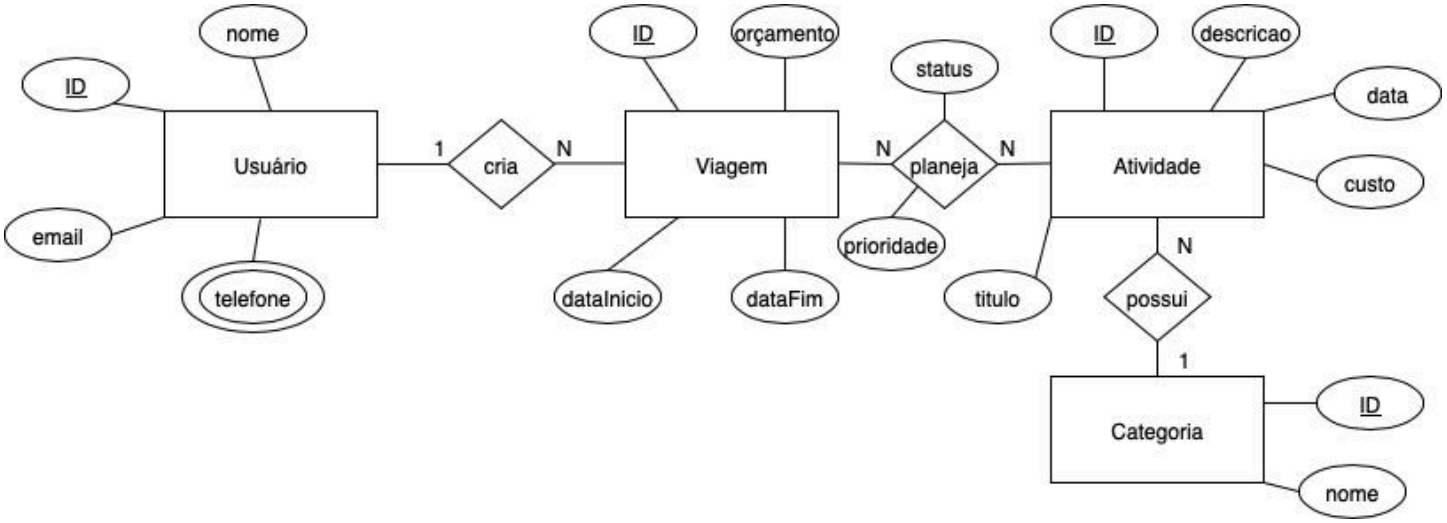
Pontifícia Universidade Católica de Minas Gerais Instituto de Ciências Exatas e Informática

Nome: Gabriel Henrique Vieira de Oliveira, Vinicius Cezar Pereira Menezes
Data: 09/11/2025
Matéria: Algoritmos e Estruturas de Dados 3
Link Github: https://github.com/GabrielHenrique20/TP_AEDS3

Formulário

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

Em nosso projeto, há a presença de apenas um relacionamento N:N, que seria entre as tabelas/CRUDS de Viagem e Atividade, onde várias viagens podem ter várias atividades e vice-versa, como descrito no Diagrama de Entidade e Relacionamento abaixo.



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

A estrutura de índice utilizada para implementar a relação N:N entre Viagem <-> Atividade foi a Árvore B +. Essa escolha permite armazenar chaves ordenadas e percorrer facilmente todos os vínculos de uma Viagem ou Atividade, o que é essencial em um relacionamento como esse, onde há múltiplas associações para a mesma chave.

```
// Atributos
private Arquivo<RelViagemAtividade> arqRel;
private ArvoreBMais<ParIntInt> ixViagemRel;
private ArvoreBMais<ParIntInt> ixAtividadeRel;

private ViagemDAO viagemDAO;
private AtividadeDAO atividadeDAO;
```

```
// Construtor
public RelViagemAtividadeDAO() throws Exception {
    arqRel = new Arquivo<>(nomeArquivo: "RelViagemAtividade", RelViagemAtividade.class.getConstructor());
    ixViagemRel = new ArvoreBMais<>(ParIntInt.class.getConstructor(), o: 4, na: "relViagemAtividade.db");
    ixAtividadeRel = new ArvoreBMais<>(ParIntInt.class.getConstructor(), o: 4, na: "relAtividadeViagem.db");
    viagemDAO = new ViagemDAO();
    atividadeDAO = new AtividadeDAO();
}

// Métodos CRUD e listagens específicas
```

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

A implementação da chave composta na tabela intermediária foi feita usando uma abordagem híbrida:

Identificador Único (PK): Um ID que se incrementa é usado como chave primária física do registro da tabela intermediária (RelViagemAtividade). Isso simplifica as operações CRUD, já que cada vínculo tem um identificador próprio.

```
// Chave primária composta: (idViagem, idAtividade)
public class RelViagemAtividade implements Registro {
    private int id; // PK do registro da relação
    private int idViagem; // FK
    private int idAtividade; // FK
    private String status; // atributo do relacionamento
    private int prioridade; // atributo do relacionamento
```

Chave Composta Lógica: A chave composta lógica é formada pelo par (idViagem, idAtividade). Esse par representa a conexão exclusiva entre uma viagem e uma atividade, garantindo que não haja duplicações.

Índices na DAO: A DAO mantém dois índices B +, um para idViagem e outro para idAtividade, o que permite acessar os vínculos por qualquer um dos lados do relacionamento de forma eficiente.

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

A busca eficiente dos registros é feita por meio dos dois índices B + mantidos na DAO (ixViagemRel e ixAtividadeRel). Cada índice permite acessar rapidamente todos os vínculos associados a uma chave específica (idViagem ou idAtividade), sem precisar percorrer todo o arquivo .db. Isso garante que todas as listagens feitas sejam executadas de forma rápida e ordenada.

Quando você lista as **atividades de uma viagem**, o código chama:

```
private Integer localizarRelId(int idViagem, int idAtividade) throws Exception {
    for (ParIntInt p : ixViagemRel.read(new ParIntInt(idViagem, -1))) {
        RelViagemAtividade r = arqRel.read(p.get2());
        if (r != null && r.getIdAtividade() == idAtividade)
            return p.get2();
    }
    return null;
}
```

- O índice B + retorna todos os IDs de vínculo que possuem o idViagem informado;
- Em seguida, as variáveis são usadas para ler o registro completo no arquivo principal;

```
public java.util.List<RelViagemAtividade> listarRelacoesDaViagem(int idViagem) throws Exception {
    java.util.ArrayList<RelViagemAtividade> out = new java.util.ArrayList<>();
    for (ParIntInt p : ixViagemRel.read(new ParIntInt(idViagem, -1))) {
        RelViagemAtividade r = arqRel.read(p.get2());
        if (r != null)
            out.add(r);
    }
    return out;
}

public java.util.List<RelViagemAtividade> listarRelacoesDaAtividade(int idAtividade) throws Exception {
    java.util.ArrayList<RelViagemAtividade> out = new java.util.ArrayList<>();
    for (ParIntInt p : ixAtividadeRel.read(new ParIntInt(idAtividade, -1))) {
        RelViagemAtividade r = arqRel.read(p.get2());
        if (r != null)
            out.add(r);
    }
    return out;
}
```

Utilizando desse método, não precisamos percorrer o arquivo inteiro, ou seja, a busca ocorre diretamente na estrutura de índice, que fica organizada em nós e folhas ordenadas da árvore, garantindo maior eficiência da estrutura.

5) Como o sistema trata a integridade referencial (remoção/atualização) entre as tabelas?

O sistema mantém a integridade referencial manualmente na DAO. Quando uma viagem ou atividade é removida ou alterada, o código atualiza ou elimina automaticamente todos os vínculos relacionados na tabela intermediária (RelViagemAtividade), garantindo que não fiquem registros órfãos no relacionamento N:N.

Na prática, isso vai funcionar da seguinte maneira:

Remoção em cascata (garantia de integridade):

- Ao excluir uma viagem, o método *removerTodosDaViagem(int idViagem)* é chamado: ele percorre o índice ixViagemRel e remove todos os vínculos relacionados a essa viagem, limpando tanto o arquivo principal quanto os índices B +;
- O mesmo ocorre ao excluir uma atividade, com o método *removerTodosDaAtividade(int idAtividade)*;
- Assim, nenhum vínculo permanece com IDs inexistentes, mantendo a integridade do relacionamento;

```java
// Cascatas
public void removerTodosDaViagem(int idViagem) throws Exception {
    java.util.ArrayList<Integer> relIds = new java.util.ArrayList<>();
    for (ParIntInt p : ixViagemRel.read(new ParIntInt(idViagem, -1))) {
        relIds.add(p.get2());
    }
    for (Integer rid : relIds) {
        RelViagemAtividade r = arqRel.read(rid);
        if (r != null) {
            ixAtividadeRel.delete(new ParIntInt(r.getIdAtividade(), rid));
            ixViagemRel.delete(new ParIntInt(idViagem, rid));
            arqRel.delete(rid);
        }
    }
}

public void removerTodosDaAtividade(int idAtividade) throws Exception {
    java.util.ArrayList<Integer> relIds = new java.util.ArrayList<>();
    for (ParIntInt p : ixAtividadeRel.read(new ParIntInt(idAtividade, -1))) {
        relIds.add(p.get2());
    }
    for (Integer rid : relIds) {
        RelViagemAtividade r = arqRel.read(rid);
        if (r != null) {
            ixViagemRel.delete(new ParIntInt(r.getIdViagem(), rid));
            ixAtividadeRel.delete(new ParIntInt(idAtividade, rid));
            arqRel.delete(rid);
        }
    }
}
```

Atualização consistente:

- Se os atributos de uma viagem ou atividade forem alterados (mas o id permanecer o mesmo), os vínculos continuam válidos;
- Portanto, não há inconsistência em atualizações, apenas nas exclusões (onde o sistema já faz a limpeza

automática);

Ambos os índices são atualizados:

- Ao remover ou adicionar vínculos, o sistema atualiza os dois índices B+ (ixViagemRel e ixAtividadeRel) para manter a coerência entre as tabelas;

6) Como foi organizada a persistência dos dados dessa nova tabela (mesmo padrão de cabeçalho e lápide)?

A persistência da nova tabela intermediária (RelViagemAtividade) segue o mesmo padrão de armazenamento utilizado nas demais entidades do sistema: possui cabeçalho com o último ID gerado e utiliza lápide (flag de exclusão lógica) para controlar registros ativos e removidos, garantindo compatibilidade com o framework genérico de CRUD.

```
// Serialização e desserialização
public byte[] toByteArray() throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    dos.writeInt(this.id);
    dos.writeInt(this.idViagem);
    dos.writeInt(this.idAtividade);
    byte[] bs = this.status.getBytes(charsetName: "UTF-8");
    dos.writeShort(bs.length);
    dos.write(bs);
    dos.writeInt(this.prioridade);
    return baos.toByteArray();
}
```

```
@Override
public String toString() {
    return "Rel[ id=" + id + ", viagem=" + idViagem + ", atividade=" + idAtividade +
           ", status=" + status + ", prioridade=" + prioridade + " ]";
}
```

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

- **1. Dependências e Verificações:** A tabela intermediária (RelViagemAtividadeDAO) mantém integridade referencial com as tabelas principais através de verificações;

```
public boolean vincular(int idViagem, int idAtividade, String status, int prioridade) throws Exception {
    // Verifica se viagem existe
    if (viagemDAO.buscarViagem(idViagem) == null)
        return false;
    // Verifica se atividade existe
    if (atividadeDAO.buscarAtividade(idAtividade) == null)
        return false;
    // ...
}
```

- **2. Operações Cascata:** Implementa remoção em cascata quando uma viagem ou atividade é excluída;

```
public void removerTodosDaViagem(int idViagem) throws Exception {
    java.util.ArrayList<Integer> relIds = new java.util.ArrayList<>();
    // Remove todos os vínculos da viagem
    for (ParIntInt p : ixViagemRel.read(new ParIntInt(idViagem, -1))) {
        relIds.add(p.get2());
    }
    // Remove registros e índices
    for (Integer rid : relIds) {
        RelViagemAtividade r = arqRel.read(rid);
        if (r != null) {
```

- **3. Consultas Relacionadas:** Oferece métodos para consultas bidirecionais;

```
public java.util.List<Atividade> listarAtividadesDaViagem(int idViagem)
public java.util.List<Viagem> listarViagensDaAtividade(int idAtividade)
```

- **4. Manutenção de Índices:** Mantém índices B+ atualizados para ambas as direções do relacionamento;

- **5. Operações CRUD Principais:**

  CREATE(C)
  Verifica existência nas tabelas principais;
  Evita duplicidade;
  Cria registro e atualiza índices;

  READ(R)
  Permite busca por viagem ou atividade;
  Retorna relacionamentos completos;

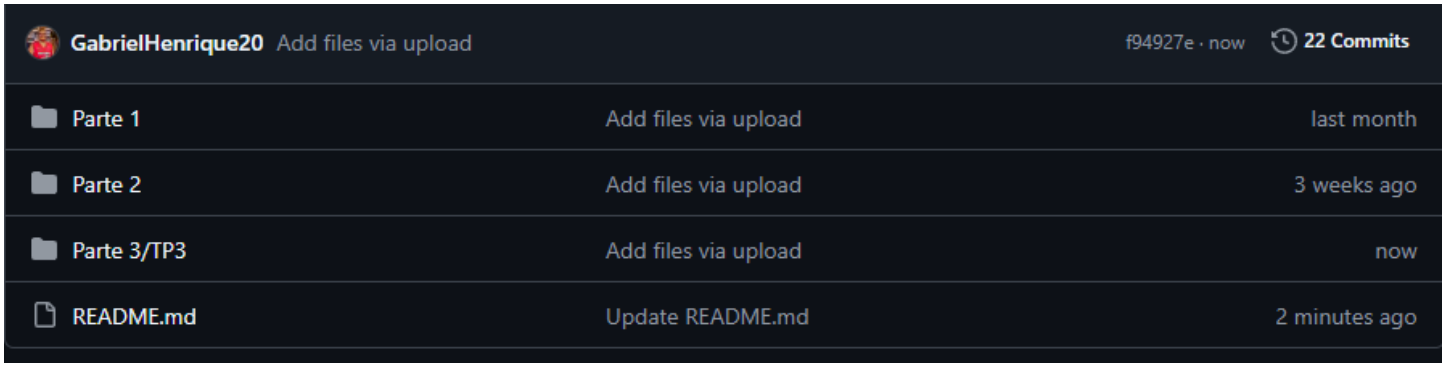
  UPDATE(U)
  Atualiza apenas atributos do relacionamento (status/prioridade);
  Mantém integridade dos vínculos;

  DELETE(D)
  Remove registro e atualizações nos índices;
  Suporte à remoção em cascata;

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

Para o projeto, foi criado um repositório central no perfil do Gabriel Henrique. A partir dele, foram estabelecidas branches distintas, permitindo que cada membro da equipe desenvolva, teste e modifique o código de forma isolada, sem impactar a estabilidade das ramificações principais.

No GitHub, o projeto do Sistema de Viagens está estruturado da seguinte maneira:



1ª pasta: Contém toda a parte 1 do nosso trabalho prático, ou seja, o primeiro CRUD de viagem feito além da documentação e seus respectivos diagramas.

2ª pasta: Contém toda a parte 2 do nosso trabalho prático, ou seja, todos os CRUDs já implementados e funcionando corretamente, além das duas relações 1:N presentes em nosso projeto.

3ª pasta: pasta com aprimoramento da parte 2, contendo os seguintes pacotes:

- **O pacote aeds3** contém as estruturas de dados implementadas pelo professor Kutova.
- **O pacote controller** contém os menus e lógica de navegação do sistema.
- **O pacote dao** realiza a persistência e controle dos arquivos de dados (com índices B+).
- **O pacote model** constitui-se das Entidades principais e tabela intermediária (RelViagemAtividade).
- **O pacore views** contém classe Principal.java (ponto de entrada da aplicação).
- **As 4 classes Buscar.** são responsáveis pela busca dos registros existentes.

4ª informação: README.md aprimorado, contendo informações adicionais sobre nosso projeto, principalmente sobre estruturação e execução do código apresentado:

📁 Estrutura do Projeto

1. O usuário inicia o programa (classe `Principal.java`).
2. É exibido um menu com opções de gerenciamento (Usuário, Viagem, Categoria, Atividade e Vínculos Viagem–Atividade (N:N)).
3. O usuário pode **adicionar**, **editar**, **buscar**, **remover** ou **listar** registros.
4. Os dados são persistidos em arquivos binários, utilizando cabeçalho e lápide para controle de integridade..

```
├── SistemaViagens/
│   ├── src/
│   │   ├── aeds3/          # Estruturas de dados (Árvore B+ e ParIntInt)
│   │   ├── controller/     # Menus e controle de navegação
│   │   ├── dao/            # Classes de persistência (DAO)
│   │   ├── model/          # Classes de modelo (Usuario, Viagem, Atividade, Categoria, RelViagemAtivid
│   │   ├── views/          # Classe Principal.java
│   │   └── Buscar*.java    # Classes auxiliares de busca
│   └── dados/              # Arquivos de dados gerados durante a execução
```

🔗 🚀 Como Executar

Ambiente Windows (recomendado)

1. **Instale o JDK** (Java Development Kit).
2. **Abra o projeto em uma IDE Java**, como VS Code (com a extensão Java) ou Eclipse.
3. **Compile o projeto** e execute a classe principal:

```
src/views/Principal.java
```

4. O menu principal será exibido no console. Basta seguir as opções para gerenciar os registros.

Ambiente Linux

1. Abra o terminal na pasta raiz do projeto **(onde estão as pastas dados/ e SistemaViagens/)**.
2. Navegue até o diretório de código-fonte:

```
cd SistemaViagens/src
```

3. Compile o código-fonte:

```
javac views/Principal.java
```