Trabalho Prático - Fase 3

Gerenciamento de Estoque com Índices em Arquivos

Integrantes:

- Arthur Braga de Campos Tinoco
- Rafael Lima Mendonça Garcia

PUC Minas - Unidade Praça da Liberdade

AEDS 3

---

1. Documentação do Projeto (Fase 1 - Refinada)

1.1. Descrição do Problema

O sistema proposto visa permitir o cadastro e gerenciamento de Itens em Estoque, Fornecedores e Categorias. A solução deve possibilitar o acompanhamento da quantidade de itens comprados e vendidos, com todos os dados sendo armazenados em arquivos binários que incluem um cabeçalho para controle de metadados e um sistema de exclusão lógica por lápide.

1.2. Objetivo do Trabalho

- Desenvolver um sistema que permita as operações de CRUD (Create, Read, Update, Delete) para as entidades de estoque, fornecedores e categorias.
- Garantir a persistência dos dados em arquivos binários com controle de exclusão lógica.
- Fornecer documentação técnica completa, contendo Diagrama de Caso de Uso (DCU), Diagrama Entidade-Relacionamento (DER) e a Arquitetura Proposta.

1.3. Requisitos Funcionais

- RF01: Gerenciar Categorias (CRUD completo).
- RF02: Gerenciar Fornecedores (CRUD completo).
- RF03: Gerenciar Itens em Estoque (CRUD completo).
- RF04: Registrar Movimentações de Estoque (entradas e saídas) - *escopo para fases futuras.*

1.4. Requisitos Não Funcionais

- RNF01: A persistência de dados deve ser realizada diretamente em arquivos binários, sem o uso de um SGBD.
- RNF02: Os arquivos de dados devem possuir um cabeçalho para armazenamento de metadados (ex: último ID utilizado).
- RNF03: A exclusão de registros deve ser implementada de forma lógica (lápide), sem remoção física dos dados.
- RNF04: O sistema deve seguir a arquitetura MVC + DAO.
- RNF05: A interface com o usuário para esta fase do projeto será via console.

1.5. Atores do Sistema

- **Administrador:** Responsável por cadastrar, editar, excluir e consultar categorias, fornecedores e itens em estoque.
- **Cliente/Operador:** Pode apenas consultar os itens disponíveis no sistema.

1.6. Diagrama de Caso de Uso

O diagrama a seguir ilustra as funcionalidades do sistema e as interações dos atores. Os casos de uso foram nomeados com verbos no infinitivo, conforme as boas práticas de modelagem.

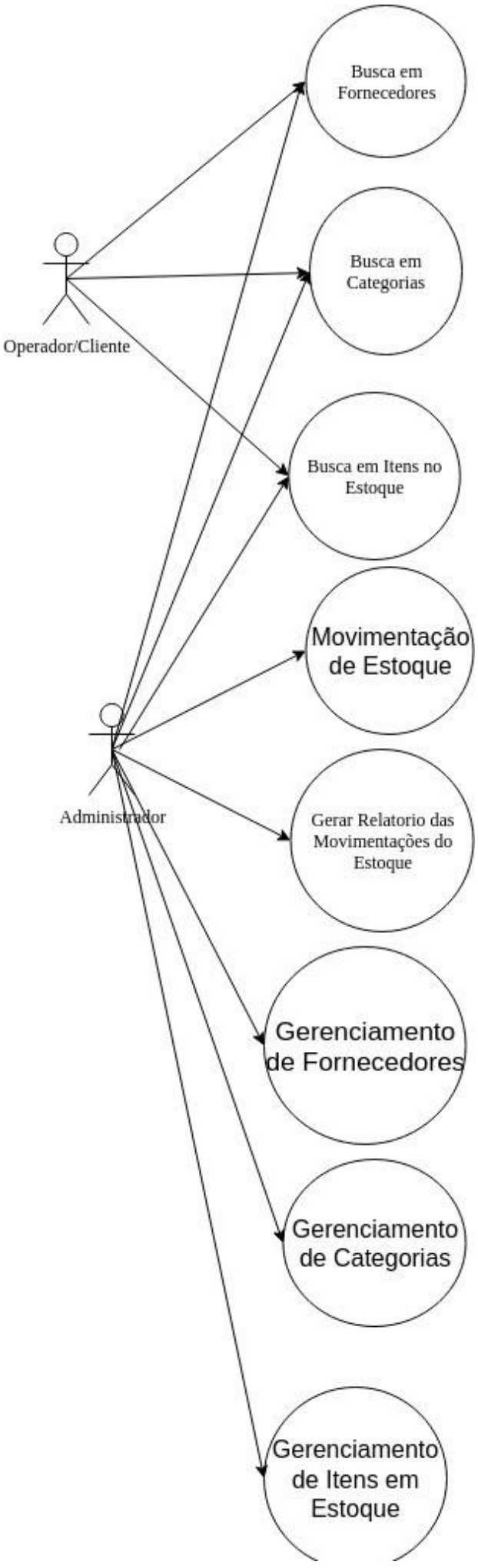
Busca em Fornecedores
Busca em Categorias
Busca em Itens no Estoque
Movimentação de Estoque
Gerar Relatorio das Movimentações do Estoque
Gerenciamento de Fornecedores
Gerenciamento de Categorias
Gerenciamento de Itens em Estoque
Operador/Cliente
Administrador

Figura 1 - Diagrama de Caso de Uso do Sistema de Estoque.

1.7. Diagrama Entidade-Relacionamento (DER)

O DER conceitual abaixo modela as entidades principais do sistema e seus relacionamentos.

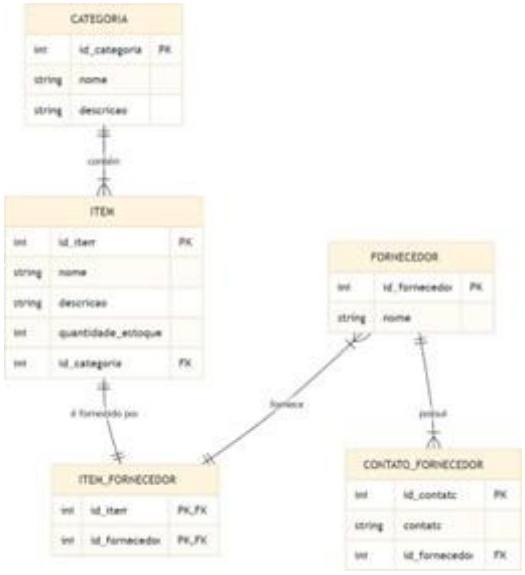

Figura 2 - Diagrama Entidade-Relacionamento Conceitual.

1.8. Arquitetura Proposta

O sistema foi estruturado seguindo o padrão arquitetural MVC + DAO:

- **Model:** Contém as classes de domínio (Categoria, Fornecedor, ItemEstoque).
- **View:** Contém a classe MainView, responsável pela interface com o usuário via console.
- **DAO (Data Access Object):** Camada de acesso aos dados, responsável pela manipulação dos arquivos binários e pela interação com as estruturas de índice.

---

2. Formulário de Projeto – Decisões Técnicas (Fase 2)

a) Qual a estrutura usada para representar os registros?

A persistência dos dados foi implementada utilizando arquivos binários de acesso aleatório (RandomAccessFile). Cada arquivo de entidade (ex: categorias.db) possui um cabeçalho de 4 bytes que armazena o último ID utilizado, garantindo o controle sequencial dos identificadores. Cada registro no arquivo segue a estrutura: [Lápide (1 byte)] [Tamanho do Registro (int - 4 bytes)] [Dados Serializados (N bytes)]. A lápide (' ') indica um registro ativo e ('*') um registro logicamente excluído. O tamanho do registro é gravado antes dos dados para permitir pular registros de tamanho variável de forma eficiente.

b) Como atributos multivalorados do tipo string foram tratados?

O atributo multivalorado de telefones na entidade Fornecedor foi tratado convertendo a lista de Strings (List<String>) em uma única String concatenada, utilizando um caractere delimitador (;). No método toByteArray, String.join(";") é usado para criar a string única antes da gravação. No método fromByteArray, o método split(";") é utilizado para reconstruir a ArrayList de telefones a partir da string lida do arquivo.

c) Como foi implementada a exclusão lógica?

A exclusão lógica foi implementada através de uma "lápide", que é o primeiro byte de cada registro no arquivo de dados. Quando um registro é criado, este byte é gravado com o valor de espaço (' '). Quando o método delete de um DAO é chamado, ele localiza o registro (usando o índice primário) e sobrescreve apenas o byte da lápide com um asterisco ('*'). As operações de leitura são programadas para ignorar qualquer registro que comece com o caractere de lápide. Além disso, a chave do registro excluído é removida do índice primário (Hash Extensível) para que não seja mais encontrada em buscas diretas.

d) Além das PKs, quais outras chaves foram utilizadas nesta etapa?

Além das chaves primárias (PKs) de todas as tabelas, foi utilizada a chave estrangeira (FK) idCategoria na tabela ItemEstoque. Esta chave estabelece o relacionamento 1:N, onde uma Categoria pode ter N Itens de Estoque.

e) Quais tipos de estruturas foram utilizadas para cada chave de pesquisa?

- **Para todas as Chaves Primárias (PKs):** Foi implementado um índice de **Hash Extensível**. Esta estrutura foi escolhida por sua alta eficiência em buscas diretas por chave (complexidade O(1) em média), ideal para operações de read, update e delete baseadas em um ID específico.

- **Para a Chave Estrangeira idCategoria:** Foi implementado um índice secundário utilizando uma **Árvore B+**. Esta estrutura foi escolhida por ser extremamente eficiente em buscas por faixa e por agrupar chaves iguais, permitindo recuperar rapidamente todos os registros ('N') associados a uma chave específica ('1'), o que é a exata definição da busca no relacionamento 1:N.

f) Como foi implementado o relacionamento 1:N?

O relacionamento 1:N entre Categoria e ItemEstoque foi implementado através de um índice secundário de Árvore B+ sobre a chave estrangeira idCategoria na tabela de ItemEstoque. A navegação funciona da seguinte forma: para listar todos os itens de uma categoria, o sistema consulta a Árvore B+ com o idCategoria desejado. A árvore retorna uma lista de todos os endereços de disco (ponteiros) para os registros de ItemEstoque que possuem aquele idCategoria. O DAO então percorre essa lista de endereços, acessando diretamente cada registro no arquivo de dados (itens_estoque.db) sem a necessidade de uma varredura sequencial. A integridade referencial é mantida no nível da aplicação: antes de criar um ItemEstoque, a MainView utiliza o método read do CategoriaDAO e FornecedorDAO para verificar se os IDs da categoria e do fornecedor informados são válidos.

g) Como os índices são persistidos em disco?

Cada estrutura de índice gerencia seus próprios arquivos binários.

- O **Hash Extensível** utiliza dois arquivos: um para o diretório (_dir.db), que armazena a profundidade global e a lista de ponteiros para os cestos; e outro para os cestos (_cestos.db), que armazena os próprios cestos (profundidade local e os pares chave/endereço).

- A Árvore B+ utiliza um único arquivo (_bplus.db) que contém um cabeçalho com o ponteiro para a página raiz, seguido pelas páginas (nós) da árvore serializadas.

A sincronização é imediata: a cada operação de create, update ou delete no DAO, a estrutura do índice em memória é modificada e, em seguida, as alterações são gravadas diretamente nos arquivos de índice correspondentes no disco.

h) Como está estruturado o projeto no GitHub?

O projeto está estruturado seguindo o padrão arquitetural MVC + DAO, organizado em pacotes Java distintos dentro da pasta src/:

- /src/model: Contém as classes de domínio (Categoria, Fornecedor, ItemEstoque).

- /src/dao: Contém as classes de acesso a dados, responsáveis pela manipulação dos arquivos binários e pela interação com os índices.
- /src/view: Contém a classe de interface com o usuário via console (MainView).
- /src/app: Contém a classe Main, que é o ponto de entrada da aplicação.
- /src/indices: Contém as implementações das estruturas de dados de indexação (HashExtensível e Árvore B+).
- /data: Diretório na raiz do projeto onde todos os arquivos de dados (.db) e de índices são criados e mantidos.

---

3. Formulário de Projeto – Decisões Técnicas (Fase 3)

1. Qual foi o relacionamento N:N escolhido e quais tabelas ele conecta?

O relacionamento N:N escolhido foi entre as entidades **Fornecedor** e **Categoria**. Este relacionamento é implementado através de uma nova tabela associativa (intermediária) chamada **FornecedorCategoria**.

Essa escolha foi feita por ser a mais lógica e limpa, permitindo que um fornecedor possa fornecer itens de múltiplas categorias (ex: "Distribuidora ABC" fornece Laticínios e Limpeza) e que uma categoria possa ser fornecida por múltiplos fornecedores, sem alterar a estrutura 1:N já existente entre ItemEstoque e Fornecedor.

2. Qual estrutura de índice foi utilizada (B+ ou Hash Extensível)? Justifique a escolha.

Foi utilizada a **Árvore B+** (B+ Tree).

Justificativa: A principal função de um índice em um relacionamento N:N é responder a perguntas de *agrupamento* (1-para-N), como "Quais são *todas* as categorias do Fornecedor X?" ou "Quais são *todos* os fornecedores da Categoria Y?".

- O **Hash Extensível** é otimizado para buscas de ponto (1-para-1). Ele é ineficiente para buscas de grupo, pois encontrar todos os registros com a mesma chave exigiria uma varredura no índice.
- A **Árvore B+** é a ferramenta correta, pois ela armazena chaves de forma ordenada e agrupada. Ao indexar por idFornecedor, todos os registros do "Fornecedor 1" ficam juntos nas folhas da árvore, permitindo-nos encontrar o primeiro e percorrer a lista ligada de folhas para recuperar todos os registros associados de forma eficiente.

3. Como foi implementada a chave composta da tabela intermediária?

A chave primária composta (idFornecedor, idCategoria) é implementada **logicamente** dentro da nova entidade model/FornecedorCategoria.java. Esta classe armazena ambos os IDs como seus únicos atributos. A unicidade do par é validada no método create do FornecedorCategoriaDAO através de uma chamada ao método read(idFornecedor, idCategoria), que realiza uma varredura para garantir que o vínculo não exista antes de ser criado.

4. Como é feita a busca eficiente de registros por meio do índice?

A busca eficiente (ex: readAllByIdFornecedor) é realizada através da **Árvore B+** indexada pelo idFornecedor. O fluxo é o seguinte:

1. O método fcDAO.readAllByIdFornecedor(id) é chamado.
2. Internamente, ele chama o método indicePorFornecedor.readAll(id).

3. A Árvore B+ desce até a folha correta e coleta todos os **endereços de disco (ponteiros)** associados àquela chave idFornecedor, percorrendo a lista ligada de folhas.

4. O DAO recebe uma List<Long> (lista de endereços) de volta.

5. O DAO itera por essa lista, usando raf.seek(endereco) para pular diretamente para a posição de cada registro no arquivo fornecedor_categoria.db, lendo apenas os registros válidos (lápide ' ').

5. Como o sistema trata a integridade referencial (remoção/atualização)?

A integridade referencial é mantida no nível da **aplicação** (na MainView):

- **Criação:** Antes de o método vincularFornecedorCategoria() ser executado, o sistema chama fornecedorDAO.read(idF) e categoriaDAO.read(idC). O vínculo N:N só é criado se ambas as entidades principais forem encontradas, evitando "vínculos órfãos".

- **Remoção:** A remoção do vínculo é feita na tabela intermediária. Se um Fornecedor ou Categoria for deletado, os vínculos N:N se tornam órfãos, mas nas buscas de N:N (ex: listarCategoriasPorFornecedor), o sistema tenta buscar a Categoria pelo ID; se ela não for encontrada (pois foi deletada), ela simplesmente não é exibida na lista.

6. Como foi organizada a persistência dos dados dessa nova tabela?

A persistência da FornecedorCategoria segue **exatamente o mesmo padrão** das tabelas anteriores:

- **Arquivo:** data/fornecedor_categoria.db.

- **Cabeçalho:** Um inteiro de 4 bytes na posição 0, que armazena a contagem total de relacionamentos ativos.

- **Registros:** Cada registro segue o formato [Lápide (1 byte)] [Tamanho do Registro (int - 4 bytes)] [Dados Serializados (N bytes)].

- **Dados:** Os dados serializados consistem em [idFornecedor (int)] [idCategoria (int)].

7. Descreva como o código da tabela intermediária se integra com o CRUD das tabelas principais.

A integração ocorre na MainView e no FornecedorCategoriaDAO:

1. Um novo DAO, FornecedorCategoriaDAO (ou fcDAO), foi criado e instanciado na MainView.

2. Um novo menu, "Relacionar Fornecedor/Categoria (N:N)", foi adicionado à MainView.

3. Este menu chama os métodos do fcDAO (ex: create, delete).

4. Para exibir os resultados de forma amigável (ex: "Listar Categorias de um Fornecedor"), o método listarCategoriasPorFornecedor na MainView primeiro usa o fcDAO para obter os IDs das categorias e, em seguida, usa o **categoriaDAO.read(id)** para buscar o nome e a descrição de cada categoria.

8. Descreva como está organizada a estrutura de diretórios e módulos no repositório após esta fase.

A estrutura de diretórios da Fase 2 foi mantida, provando sua escalabilidade. A nova funcionalidade foi integrada apenas adicionando novos arquivos aos pacotes existentes:

- src/model/FornecedorCategoria.java: Novo arquivo adicionado.

- src/dao/FornecedorCategoriaDAO.java: Novo arquivo adicionado.
- src/view/MainView.java: Arquivo existente que foi *atualizado* para incluir o novo menu e as chamadas ao novo DAO.
- data/: Agora contém os novos arquivos de dados e índice: fornecedor_categoria.db e fc_idx_fornecedor_bplus.db.

---

4. Link para o Código-Fonte

O código-fonte completo do projeto, incluindo as instruções de compilação e execução no arquivo README.md atualizado, está disponível no seguinte repositório GitHub:

Braga451/AEDSIII-Trabalho at TP2
