1977] Fagin R (1977). Multivalued dependencies and a new normal form for relational databases, ACM Transactions on Database Systems (VLDB), v. 2, n.3, pp. 264-278.

[Fagin 1981] Fagin, R. (1981). A normal form for relational databases that is based on domains and keys, ACM Transactions on Database Systems, v. 6, no. 2, pp. 235-246.

[Fowler 2016] Fowler M (2016). Padrões de Design, Ed. Alta Books.

[Gamma et al. 1994] Gamma E, Helm R, Johnson R, Vlissides J (1994). Design Patterns: Elements of Reusable Object-Oriented Software, Ed. Addison-Wesley

[Garcia-Molina et al. 2002] Garcia-Molina, H., Elmasri, R., & Chiang, J. (2002). Database systems: Concepts and design, Ed. Pearson Education.

[Ghandeharizadeh et al. 2014] Ghandeharizadeh S, Mutha A (2014). An evaluation of the Hibernate Object-Relational Mapping for processing interactive social networking actions. doi: 10.1145/2684200.2684285

[Grassmann et al. 1996] Grassmann W K, Tremblay J-P (1996). Logic and Discrete Mathematics – a Computer Science Perspective, Ed. Prentice-Hall Inc.

[Gray et al. 2013] Gray J, Tannenbaum M (2013). Operating Systems: An Ilustrated Approach, Ed. Pearson Education.

[Jatana et al. 2012] Jatana N, Puri S, Ajuja M, Kathyrua I, Gossain D (2012). A Survey and Comparison of Relational and Non-relational Database, International Journal of Engineering Research & Technology (IJERT), v. 1, no. 6, pp. 1-5.

[Keller 1997] Keller W (1997). Mapping objects to tables: a pattern language, Wolfgang Keller 1997, 2004, pp. 01-26

[Kisman et al. 2016] Kisman I, Sani, M (2016). Hibernate ORM query simplication using hibernate criteria extension (HCE). doi: 10.1109/NICS.2016.7725656

[Kojic et al. 2015] Kojic N, Milicev D (2015). A Survey of Object Relational Transformation Patterns for High-performance UML-based Applications, In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD-2015), pp. 280-285. doi: 10.5220/0005242302800285

[Larman 2004] Larman C (2004). Utilizando UML: Guia de referência para UML 2.0, Ed. Brasport.

[Litwin et al. 1990] Litwin W, Mark L, Roussopoulos N (1990). Interoperability of multiple autonomous databases, ACM Computing Surveys, v. 22, no. 3, pp. 267-293.

[Lorenz et al. 2016] Lorenz M, Hesse G, Rudolph JP (2016). Object-relational mapping revised - a guideline review and consolidation, In: Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016), v. 1, ICSOFT-EA, pp. 157-168. doi: 10.5220/0005974201570168

[Martin 2002] Martin R C (2002). Agile Software Development, Principles, Patterns, and Practices, Ed. Prentice Hall.

[Martin 2008] Martin R C (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Ed. Prentice Hall.

[Meeker et al. 2014] Meeker W Q, Hong Y (2014). Reliability meets big data: opportunities and challenges, Quality Engineering, v. 26, no. 1, pp. 102-116, Taylor & Francis Online, doi: 10.1080/08982112.2014.846119

[O'Neil 2008] Elizabeth O'Neil. 2008. Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM). In Proceedings of the 46th Annual Southeast Regional Conference on XX. ACM, 1353–1356.

[Ogheneovo et al. 2013] Ogheneovo, E E, Asagba P O, Ogini, N O (2013). An object relational mapping, International Journal of Engineering Science Invention, v. 6, pp. 1-9.

[Ramakrishnan et al. 2003] Ramakrishnan R, Gehrke J (2003). Database Management Systems.Ed. McGraw-Hill.

[Rickerby 2015] Rickerby M (2015) The rise and fall of object relational mapping, https://maetl.net/talks/rise-and-fall-of-orm, consultado em 05/maio/2024.

[Rumbaugh 2004] Rumbaugh J, Jacobson I, Mello G (2004) Modelagem de sistemas com UML. Ed. Elsevier Brasil.

[Sedighi 1993] Sedighi S M (1993) Integration of object-oriented and relational database systems, M.Sc. thesis, Dept. Computer Science, University of the University of Wollongong, NSW, Australia.

[Stefik et al. 1985] Stefik M, Bobrow D G (1985) Object-oriented programming: themes and variations, AI Magazine, v. 6, no. 4, pp. 40-62.

[Teixeira 2017] Teixeira M H (2017) Impedância objeto relacional O atrito natural entre os dois mundos, Tecnologia em Projeção, v. 8, no. 1, pp. 11-20.

[Teorey 2014] Teorey T J (2014) Projeto de Modelagem de Banco de Dados (8ª ed.), Ed. Elsevier, Rio de Janeiro.

[Torres et al. 2017] Torres A, Galante R, Pimenta M S, Martins A, Jonatan B (2017) Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. doi:10.1016/j.infsof.2016.09.009

[Tudose et al. 2021] Tudose C, Odubăşteanu, C (2021) Object-relational Mapping Using JPA, Hibernate, and Spring Data JPA, doi: 10.1109/CSCS52396.2021.00076

[Wikipedia 2024] Relational Database, https://en.wikipedia.org/wiki/Relational_database. Consultado em 04/abril/2024

[Wikipedia 2024a] Codd's 12 rules. https://en.wikipedia.org/wiki/Codd%27s_12_rules. Consultado em 04/abril/2024

Anexo 1
As 12 Regras de Codd

Uma base de dados caracterizada como base de dados relacional (RDB) pode ser abordada de uma maneira simplista como uma base de dados que adota um modelo de dados identificado como relacional, como foi proposto em [Codd 1970] e abordado no Capítulo 2 deste documento.

O termo base de dados relacional foi definido por E.F.Codd em 1970 no artigo intitulado "A Relational Model of Data for Large Shared Data Banks" (ver referências [Codd 1970] [Codd 1985a] [Codd 1985b] [Wikipedia 2024] [Wikipedia 2024a]). A definição de uma estrutura de dados que pode ser qualificada como base de dados relacional sob a ótica de Codd é caracterizada pela satisfatibilidade das chamadas 12 regras de Codd.

Como informado em [Wikipedia-2024a], Codd teve como principal objetivo, ao criar o conjunto de regras, evitar que o conceito original de RDB fosse adulterado à medida que fornecedores de DBs tentavam promover a comercialização das bases de dados já existentes, vendendo-as como relacional. As 12 regras podem ser consideradas como parte de um procedimento de teste para determinar se um determinado produto, que se autoproclama ser completamente relacional, é realmente completamente relacional. Várias implementações comerciais do modelo relacional de base de dados, entretanto, não seguem todas as regras de Codd – como consequência, o termo tem sido gradualmente usado para descrever um vasto grupo de sistemas de DB que, no mínimo:

(1) Apresentam dados ao usuário que são representados como relações em uma representação em formato tabular *i.e.*, como uma coleção de tabelas, cada uma consistindo de um conjunto de linhas e de colunas.

(2) Fornecem operadores relacionais para manipular os dados em formato tabular.

Presentemente a definição mais comum de um RDBMS (Relational Data Base Management System) é a de ser um produto que apresenta uma visão dos dados como uma coleção de linhas e colunas, mesmo que não seja estritamente baseada na teoria relacional. Considerando essa definição, RDBMSs tipicamente implementam algumas mas não todas as 12 regras.

Codd em [Codd 1985a, 1985b] procura agrupar algumas das 12 regras, comentando que (a) as regras 8, 9,10 e 11 especificam e requerem quatro tipos diferentes de independência, com o objetivo de proteger o investimento de clientes em programas de aplicação, atividades em terminais e treinamento; (b) as regras 8 e 9 que estão relacionadas à independência física e lógica respectivamente, têm sido discutidas por muitos anos; (c) as regras 10 e 11 dizem respeito à independência de integridade e independência de distribuição, que são aspectos da abordagem relacional que receberam atenção inadequada durante os anos iniciais em que foram propostas. Codd, entretanto, previu que as regras 10 e 11 se tornariam tão importantes quanto as regras 8 e 9. O autor também informou que as regras propostas foram baseadas em uma única regra fundamental, que ele nomeou Regra Zero. Na sequência são apresentadas e discutidas as 13 regras como estabelecidas em [Codd 1985a, 1985b].

Regra 0. Regra Fundamental

Qualquer sistema que é veiculado ou proclamado ser um sistema de administração de bases de dados relacional deve ser capaz de administrar bases de dados inteiramente por meio de suas capacidades relacionais.

A Regra Zero deve ser satisfeita independentemente de o sistema lidar ou não com qualquer característica não-relacional de administrar dados. Qualquer RDBMS que não satisfizer a Regra 0 não pode ser caracterizado como um RDBMS. Possíveis consequências da Regra Zero:

(a) qualquer sistema divulgado como um RDBMS deve poder realizar inserções, atualizações e deleções na DB em nível relacional *i.e.*, *multiple-record-at-a-time* (múltiplos registros de uma vez). A execução de atividades via expressão *multiple-record-at-a-time* inclui, como casos especiais, situações em que nenhum ou apenas um registro é recuperado, inserido, atualizado ou deletado. A relação (tabela) que será discutida oportunamente pode ter 0 (zero) linhas ou uma linha e ainda assim ser uma relação válida.

(b) a necessidade de suporte à regra de informação (Regra 1) e regra de acesso garantido (Regra 2).

Como comentado anteriormente, Codd em [Codd 1985a] discute o perigo de compradores e usuários de um sistema que é comercializado como relacional e que não satisfaz a Regra 0: compradores e usuários estarão esperando pelas vantagens de um verdadeiro DBMS relacional e elas não virão. As 12 regras de Codd juntamente com as 9 estruturais, as 18 manipulativas e as 3 características de integridade do modelo relacional determinam, com detalhes, a extensão da veracidade da declaração de um vendedor ao estar vendendo um DBMS totalmente relacional.

Regra 1. Regra de Informação

Toda informação em uma base de dados relacional é representada explicitamente em nível lógico e exatamente de uma única maneira, pelos valores nas tabelas.

Até mesmo nomes de tabelas, nomes de colunas e nomes de domínios são representados como cadeias de caracteres em algumas tabelas. As tabelas que contêm esses nomes normalmente fazem parte do catálogo interno do sistema. O catálogo é, portanto, uma base de dados relacional dinâmica e ativa que representa os metadados (*i.e.*, dados que descrevem o restante dos dados no sistema). A Regra 1 é mandatória não apenas para a promover a produtividade do usuário, mas também para facilitar o trabalho de fornecedores de software quando da especificação de pacotes adicionais de software (*e.g.*, desenvolvimento de aplicativos auxiliares, sistemas especialistas, etc.) que têm interface com DBMS relacionais e que, por concepção, estão bem integrados ao DBMS relacional. Os softwares disponibilizados nos pacotes recuperam informações já existentes no catálogo e, dependendo da situação, inserem novas informações no catálogo por meio da própria utilização do DBMS relacional. O uso da Regra 1 também torna mais simples e mais eficaz a tarefa realizada pelo administrador da DB relacional, que é a de manter a DB relacional em estado de integridade geral.

Regra 2. Regra do Acesso Garantido

Garante que todo e qualquer dado (valor atômico) em uma base de dados relacional seja logicamente acessível por meio de uma combinação de nome de tabela, valor de chave primária e nome de coluna.

Cada dado em um DB relacional pode ser acessado por meio de um vasto número de diferentes expressões lógicas. É importante, entretanto, que pelo menos uma das expressões seja prevalente, independentemente da DB relacional sendo considerada, ressaltando que muitos conceitos relacionados a computador e à sua estrutura (tal como

verificação de endereços sucessivos) foram deliberadamente omitidos do modelo relacional. A Regra 1 expressa um esquema de endereçamento associativo que é específico do modelo relacional e que independe do endereçamento usual orientado a computador. No entanto, o conceito de chave primária, detalhado na Capítulo 02 é uma parte essencial do endereçamento.

Regra 3. Regra do Tratamento Sistemático de Valores Nulos

Valores nulos (distintos de uma cadeia de caracteres vazia ou de uma cadeia de caracteres em branco e distintos de zero ou de qualquer outro número) são completamente mente aceitos para representar informações ausentes e informações que não são aplicáveis de forma sistemática em DBMS totalmente relacional, independentemente do tipo de dados.

Para garantir a integridade da DB relacional deve ser possível especificar "nulos não permitidos" para cada coluna de chave primária e para quaisquer outras colunas que o administrador da DB considere uma restrição de integridade apropriada. É importante lembrar que técnicas anteriores envolviam a definição de um valor especial (peculiar a cada coluna ou campo) para representar a informação faltante. Essa prática, entretanto, resultaria em um procedimento assistemático em um DB relacional uma vez que usuários teriam que empregar técnicas diferentes para cada coluna ou domínio, o que traria dificuldades devido ao alto nível da linguagem em uso e, também, contribuiria para diminuir a produtividade do usuário.

Regra 4. Catálogo dinâmico *on-line* baseado no modelo relacional

A descrição da base de dados é representada em nível lógico da mesma forma que dados comuns, de modo que os usuários autorizados possam usar em suas interrogações a mesma linguagem relacional que eles usam em dados regulares.

O uso da Regra 4 tem duas consequências:

(1) cada usuário (seja um programador de aplicação ou um usuário final) precisa aprender apenas um modelo de dados, o que pode ser considerado uma vantagem que sistemas não relacionais geralmente não oferecem (o IMS da IBM, juntamente com seu dicionário, exige que o usuário aprenda dois modelos distintos);

(2) usuários autorizados podem facilmente estender o catálogo para torná-lo um dicionário de dados relacional ativo e completo sempre que o fornecedor não o fizer.

Regra 5. Regra da sublinguagem de dados compreensível

Um sistema relacional pode disponibilizar diversas linguagens bem como várias maneiras de uso de terminal (por exemplo, por meio do preenchimento de lacunas). No entanto, deve haver pelo menos uma linguagem que permite que instruções possam ser expressas de acordo com uma sintaxe bem definida, tal como cadeias de caracteres, e que essa linguagem seja abrangente o suficiente para que possa ser usada em vários processos:

- *Definição de dados.*
- *Definição de views.*
- *Manipulação de dados (interativa e via programa).*
- *Restrições de integridade.*
- *Autorização.*
- *Limites de transação (início, confirmação e reversão).*

A abordagem relacional é intencionalmente bem dinâmica. Raramente será necessário interromper a atividade do DB (em contraste com o SGBD não relacional). Portanto, não faz sentido separar os processos listados, usando linguagens distintas.

Regra 6. Atualização de *views*

Todas as views que teoricamente são atualizáveis são também passíveis de serem atualizadas pelo sistema.

É importante lembrar que teoricamente uma *view* é atualizável se existir um algoritmo independente de tempo que possa determinar, inequivocamente, uma única série de mudanças nas relações da base, que terão como efeito, precisamente, as mudanças solicitadas na *view*. No contexto em questão o termo "atualização" reflete operações de inserção e exclusão, bem como de modificação.

Regra 7. Operações de *insert, update* e *delete* em alto nível

A capacidade de tratar uma relação base ou uma relação derivada como sendo um único operando, se aplica não apenas à recuperação de dados, mas também às operações de inserção, atualização e exclusão de dados.

O uso da Regra 7 dá ao sistema maior abrangência na otimização da eficiência de suas ações em tempo de execução. A regra estabelece que o sistema possa determinar quais caminhos de acesso devem ser explorados para a obtenção de código mais eficiente. Também, o estabelecido pela regra pode ser de grande relevância na obtenção de um

tratamento eficiente de transações em bancos de dados distribuídos. Em um contexto distribuído os usuários prefeririam que custos de comunicação fossem poupados, evitando assim a necessidade de transmitir um pedido separado para cada registo obtido de locais remotos.

Regra 8. Independência de dados físicos

Programas aplicativos e atividades de terminal permanecem logicamente inalterados sempre que quaisquer alterações são feitas ou nas representações de armazenamento ou nos métodos de acesso.

Para lidar com programas aplicativos e atividades de terminal o SGBD deve dispor de uma fronteira clara e nítida entre (a) aspectos lógicos e semânticos, por um lado, e (b) aspectos físicos e de desempenho das tabelas da BD, por outro. Programas aplicativos devem lidar apenas com os aspectos lógicos. SGBDs não relacionais raramente fornecem suporte completo para esta regra.

Regra 9. Independência de dados lógicos

Programas aplicativos e atividades de terminal permanecem logicamente inalterados quando alterações de preservação de informações de qualquer tipo que teoricamente permitem a integridade são feitas nas tabelas base.

Considere as duas situações: (1) dividir uma tabela em duas tabelas, seja por linhas usando conteúdo de linha ou por colunas usando nomes de colunas, se as chaves primárias forem preservadas em cada resultado; (2) combinar duas tabelas em uma por meio de uma junção sem perdas (os autores da Universidade de Stanford e do MIT chamam essas junções de “sem perdas”). Para fornecer este serviço sempre que possível, o SGBD deve ser capaz de lidar com inserções, atualizações e exclusões em todas as visualizações que são teoricamente atualizáveis. Esta regra permite que o design do banco de dados lógico seja alterado dinamicamente se, por exemplo, tal alteração melhorar o desempenho.

As regras de independência de dados físicos e lógicos permitem que projetistas de DBs para DBMS relacionais cometam erros em seus projetos sem as pesadas penalidades impostas pelos DBMS não relacionais. Isso, por sua vez, significa que é muito mais fácil começar com um DBMS relacional porque não é necessário tanto planejamento orientado ao desempenho antes da “decolagem”.

Regra 10. Independência de integridade

Restrições de integridade específicas a uma determinada DB relacional devem ser passíveis de serem definidas na sub-linguagem de dados relacionais e, também, passíveis de serem armazenadas no catálogo e não nos programas aplicativos.

Além das duas regras de integridade (integridade de entidade e integridade referencial) que se aplicam a todos os DBs relacionais, fica claro que há ainda a necessidade de poder especificar restrições de integridade adicionais que reflitam políticas de negócios ou regulamentações governamentais.

Suponha que o modelo relacional tenha sido fielmente seguido e, portanto, as restrições de integridade adicionais são definidas em termos da sub-linguagem de dados de alto nível e das definições armazenadas no catálogo, e não nos programas aplicativos. Informações sobre objetos identificados inadequadamente nunca são registradas em um BD relacional. Para promover especificidade, as duas regras de integridade a seguir se aplicam a todos os DBs relacionais:

- Integridade da entidade. Nenhum componente de uma chave primária pode ter um valor nulo.
- Integridade referencial. Para cada valor de chave estrangeira não nulo distinto em um BD relacional, deve existir um valor de chave primária correspondente do mesmo domínio.

Como acontece esporadicamente de políticas empresariais ou regulamentações governamentais mudarem, provavelmente será necessário alterar as restrições de integridade. Normalmente, isso pode ser feito em um DBMS totalmente relacional, alterando uma ou mais instruções de integridade armazenadas no catálogo. Em muitos casos, nem os programas aplicativos nem as atividades do terminal são prejudicados logicamente. Raramente DBMSs não relacionais suportam essa regra como parte do mecanismo do DBMS, ao qual a regra pertence. Em vez disso, eles dependem de um pacote de dicionário, que pode ou não estar presente e que pode ser facilmente ignorado.

Regra 11. Um DBMS relacional tem independência de distribuição

Por independência de distribuição nesse contexto deve ser entendido que o DBMS possui uma sub-linguagem de dados que permite que programas aplicativos e atividades de terminal permaneçam logicamente intactos quando:

- *a distribuição de dados é implementada pela primeira vez (se o DBMS originalmente instalado gerencia apenas dados não distribuídos);*
- *os dados são redistribuídos (se o DBMS gerencia dados distribuídos).*

Observe que a definição é cuidadosamente redigida para que tanto o DBMS distribuído quanto o não distribuído possam garantir totalmente a Regra 11. É importante distinguir entre processamento distribuído e dados distribuídos. Em processamento distribuído o trabalho (por exemplo, programas) é transmitido aos dados. Já em dados distribuídos os dados são transmitidos para a realização de uma atividade. É fato que muitos DBMS não relacionais suportam processamento distribuído, mas não dados distribuídos. Os únicos sistemas que suportam o conceito de fazer com que todos os dados distribuídos pareçam locais são SGBDs relacionais.

No caso de um SGBD relacional distribuído, uma única transação pode abranger vários *sites* remotos. Esse 'espalhamento' é gerenciado inteiramente nos bastidores – o sistema pode ter que executar a recuperação em vários locais. Cada programa ou atividade de terminal trata a totalidade dos dados como se fossem todos locais do site onde o programa aplicativo ou atividade de terminal está sendo executado.

Um SGBD totalmente relacional que não suporta bancos de dados distribuídos tem a capacidade de ser estendido para fornecer esse suporte, deixando os programas aplicativos e as atividades de terminal logicamente intactas, tanto no momento da distribuição inicial quanto sempre que a redistribuição posterior for feita. Existem quatro razões importantes pelas quais um DBMS relacional desfruta dessa vantagem:

- Flexibilidade de decomposição na decisão de como implantar os dados.
- Poder de recomposição dos operadores relacionais ao combinar os resultados de subtransações executadas em diferentes *sites*.
- Economia de transmissão devido ao fato de não ser necessário enviar uma mensagem de solicitação para cada registro a ser recuperado de qualquer site remoto.
- Capacidade de análise da intenção (devido ao alto nível das linguagens relacionais) para uma otimização de execução amplamente melhorada.

Regra 12. Regra da não subversão

Se um sistema relacional tiver uma linguagem de baixo nível (registro único por vez), esse baixo nível não pode ser usado para subverter ou contornar as regras e restrições de integridade expressas na linguagem relacional de nível superior (múltiplos registros por vez).

Na abordagem relacional a preservação da integridade é independente da estrutura lógica de dados para alcançar a independência da integridade. Esta regra é extremamente difícil para um sistema "nascido de novo" obedecer porque tal sistema já suporta uma interface abaixo da interface de restrição relacional. Os fornecedores de sistemas "nascidos de novo" não parecem ter dado a devida atenção a este problema.

Anexo 2
Unifying Modeling Language (UML)

O desenvolvimento de software é um processo que envolve planejamento e, também, a administração do processo de desenvolvimento. Usualmente esse processo tem uma trajetória abrangente devido às várias fases pelas quais passa desde o seu início, que acontece na fase de levantamento de requisitos do sistema, até a sua fase final, que envolve implantação e testes.

Com o objetivo de promover a qualidade do software a ser desenvolvido e evitar problemas durante o processo de desenvolvimento é mandatório que a modelagem das especificações do software considerado seja realizada sempre acompanhada de uma documentação atualizada.

O uso da linguagem gráfica UML (*Unified Modeling Language*) [Booch *et al.* 1996] vem ao encontro da concretização dos cuidados a serem tomados para que problemas possam ser evitados durante o processo de desenvolvimento de sistemas computacionais orientados a objetos. UML é uma linguagem pesadamente gráfica que dispõe de um conjunto razoavelmente grande de elementos gráficos com diferentes caracterizações semânticas, que usados para representar variados componentes de um sistema computacional.

A representação gráfica de um sistema por meio de UML desempenha um importante papel, auxiliando desenvolvedores a visualizarem as inúmeras perspectivas funcionais das partes que compõem o sistema computacional sendo desenvolvido e, consequentemente, colaboram na promoção de uma compreensão mais apurada, por parte de desenvolvedores, de seu funcionamento. A linguagem gráfica inclui elementos tais como:

- Atividades (Jobs)
- Componentes individuais do sistema e como podem interagir com outros componentes de software
- Como o sistema será executado
- Como entidades interagem com outras(componentes e interfaces)
- Interface de usuário externa

Embora originalmente a linguagem UML tenha sido criada para documentar o projeto de sistemas computacionais orientados a objetos, em várias situações a linguagem tem sido estendida para abranger um conjunto maior de documentação de outros tipos de projetos.

A2.1 Modelagem

Modelos e Diagramas são os conceitos fundamentais empregados no desenvolvimento da UML, que foi criada com o propósito de representar e documentar o desenvolvimento de aplicações computacionais. Em um ambiente de desenvolvimento de software, modelos colaboram para disponibilizar uma visualização abstrata do sistema em desenvolvimento e diagramas colaboram na representação concreta do sistema.

A UML é bem versátil uma vez que possibilita a modelagem de sistemas computacionais em diferentes níveis de detalhe. A modelagem de um projeto de software pode ser organizada e realizada por meio do uso de diagramas, modelos e de perfis. Diagramas de modelagem podem ser usados para detectar casos de uso de um sistema em modelos de caso de uso.

O uso de modelos pode ser conveniente quando (a) existe interesse em representar visualmente uma proposta de sistema a ser desenvolvido, (b) disponibilizar a representação de um sistema para discussão entre pares e/ou apresentação ao cliente, (c) desenvolver, programar e testar um sistema em construção e (d) usar diagramas UML para a criação de código.

A2.2 Diagrama

É importante discernir entre o modelo UML e o conjunto de diagramas disponibilizado pela linguagem. Um diagrama é a representação parcial do modelo do sistema. O conjunto de diagramas não necessita refletir completamente o modelo e a remoção de um diagrama não muda o modelo. O modelo pode também conter documentação que direciona os elementos do modelo e os diagramas (tal como escrito em use cases).

A ULM 2 oferece muitos tipos de diagramas que são apresentados divididos em duas categorias. Alguns tipos representam informação estrutural e o resto representa tipos gerais de comportamento, incluindo alguns poucos que representam aspectos diferentes de interações. Esses diagramas são categorizados e organizados hierarquicamente como apresentado na sequência e mostrados nas Figura A2.1, Figura A2.2 e Figura A2.3.

1. View Estrutural (ou Estática) que enfatiza a estrutura estática do sistema usando objetos, atributos, operações e relações. Inclui diagramas de classe e diagramas de estrutura composta.

1.1 Classe: caracteriza um conjunto de informações, comportamentos e relacionamentos que descrevem uma entidade

1.2 Interface: caracteriza a parte visível de uma classe. Normalmente utilizada para descrever as assinaturas públicas de operações em uma classe.

1.3 Componente: uma unidade física de software que pode residir na memória de um processador que implementa um conjunto de interfaces.

2.View Comportamental (ou Dinâmica) que enfatiza o comportamento dinâmico do sistema, evidenciados colaborações entre objetos e mudanças nos estados internos de objetos. Inclui: diagramas de sequencia, diagramas de atividade e diagramas de estado da máquina.

2.1 Interações: comunicações entre objetos.

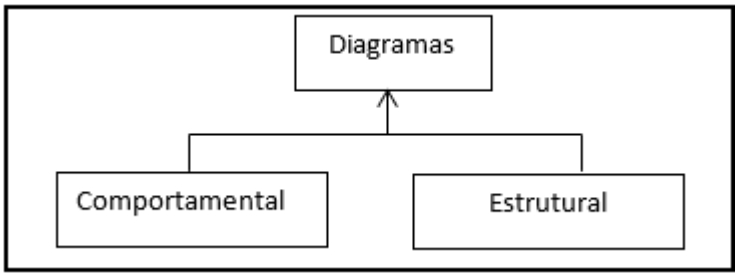

Figura A2.1 Diagramas são divididos em duas categorias: Comportamental (ver Figura A2.2) e Estrutural (ver Figura A2.3).

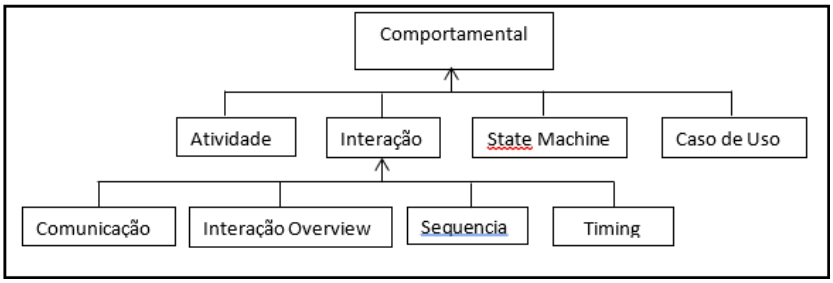

Figura A2.2 Diagramas da categoria Comportamental organizados hierarquicamente.

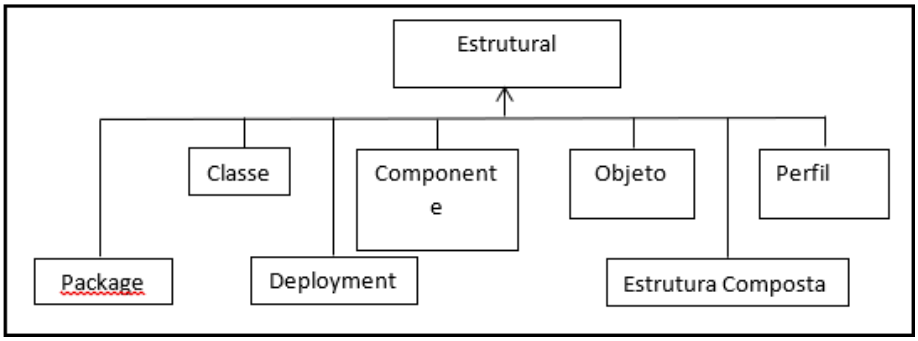

Figura A2.3 Diagramas da categoria Estrutural.

Anexo 3
Padrões de Desenvolvimento de Software

O desenvolvimento de software é uma disciplina complexa que exige uma combinação de conhecimentos técnicos, boas práticas e padrões bem estabelecidos. Entre os conceitos mais influentes e amplamente adotados na indústria, destacam-se os padrões de projeto do Gang of Four (GoF), os princípios SOLID, as práticas de código limpo (Clean Code) e o padrão de arquitetura MVC.

A3.1 Gang of Four (GoF)

Os padrões de projeto do Gang of Four [Gamma *et al.* 1994], são fundamentais para a programação orientada a objetos. Os autores identificam 23 padrões de projeto que resolvem problemas recorrentes em design de software, divididos em suas respectivas categorias: padrões criacionais, estruturais e comportamentais.

Padrões Criacionais:

1. **Abstract Factory**: Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Útil para garantir a compatibilidade entre produtos.
2. **Builder**: Separa a construção de um objeto complexo da sua representação, permitindo a criação passo a passo. É útil quando a criação do objeto envolve muitas etapas.
3. **Factory Method**: Define uma interface para criar um objeto, mas deixa para as subclasses a decisão de qual classe instanciar. Promove a reutilização de código ao encapsular a lógica de criação.
4. **Prototype**: Permite a criação de novos objetos copiando um objeto existente (protótipo). É útil quando a criação direta do objeto é complexa ou custosa.
5. **Singleton**: Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela. Útil para recursos que precisam ser compartilhados, como conexões de banco de dados.

Padrões Estruturais:

6. **Adapter**: Permite que interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra esperada pelos clientes. É útil para integrar classes com interfaces incompatíveis.
7. **Bridge**: Separa a abstração da implementação, permitindo que ambas variem independentemente. Útil quando as abstrações e implementações devem ser extensíveis por subclasses.
8. **Composite**: Compõe objetos em estruturas de árvore para representar hierarquias parte-todo, permitindo que clientes tratem objetos individuais e composições de objetos de maneira uniforme.
9. **Decorator**: Adiciona responsabilidades a um objeto dinamicamente, sem modificar sua estrutura. Útil para adicionar funcionalidades a objetos de forma flexível e escalável.
10. **Facade**: Fornece uma interface simplificada para um conjunto complexo de interfaces de um subsistema, facilitando o uso do subsistema pelo cliente.
11. **Flyweight**: Reduz a sobrecarga de criação de um grande número de objetos semelhantes, compartilhando o máximo de dados possível entre eles. Útil para melhorar a eficiência em sistemas com muitos objetos semelhantes.
12. **Proxy**: Fornece um substituto ou marcador para outro objeto para controlar o acesso a ele. Útil para adicionar controle de acesso, carregar objetos sob demanda, entre outros.

Padrões Comportamentais:

13. **Chain of Responsibility**: Passa uma solicitação ao longo de uma cadeia de handlers, onde cada handler decide processar a solicitação ou passá-la adiante. Útil para desacoplar o remetente do receptor da solicitação.
14. **Command**: Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, filas ou registros de solicitações. Útil para implementação de operações reversíveis e filas de operações.
15. **Interpreter**: Define uma gramática para uma linguagem e usa um intérprete para interpretar sentenças dessa linguagem. Útil para implementar linguagens de domínio específico.
16. **Iterator**: Fornece uma maneira de acessar sequencialmente os elementos de uma coleção sem expor sua representação subjacente. Útil para percorrer coleções de maneira uniforme.

17. **Mediator**: Define um objeto que encapsula a forma como um conjunto de objetos interage, promovendo um acoplamento fraco e centralizando a comunicação. Útil para reduzir interdependências complexas entre objetos.
18. **Memento**: Permite capturar e restaurar o estado interno de um objeto sem violar seu encapsulamento. Útil para implementar funcionalidades de desfazer operações.
19. **Observer**: Define uma dependência um-para-muitos entre objetos, de forma que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. Útil para implementar notificações.
20. **State**: Permite que um objeto altere seu comportamento quando seu estado interno muda. Útil para implementar máquinas de estado.
21. **Strategy**: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.
22. **Template Method**: Define o esqueleto de um algoritmo em uma operação, deixando alguns passos para subclasses implementarem. Útil para definir a estrutura de um algoritmo de forma extensível.
23. **Visitor**: Representa uma operação a ser executada nos elementos de uma estrutura de objeto, permitindo definir novas operações sem mudar as classes dos elementos nos quais opera. Útil para adicionar operações a estruturas de objetos complexas.

A3.2 SOLID

Os princípios SOLID são um conjunto de cinco diretrizes destinadas a promover um design de software mais flexível e fácil de manter, também conhecido como *Uncle Bob*, esses princípios são descritos por [Martin 2002].

1. **Single Responsibility Principle (SRP)**: Uma classe deve ter apenas uma razão para mudar, ou seja, deve ter uma única responsabilidade.
2. **Open/Closed Principle (OCP)**: Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação.
3. **Liskov Substitution Principle (LSP)**: Objetos de uma classe derivada devem poder substituir objetos da classe base sem alterar o comportamento desejado.
4. **Interface Segregation Principle (ISP)**: Múltiplas interfaces específicas são melhores do que uma única interface geral.
5. **Dependency Inversion Principle (DIP)**: Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

A3.3 Clean Code

O conceito de código limpo, O autor [Martin 2008] enfatiza a importância de escrever código que seja fácil de ler, entender e manter. Ele propõe várias práticas e princípios para alcançar, seguem algumas:

Nomes Significativos:

- Escolher nomes claros e descritivos para variáveis, funções, classes e outros elementos do código é crucial para a legibilidade.
- Nomes devem comunicar a intenção. Por exemplo, calculaSalarioAnual é mais descritivo que calcSal.

Funções Pequenas e Coesas:

- Funções devem ser pequenas, realizando apenas uma tarefa específica e claramente definida.
- Seguir o princípio do *Single Responsibility Principle* (SRP), onde cada função tem uma única responsabilidade ou motivo para mudar.

Evitar Comentários Desnecessários:

- O código deve ser autoexplicativo, reduzindo a necessidade de comentários.
- Comentários podem se tornar obsoletos e enganosos se não forem mantidos adequadamente. Utilize-os para explicar *por quê*, não *o quê*.

Formatação Consistente:

- Consistência na formatação, como indentação e espaçamento, melhora a legibilidade.
- Seguir convenções de código e padrões da equipe ou projeto.

Tratamento de Erros:

- Erros devem ser tratados adequadamente, preferencialmente utilizando exceções em vez de códigos de erro.
- Evitar o uso de estruturas complexas de tratamento de erros, mantendo o código claro e legível.

A3.4 MVC (Model-View-Controller)

O padrão de arquitetura MVC, inicialmente descrito na década de 1970 para a linguagem de programação *Smalltalk*. O livro *Design Patterns* do GoF também menciona o MVC como um exemplo de padrão arquitetural. O padrão de arquitetura MVC é uma abordagem amplamente adotada para estruturar aplicações de software, especialmente aquelas que

possuem uma interface gráfica de usuário (GUI). O MVC divide uma aplicação em três componentes interconectados principais: Model, View e Controller. Essa separação de responsabilidades facilita a manutenção, a escalabilidade e a reutilização do código.

Componentes do MVC

1. **Model (Modelo)**:
    - **Responsabilidade**: O Model representa a lógica de negócios e os dados da aplicação. Ele é responsável por gerenciar o estado da aplicação, acesso a dados e regras de negócios.
    - **Comunicação**: O Model notifica a View sobre as mudanças nos dados e fornece os dados solicitados pelo Controller.
    - **Exemplo**: Em uma aplicação de e-commerce, o Model pode incluir classes como Produto, Carrinho, Pedido, que contêm a lógica para manipular os dados desses elementos.
2. **View (Visão)**:
    - **Responsabilidade**: A View é responsável pela apresentação dos dados ao usuário. Ela define a estrutura visual e a interface de usuário.
    - **Comunicação**: A View obtém os dados do Model para exibi-los e envia as interações do usuário ao Controller.
    - **Exemplo**: Em uma aplicação web, a View pode ser representada por páginas HTML, CSS e JavaScript, ou templates em frameworks como Angular, React ou Vue.js.
3. **Controller (Controlador)**:
    - **Responsabilidade**: O Controller atua como um intermediário entre a View e o Model. Ele recebe as entradas do usuário através da View, processa essas entradas (interagindo com o Model) e retorna o resultado para a View.
    - **Comunicação**: O Controller invoca métodos do Model para alterar seu estado ou recuperar dados e seleciona a View apropriada para renderizar a resposta.
    - **Exemplo**: Em uma aplicação de e-commerce, um Controller pode gerenciar ações como adicionar um produto ao carrinho, finalizar uma compra ou atualizar o perfil do usuário.

Benefícios do MVC

1. **Separação de Preocupações**: Cada componente tem uma responsabilidade claramente definida, o que facilita a manutenção e o desenvolvimento paralelo.
2. **Reutilização de Código**: A lógica de negócios (Model) é independente da interface de usuário (View), permitindo a reutilização em diferentes interfaces.
3. **Facilidade de Teste**: A separação facilita a escrita de testes unitários e de integração, especialmente para a lógica de negócios no Model e a lógica de controle no Controller.
4. **Manutenção e Extensibilidade**: Alterações na interface de usuário podem ser feitas independentemente da lógica de negócios, e vice-versa, tornando a aplicação mais fácil de manter e expandir.

A3.5 DAO (Data Access Object)

O padrão DAO (Data Access Object) é uma prática de design fundamental no desenvolvimento de software, reconhecida por sua capacidade de separar a lógica de acesso aos dados da lógica de negócios em aplicações orientadas a objetos. O conceito de DAO foi formalmente apresentado [Gamma *et all.* 1994].

No contexto do padrão DAO, a principal preocupação é separar as operações de acesso aos dados das operações de manipulação de dados e lógica de negócios. Isso é alcançado encapsulando operações CRUD (*Create, Read, Update, Delete*) em objetos DAO dedicados, que oferecem uma interface abstrata para interação com diferentes fontes de dados, como bancos de dados relacionais.

O padrão DAO continua sendo uma peça fundamental na caixa de ferramentas de desenvolvedores de software, proporcionando uma abstração eficaz para o acesso a dados e promovendo boas práticas de arquitetura. Sua origem nos princípios estabelecidos pelos "Gang of Four" e sua evolução para aplicações modernas demonstram sua relevância contínua no desenvolvimento de software orientado a objetos.

Em suma, ao implementar o padrão DAO, os desenvolvedores não apenas melhoram a estrutura e a manutenibilidade de suas aplicações, mas também adotam uma abordagem robusta para lidar com a complexidade do acesso a dados em ambientes de software sofisticados e em constante evolução.

Benefícios do Padrão DAO

A adoção do padrão DAO traz uma série de benefícios significativos para o desenvolvimento de software:

1. **Separação de Responsabilidades:** Ao isolar o acesso aos dados em objetos DAO, a lógica de negócios torna-se mais limpa e focada em suas responsabilidades principais, promovendo uma arquitetura mais modular e de fácil manutenção.
2. **Reutilização de Código:** Os objetos DAO encapsulam operações comuns de acesso a dados, facilitando a reutilização do código em toda a aplicação. Isso reduz a duplicação de código e promove uma implementação consistente de operações de acesso aos dados.
3. **Facilidade de Testabilidade:** Por permitir a substituição fácil de implementações DAO (por exemplo, para testes unitários), o padrão DAO melhora a testabilidade do código. Testes podem ser realizados de forma mais eficaz e sem depender diretamente de infraestruturas externas, como bancos de dados.

Aplicações Modernas e Frameworks

Desde sua introdução, o padrão DAO tem sido amplamente adotado em frameworks modernos de desenvolvimento de software. Por exemplo, o framework Hibernate para Java utiliza o padrão DAO como parte de suas práticas recomendadas para interação com bancos de dados relacionais. Da mesma forma, o Spring Framework oferece suporte robusto para implementação de DAOs, promovendo a modularidade e a escalabilidade em aplicações empresariais.

Anexo 4 e 5
Script DDL e DML &
Código Fonte do Padrão ORM-DAO

Script DDL e DML

https://github.com/dmarcal/mestrado/tree/main/Anexo4

Código Fonte do Padrão ORM-DAO

https://github.com/dmarcal/mestrado/tree/main/Anexo5