INSTITUTO FEDERAL DE MINAS GERAIS
CAMPUS SÃO JOÃO EVANGELISTA
SISTEMAS DE INFORMAÇÃO

CAMILA PARANHOS FERNANDES
FILIPE RUBSON DE ALMEIDA SANTOS
PATRÍCIA DA SILVA COSTA
VALDIR DE SOUZA CARVALHO NETO

BANCO DE DADOS II - SI 241

PROJETO FINAL
PARTE 2 - ARTEFATO PDF

São João Evangelista
2025

Observações:

- ➔ **AS REGRAS DE NEGÓCIO DEVEM SER TESTADAS NA ORDEM DO PDF PARA OS RESULTADOS COINCIDIREM;**
- ➔ **OS ARQUIVOS *DML* ENTREGUES NESTA PARTE (2) DEVEM SER EXECUTADOS AO INVÉS DOS ENTREGUES NA PARTE 1! CADA UM NO SEU RESPECTIVO SGBD.**

1. VISÕES (Views)

1.1. View 1: Relatório de Saúde Financeira por Curso

1.1.1. Enunciado ECA

Evento (Entrada): Solicitação de relatório gerencial financeiro.

Condição (Lógica): Agrupar os dados financeiros por nome_curso. Deve-se diferenciar os valores já recebidos dos valores a receber.

Ação (Saída): Apresentar uma tabela contendo: Nome do Curso; Quantidade de Alunos Ativos; Total Recebido (Soma condicional onde status = 'Pago'); Total a Receber/Inadimplência (Soma condicional onde status = 'Pendente' ou 'Vencido').

1.1.2. Decisões de Implementação (Documentação)

Inclusão Total (LEFT JOIN na Cadeia): A View utiliza LEFT JOIN iniciando na tabela cursos e se estendendo para turmas e matrículas. Esta decisão é gerencialmente crítica: garante que **todos os cursos** cadastrados (mesmo aqueles sem alunos ativos ou sem turmas) apareçam no relatório, retornando 0.00 para os totais e 0 para alunos.

Filtro de Alunos Ativos no JOIN: A condição 'm.status_mat = 'Ativa'' é aplicada diretamente no LEFT JOIN com matriculas. Isso permite que a View conte apenas os alunos que estão de fato gerando receita atual, mas sem descartar cursos que possam estar inativos (pois o curso é o lado esquerdo do JOIN).

Cálculo Robusto com COALESCE: O uso de COALESCE(SUM(...), 0.00) é essencial em conjunto com o LEFT JOIN. Ele converte os resultados NULL (que ocorrem quando um curso não tem parcelas ou alunos ativos) para o valor numérico 0.00, prevenindo erros de cálculo e garantindo relatórios financeiros precisos.

Classificação Financeira (CASE WHEN): Mantém-se o uso de CASE WHEN para categorizar e somar valores de forma separada ('total_recebido' para status 'Pago' e 'total_a_receber' para status 'Pendente' e 'Vencido'), fornecendo uma visão clara do fluxo de caixa por curso.

1.1.3. Transcrição do Código SQL

1.1.3.1. MySQL e PostgreSQL

```
CREATE OR REPLACE VIEW vw_saude_financeira AS
SELECT c.nome_curso AS nome_curso, COUNT(DISTINCT m.aluno_id) AS qtd_alunos_ativos,
COALESCE(SUM(CASE WHEN men.status_pag = 'Pago' THEN men.valor_nominal ELSE 0 END), 0.00) AS total_recebido,
COALESCE(SUM(CASE WHEN men.status_pag IN ('Pendente', 'Vencido') THEN men.valor_nominal ELSE 0 END), 0.00) AS total_a_receber
FROM cursos c LEFT JOIN turmas t ON t.curso_id = c.curso_id
LEFT JOIN matriculas m ON m.turma_id = t.turma_id AND m.status_mat = 'Ativa'
LEFT JOIN mensalidades men ON men.matricula_id = m.matricula_id
GROUP BY c.nome_curso ORDER BY c.nome_curso;
```

1.1.4. Casos de Teste (Descrição)

- TESTE 1: Verifica se o relatório agrupa corretamente pelo nome do curso e se os totais estão corretos. Vamos verificar o curso 'Inglês Básico I' (ID 1). Total de alunos ativos: 8; Total Recebido: 7200.00; Total a Receber: 10500.00;

```
call tr_transferir_turma(1, 18);
SELECT nome_curso, qtd_alunos_ativos, total_recebido, total_a_receber FROM vw_saude_financeira WHERE nome_curso = 'Inglês Básico I';
```

- TESTE 2: Verifica se o curso 'Espanhol Básico' (ID 6) está calculando corretamente. Total de alunos ativos: 5; Total Recebido: 2100.00; Total a Receber: 8700.00;

```
SELECT nome_curso, qtd_alunos_ativos, total_recebido, total_a_receber
FROM vw_saude_financeira WHERE nome_curso = 'Espanhol Básico I';
```

1.2. View 2: Mapa de Ocupação de Salas

1.2.1. Enunciado ECA

Evento (Entrada): Consulta operacional para alocação de novas turmas.

Condição (Lógica): Calcular a ocupação atual (COUNT de matrículas ativas por sala) e compará-la com a capacidade máxima da sala.

Ação (Saída): Apresentar: Nome da Sala e Capacidade; Ocupação Atual; Coluna calculada "Status Lotação": Retornar 'LOTADA' se (Ocupação >= Capacidade), caso contrário retornar 'DISPONÍVEL'.

1.2.2. Decisões de Implementação (Documentação)

Uso de LEFT JOIN (Inclusão Total): A View utiliza LEFT JOIN começando pela tabela salas (s) e, em seguida, turmas (t) e matrículas (m). Essa decisão é fundamental para garantir que TODAS as salas sejam listadas no relatório (mesmo que estejam vazias ou sem turmas agendadas), garantindo um mapa de ocupação completo.

Filtro de Matrículas Ativas: A contagem de alunos para a ocupação é restrita a 'm.status_mat = 'Ativa''. Isso assegura que a métrica 'ocupacao_atual' reflita apenas alunos que estão de fato ocupando a vaga, alinhando-se com a lógica da Trigger de Lotação.

Classificação de Status (CASE WHEN): Utiliza-se a função CASE WHEN para criar a coluna 'status_lotacao'.

A regra 'COUNT(m.matricula_id) >= s.capacidade' classifica a sala como 'LOTADA', fornecendo uma indicação visual e rápida para o planejamento de novas turmas.

1.2.3. Transcrição do Código SQL

1.2.3.1. MySQL e PostgreSQL

```
CREATE OR REPLACE VIEW vw_ocupacao_salas AS
SELECT s.sala_id, s.nome_sala, s.capacidade, COUNT(m.matricula_id) AS ocupacao_atual,
CASE WHEN COUNT(m.matricula_id) >= s.capacidade THEN 'LOTADA'
ELSE 'DISPONÍVEL' END AS status_lotacao
FROM salas s LEFT JOIN turmas t ON t.sala_id = s.sala_id
LEFT JOIN matriculas m ON m.turma_id = t.turma_id
AND m.status_mat = 'Ativa'
GROUP BY s.sala_id, s.nome_sala, s.capacidade;
```

1.2.4. Casos de Teste (Descrição)

- TESTE 1: Validação de Sala Lotada. Verifica a Sala ID 3 (Lab. A), que tem Capacidade 18. Após a preparação de DML (que adicionou 17 alunos na Turma 19, que já tinha 1 aluno), a ocupação deve ser 18. O retorno esperado é ocupacao_atual = 18 e status_lotacao = 'LOTADA'.

```
SELECT sala_id, nome_sala, capacidade, ocupacao_atual, status_lotacao FROM vw_ocupacao_salas WHERE sala_id = 3;
```

- TESTE 2: Validação de Sala Vazia (LEFT JOIN). Verifica se uma sala que não possui nenhuma turma (ex: Sala ID 5 - Sala de Reunião, que tem Capacidade 10) é listada corretamente com ocupação zero e status 'DISPONÍVEL'. O retorno esperado é ocupacao_atual = 0 e status_lotacao = 'DISPONÍVEL'.

```
SELECT sala_id, nome_sala, capacidade, ocupacao_atual, status_lotacao FROM vw_ocupacao_salas WHERE sala_id = 5;
```

2. FUNCTIONS (Funções)

2.1. Função 1: Calcular Dívida Ativa do Aluno

2.1.1. Enunciado ECA

Evento (Entrada): O sistema ou um usuário solicita a verificação financeira de um aluno específico (passando o id_aluno).

Condição (Lógica): O sistema deve identificar na tabela mensalidades todos os registros vinculados a este aluno que satisfaçam uma das seguintes condições: O status de pagamento é explicitamente 'Vencido'; OU O status de pagamento é 'Pendente', mas a data_vencimento é anterior à data atual do sistema (indicando atraso não processado).

Ação (Saída): Retornar a soma (SUM) dos valores nominais dessas parcelas filtradas. Caso nenhuma parcela se enquadre nas condições (aluno adimplente), a função deve retornar, obrigatoriamente, o valor 0.00 em vez de NULL.

2.1.2. Decisões de Implementação (Documentação)

Tratamento de Nulos (COALESCE): A decisão mais crítica desta função foi o uso de COALESCE(SUM(...), 0). Em SQL, a soma de um conjunto vazio resulta em NULL.

Para regras de negócio financeiras, retornar NULL pode causar erros de cálculo em aplicações (ex: NULL + 100 = NULL). O COALESCE garante que um aluno sem dívidas retorne 0.00.

Lógica de Vencimento Dupla: A cláusula WHERE implementa uma verificação de segurança. Não confiamos apenas no status 'Vencido' (que pode depender de uma rotina noturna de atualização). Verificamos também se é 'Pendente' com data passada (data_vencimento < CURRENT_DATE), garantindo que a dívida seja calculada em tempo real, mesmo se o status no banco estiver desatualizado.

Compatibilidade de Datas: No MySQL utilizamos CURDATE() e no PostgreSQL CURRENT_DATE, respeitando as funções nativas de cada SGBD para capturar a data do servidor.

2.1.3. Transcrição do Código SQL

2.1.3.1. MySQL

```
drop function if exists fc_calcular_divida_ativa;
delimiter //
create function fc_calcular_divida_ativa(param_id_aluno int)
returns decimal(10,2)
deterministic
begin
```

```
declare total_divida decimal(10,2);
select coalesce(sum(men.valor_nominal), 0) into total_divida
from mensalidades men join matriculas ma on ma.matricula_id = men.matricula_id
where param_id_aluno = ma.aluno_id and(
men.status_pag = 'Vencido'or(men.status_pag = 'Pendente' and men.data_vencimento <
curdate()));
return total_divida;
end //
delimiter ;
```

2.1.3.2. PostgreSQL

```
DROP FUNCTION IF EXISTS fc_calcular_divida_ativa(int);
CREATE OR REPLACE FUNCTION fc_calcular_divida_ativa(param_id_aluno int)
RETURNS decimal(10,2) AS $$
DECLARE total_divida decimal(10,2);
BEGIN
SELECT COALESCE(SUM(men.valor_nominal), 0) INTO total_divida
FROM mensalidades men JOIN matriculas ma ON ma.matricula_id = men.matricula_id
WHERE param_id_aluno = ma.aluno_id AND ( men.status_pag = 'Vencido'
OR (men.status_pag = 'Pendente' AND men.data_vencimento < CURRENT_DATE));
RETURN total_divida;
END;
$$ LANGUAGE plpgsql;
```

2.1.4. Casos de Teste (Descrição)

- Este teste valida o cenário positivo. O aluno ID 4 (Daniel) possui parcelas explicitamente vencidas no script DML. O retorno esperado é a soma exata dessas parcelas, confirmando que a lógica de soma está correta.

```
select pessoa_id, p.nome, fc_calcular_divida_ativa(pessoa_id) as divida_total from pessoas p where pessoa_id = 4;
```

- Este caso testa a robustez da função via COALESCE. O aluno ID 20 tem todas as mensalidades pagas. O sistema não deve retornar NULL nem erro, mas sim o valor numérico 0.00, provando que o tratamento de exceção para conjuntos vazios funciona.

```
select p.nome, fc_calcular_divida_ativa(p.pessoa_id) as divida_total from pessoas p where p.pessoa_id = 20;
```

2.2. Função 2: Média de Desempenho por Idioma

2.2.1. Enunciado ECA

Evento (Entrada): Necessidade de emitir um boletim ou relatório de desempenho de um aluno em um idioma específico (passando id_aluno e id_idioma).

Condição (Lógica): O sistema deve rastrear as avaliações do aluno percorrendo o caminho: Avaliações → Matrículas → Turmas → Cursos → Idiomas. Devem ser consideradas apenas as notas das avaliações que pertençam a cursos do idioma solicitado.

Ação (Saída): Calcular e retornar a média aritmética (AVG) dessas notas, arredondada para duas casas decimais. Se o aluno nunca cursou aquele idioma (sem avaliações), retornar 0.00.

2.2.2. Decisões de Implementação (Documentação)

Navegação entre Tabelas (Joins): A complexidade desta função reside na distância entre a tabela avaliacoes (onde está a nota) e a tabela idiomas (o filtro desejado).

Optou-se por uma cadeia de INNER JOINs (Avaliações → Matrículas → Turmas → Cursos) para garantir integridade. Isso assegura que a nota pertence, de fato, àquele aluno naquele contexto específico.

Abstração de Cursos: A função permite calcular a média de um aluno em "Inglês" (Idioma), independentemente de ele ter feito "Inglês Básico 1" ou "Inglês Avançado". O agrupamento é feito pelo idioma pai, oferecendo uma visão macro do desempenho do estudante.

Determinismo: As funções foram marcadas como DETERMINISTIC (no MySQL), informando ao otimizador do banco que, para os mesmos dados de entrada, o resultado será sempre o mesmo, o que pode otimizar a performance em consultas repetitivas.

2.2.3. Transcrição do Código SQL

2.2.3.1. MySQL

```
drop function if exists fc_media_idioma;
delimiter //
create function fc_media_idioma(param_id_aluno int, param_id_idioma int)
returns decimal(4,2)
deterministic
begin
declare media decimal(4,2);
select coalesce(avg(av.nota), 0) into media
from avaliacoes av join matriculas ma on ma.matricula_id = av.matricula_id
join turmas tu on tu.turma_id = ma.turma_id join cursos cu on cu.curso_id = tu.curso_id
where ma.aluno_id = param_id_aluno and cu.idioma_id = param_id_idioma;
```

```
return media;
end //
delimiter ;
```

2.2.3.2. PostgreSQL

```
drop function if exists fc_media_idioma(int, int);
create or replace function fc_media_idioma(param_id_aluno int, param_id_idioma int)
returns decimal(4,2) as $$
declare media decimal(4,2);
begin
select coalesce(avg(av.nota), 0) into media from avaliacoes av
join matriculas ma on ma.matricula_id = av.matricula_id join turmas tu on tu.turma_id = ma.turma_id
join cursos cu on cu.curso_id = tu.curso_id where ma.aluno_id = param_id_aluno and cu.idioma_id = param_id_idioma;
return media;
end;
$$ language plpgsql;
```

2.2.4. Casos de Teste (Descrição)

- Verifica se os JOINS estão recuperando corretamente as notas. O aluno ID 4 cursou turmas de Inglês. A função deve percorrer todo o caminho relacional e retornar a média aritmética das notas lançadas.

```
call tr_realizar_pagamento(1, 5, 300.00);
-- Verificação (Deve mostrar status 'Pago' e data_pag preenchida)
select matricula_id, numero_parcela, status_pag, data_pag, valor_pag from mensalidades where matricula_id = 1 and numero_parcela = 5;
```

- Este caso testa o bloqueio por registro inexistente. Tentamos realizar a baixa da Parcela 99, que não existe na Matrícula ID 1. O procedimento deve retornar 0 linhas afetadas, executar o ROLLBACK e retornar o erro 'Pagamento Falhou: Parcela não encontrada ou erro na baixa.'.O retorno esperado é a mensagem de erro.

```
call tr_realizar_pagamento(1, 99, 300.00);
-- Verificação (Deve retornar 0 linhas)
select matricula_id, numero_parcela, status_pag, data_pag, valor_pag from mensalidades where matricula_id = 1 and numero_parcela = 99;
```

3. TRANSACTIONS (Transações)

3.1. Transaction 1: Transferência de Turma Segura

3.1.1. Enunciado ECA

Evento (Entrada): Solicitação de transferência de um aluno de sua turma atual para uma nova turma de destino (id_matricula, id_nova_turma).

Condição (Lógica): Verifica-se a disponibilidade na turma de destino. A regra é: (Capacidade da Sala - Total de Alunos Ativos na Nova Turma) > 0.

Ação (Saída): Se a condição for verdadeira: Executar o UPDATE do turma_id na tabela matriculas e confirmar a transação (COMMIT). Se a condição for falsa: Desfazer qualquer alteração (ROLLBACK) e levantar uma exceção informando que não há vagas disponíveis.

3.1.2. Decisões de Implementação (Documentação)

Atomicidade: O conceito transacional é mantido. No MySQL, usamos 'START TRANSACTION', 'COMMIT' e 'ROLLBACK' explícitos. No PostgreSQL, ao usar 'CREATE OR REPLACE PROCEDURE' (PL/pgSQL), o bloco BEGIN/END funciona como uma transação implícita; o 'COMMIT' ocorre automaticamente no final do bloco, a menos que um erro ('RAISE EXCEPTION') seja lançado, o que força um ROLLBACK automático.

Controle de Erro: A sinalização de erro foi adaptada. No MySQL, utiliza-se SIGNAL SQLSTATE '45000'. No PostgreSQL, o comando é RAISE EXCEPTION, que é o mecanismo nativo para abortar a execução e desfazer a transação.

Validação de Vaga: A lógica de verificação (contagem de alunos ativos na nova turma vs. capacidade) é mantida, garantindo que a integridade do negócio seja a prioridade.

3.1.3. Transcrição do Código SQL

3.1.3.1. MySQL

```
drop procedure if exists tr_transferir_turma;
delimiter //
create procedure tr_transferir_turma(param_matricula_id INT, param_nova_turma_id INT)
begin
declare capacidade_sala INT;
declare alunos_ativos_na_nova_turma INT;
declare turma_atual_id INT;

start transaction;
```

```
select turma_id into turma_atual_id from matriculas WHERE matricula_id = param_matricula_id;

if turma_atual_id is null then rollback;
signal sqlstate '45000' set message_text = 'Erro: Matrícula não encontrada.';
END IF;

select S.capacidade into capacidade_sala from turmas T join salas S on T.sala_id = S.sala_id where T.turma_id = param_nova_turma_id;

select count(M.matricula_id) into alunos_ativos_na_nova_turma from matriculas M where M.turma_id = param_nova_turma_id and M.status_mat = 'Ativa';

if (alunos_ativos_na_nova_turma + 1) > capacidade_sala then rollback;
signal sqlstate '45000' set message_text = 'Transferência Falhou: Sala de destino lotada.';
else update matriculas set turma_id = param_nova_turma_id where matricula_id = param_matricula_id;
commit;
end if;

end //
delimiter ;
```

3.1.3.2. PostgreSQL

```
CREATE OR REPLACE PROCEDURE escola_idiomas.tr_transferir_turma(
   param_matricula_id INTEGER,
   param_nova_turma_id INTEGER
)
LANGUAGE plpgsql
AS $$
DECLARE
   capacidade_sala INTEGER;
   alunos_ativos_na_nova_turma INTEGER;
   turma_atual_id INTEGER;
BEGIN
SELECT turma_id INTO turma_atual_id FROM escola_idiomas.matriculas WHERE matricula_id = param_matricula_id;

IF turma_atual_id IS NULL THEN RAISE EXCEPTION 'Erro: Matrícula não encontrada.';
```

```
END IF;

SELECT S.capacidade INTO capacidade_sala FROM escola_idiomas.turmas T JOIN escola_idiomas.salas S ON T.sala_id = S.sala_id WHERE T.turma_id = param_nova_turma_id;

SELECT COUNT(M.matricula_id) INTO alunos_ativos_na_nova_turma FROM escola_idiomas.matriculas M WHERE M.turma_id = param_nova_turma_id AND M.status_mat = 'Ativa';

        IF (alunos_ativos_na_nova_turma + 1) > capacidade_sala THEN RAISE EXCEPTION 'Transferência Falhou: Sala de destino lotada.';
   ELSE
            UPDATE escola_idiomas.matriculas SET turma_id = param_nova_turma_id WHERE matricula_id = param_matricula_id;

   END IF;
END;
$$;
```

3.1.4. Casos de Teste (Descrição)

- Este teste valida o cenário positivo (Transfêrencia Permitida). Aluno: Daniel Silva (Matrícula ID 1). Turma Atual: ING-BAS-T01-2025 (ID 1). Turma Destino: ESP-BAS-T04-2025 (ID 18). Ocupação T18: 1 aluno (Matrícula 38). Capacidade: 25. HÁ VAGA. O retorno esperado é a Matrícula 1 ter o campo turma_id = 18.

```
call tr_transferir_turma(1, 18);
-- Verificação (Deve mostrar turma_id = 18)
select M.matricula_id, T.turma_id, P.nome, T.nome_turma from matriculas M join pessoas P on M.aluno_id = P.pessoa_id join turmas T on M.turma_id = T.turma_id where M.matricula_id = 1;
```

- Este caso testa o bloqueio por lotação (Transação desfeita). Usamos a Turma ID 19 (ITA-RAP-T02-2025) que está lotada. A tentativa de mover a Matrícula ID 9 (Lucas) para lá DEVE FALHAR. O retorno esperado é o erro 'Transferência Falhou: Sala de destino lotada.' e a Matrícula ID 9 DEVE PERMANECER com sua turma original (ID 14) após o ROLLBACK.

```
call tr_transferir_turma(9, 19);

-- Verificação (Deve mostrar a turma original: turma_id = 14)
select M.matricula_id, T.turma_id, P.nome, T.nome_turma from matriculas M join pessoas P on M.aluno_id = P.pessoa_id join turmas T on M.turma_id = T.turma_id where M.matricula_id = 9;
```

3.2. Transaction 2: Pagamento Completo/Baixa

3.2.1. Enunciado ECA

Evento (Entrada): Registro de pagamento de uma mensalidade no caixa (id_mensalidade, valor_recebido, id_forma_pagamento).

Condição (Lógica): A existência da mensalidade e a consistência dos dados.

Ação (Saída): Atualizar atomicamente (tudo ou nada) três campos na tabela mensalidades: Alterar status_pag para 'Pago'. Registrar a data_pag com a data atual do sistema. Registrar o valor_pag com o montante efetivamente recebido. Se qualquer um desses updates falhar, toda a operação é desfeita para evitar inconsistência (ex: constar como pago mas sem data).

3.2.2. Decisões de Implementação (Documentação)

Atomicidade e Consistência: O comando UPDATE altera os três campos críticos simultaneamente (status_pag, data_pag, valor_pag). Em caso de falha de qualquer natureza (ex: registro não encontrado, falha de integridade), o sistema executa o ROLLBACK, impedindo que a mensalidade fique com status 'Pago' sem o registro da data ou valor, mantendo a consistência (ACID) em ambos os SGBDs.

Controle de Linhas Afetadas: Para validar a existência da parcela, utilizamos a função de contagem de linhas afetadas (ROW_COUNT() no MySQL/MariaDB e GET DIAGNOSTICS... ROW_COUNT no PostgreSQL). Isso garante que o COMMIT só ocorra se exatamente UMA parcela for atualizada, prevenindo confirmação para registros inexistentes.

3.2.3. Transcrição do Código SQL

3.2.3.1. MySQL

```
drop procedure if exists tr_realizar_pagamento;
delimiter //
create procedure tr_realizar_pagamento(param_matricula_id INT, param_numero_parcela INT, param_valor_pago DECIMAL(10,2))
begin
declare linhas_afetadas int default 0;

start transaction;

update mensalidades set status_pag = 'Pago', data_pag = current_date(), valor_pag = param_valor_pago
```

```
where matricula_id = param_matricula_id and numero_parcela = param_numero_parcela;

set linhas_afetadas = row_count();

if linhas_afetadas = 1 then commit;
else rollback;
signal sqlstate '45000' set message_text = 'Pagamento Falhou: Parcela não encontrada ou erro na baixa.';
   end if;

end //
delimiter ;
```

3.2.3.2. PostgreSQL

```
CREATE OR REPLACE PROCEDURE escola_idiomas.tr_realizar_pagamento(param_matricula_id INTEGER, param_numero_parcela INTEGER, param_valor_pago DECIMAL)
LANGUAGE plpgsql
AS $$
DECLARE
   linhas_afetadas INTEGER;
BEGIN

   UPDATE escola_idiomas.mensalidades
   SET
      status_pag = 'Pago',
      data_pag = CURRENT_DATE,
      valor_pag = param_valor_pago
   WHERE
                  matricula_id = param_matricula_id AND numero_parcela = param_numero_parcela;

   GET DIAGNOSTICS linhas_afetadas = ROW_COUNT;

   IF linhas_afetadas = 1 THEN NULL;
   ELSE
         RAISE EXCEPTION 'Pagamento Falhou: Parcela não encontrada ou erro na baixa.';
   END IF;

END;
$$;
```

3.2.4. Casos de Teste (Descrição)

- Este teste valida o cenário positivo (Baixa Bem-sucedida). A mensalidade Matrícula ID 1, Parcela 5, está com status 'Pendente'. O procedimento deve realizar o UPDATE dos três campos cruciais (status, data, valor). O retorno esperado é status_pag = 'Pago', data_pag preenchida (data atual) e valor_pag = 300.00.

```
call tr_realizar_pagamento(1, 5, 300.00);
-- Verificação (Deve mostrar status 'Pago' e data_pag preenchida)
select matricula_id, numero_parcela, status_pag, data_pag, valor_pag from mensalidades where matricula_id = 1 and numero_parcela = 5;
```

- Este caso testa o bloqueio por registro inexistente. Tentamos realizar a baixa da Parcela 99, que não existe na Matrícula ID 1. O procedimento deve retornar 0 linhas afetadas, executar o ROLLBACK e retornar o erro 'Pagamento Falhou: Parcela não encontrada ou erro na baixa.'. O retorno esperado é a mensagem de erro.

```
call tr_realizar_pagamento(1, 99, 300.00);
-- Verificação (Deve retornar 0 linhas)
select matricula_id, numero_parcela, status_pag, data_pag, valor_pag from mensalidades where matricula_id = 1 and numero_parcela = 99;
```

4. PROCEDURES (Procedimentos)

4.1. Procedure 1: Geração Automática de Mensalidades

4.1.1. Enunciado ECA

Evento (Entrada): Ocorre a confirmação de uma nova matrícula no sistema (chamada da procedure passando id_matricula).

Condição (Lógica): O curso associado à matrícula deve possuir uma duração em meses definida (duracao_meses > 0). O sistema deve iterar (loop) de 1 até N (duração do curso).

Ação (Saída): Inserir registros na tabela mensalidades correspondentes a cada parcela. A data de vencimento de cada parcela deve ser calculada para o dia 10 dos meses subsequentes à data da matrícula, com status inicial definido como 'Pendente'.

4.1.2. Decisões de Implementação (Documentação)

Tratamento de Validação (NULL/Zero): A cláusula IF inicial verifica se a duração do curso (v_duracao_meses) é NULL ou menor ou igual a zero. Essa validação garante que a procedure não tente realizar uma divisão por zero (v_valor_total / v_duracao_meses) e evita a geração de parcelas inválidas, abortando a operação com um erro (SIGNAL SQLSTATE).

Cálculo de Data de Vencimento: A data de vencimento é calculada dinamicamente utilizando o comando DATE_ADD(..., INTERVAL v_contador MONTH) em conjunto com a data de matrícula (v_data_matricula). Para forçar o vencimento sempre para o dia 10, usa-se a função CONCAT(YEAR(...), '-', MONTH(...), '-10'), garantindo consistência na política de cobrança mensal.

Eficiência de Loop (WHILE): Utiliza-se a estrutura de repetição WHILE para iterar de 1 até a duração total do curso (v_duracao_meses). O comando INSERT é executado dentro do loop, criando todas as parcelas de forma sequencial e automática em uma única execução da procedure.

Status Inicial Padrão: Todas as parcelas são inseridas com status_pag='Pendente'. Isso estabelece o estado financeiro inicial correto e diferencia as parcelas que ainda serão devidas.

4.1.3. Transcrição do Código SQL

4.1.3.1. MySQL

```
DELIMITER //
CREATE PROCEDURE sp_gerar_mensalidades(IN p_matricula_id INT)
BEGIN
   DECLARE v_duracao_meses INT;
   DECLARE v_valor_total DECIMAL(10, 2);
```

```
   DECLARE v_valor_parcela DECIMAL(10, 2);
   DECLARE v_data_matricula DATE;
   DECLARE v_contador INT DEFAULT 1;
   DECLARE v_data_vencimento DATE;
   DECLARE v_curso_id INT;

SELECT m.data_matricula, m.valor_total_curso, c.duracao_meses, m.turma_id
INTO v_data_matricula, v_valor_total, v_duracao_meses, v_curso_id
FROM matriculas m JOIN turmas t ON m.turma_id = t.turma_id
JOIN cursos c ON t.curso_id = c.curso_id WHERE m.matricula_id = p_matricula_id;

IF v_duracao_meses IS NULL OR v_duracao_meses <= 0 THEN SIGNAL SQLSTATE
'45000' SET MESSAGE_TEXT = 'Curso não possui duração válida para geração de
mensalidades.';
ELSE SET v_valor_parcela = v_valor_total / v_duracao_meses;
WHILE v_contador <= v_duracao_meses DO
SET v_data_vencimento = DATE_ADD(v_data_matricula, INTERVAL v_contador
MONTH);
SET v_data_vencimento = CONCAT(YEAR(v_data_vencimento), '-',
MONTH(v_data_vencimento), '-10');

INSERT INTO mensalidades (matricula_id, numero_parcela, data_vencimento,
valor_nominal, status_pag)
VALUES (p_matricula_id, v_contador, v_data_vencimento,
v_valor_parcela,'Pendente');

SET v_contador = v_contador + 1;
END WHILE;
END IF;
END //
DELIMITER ;
```

4.1.3.2. PostgreSQL

```
CREATE OR REPLACE PROCEDURE escola_idiomas.sp_gerar_mensalidades(
   p_matricula_id INT
)
LANGUAGE plpgsql
AS $$
DECLARE
   v_duracao_meses INT;
   v_valor_total DECIMAL(10, 2);
   v_data_matricula DATE;
```

```
    v_valor_parcela DECIMAL(10, 2);
    v_contador INT := 1;
    v_data_vencimento DATE;
BEGIN
    -- 1. Obter dados da Matrícula e do Curso
SELECT m.data_matricula, m.valor_total_curso, c.duracao_meses
INTO v_data_matricula, v_valor_total, v_duracao_meses
FROM escola_idiomas.matriculas m JOIN escola_idiomas.turmas t ON m.turma_id =
t.turma_id
JOIN escola_idiomas.cursos c ON t.curso_id = c.curso_id
WHERE m.matricula_id = p_matricula_id;

    -- 2. Condição de Validação (Duração do Curso)
IF v_duracao_meses IS NULL OR v_duracao_meses <= 0 THEN RAISE EXCEPTION
'Curso não possui duração válida para geração de mensalidades.';
ELSE v_valor_parcela := v_valor_total / v_duracao_meses;
WHILE v_contador <= v_duracao_meses LOOP
v_data_vencimento := (v_data_matricula + (v_contador || ' month')::interval);
v_data_vencimento := DATE_TRUNC('month', v_data_vencimento) + INTERVAL '9
days'; -- Define o dia 10

INSERT INTO escola_idiomas.mensalidades (matricula_id, numero_parcela,
data_vencimento, valor_nominal, status_pag)
VALUES (p_matricula_id, v_contador, v_data_vencimento, v_valor_parcela,
'Pendente');
v_contador := v_contador + 1;

END LOOP;
END IF;
END;
$$;
```

4.1.4. Casos de Teste (Descrição)

- TESTE DE SUCESSO (CENÁRIO POSITIVO) A matrícula ID 50 (fictícia) no curso 'Italiano Rápido' (3 meses, valor 900.00) será processada. A procedure deve inserir 3 parcelas de 300.00 com status 'Pendente'. O vencimento deve ser o dia 10 dos meses subsequentes (10/12/2025, 10/01/2026, 10/02/2026).

```
-- Preparação (garantindo que Aluno e Matrícula existam)
SET search_path TO escola_idiomas; -- EXECUTAR APENAS NO POSTGRESQL
INSERT INTO pessoas (pessoa_id, nome, data_nasc, cpf, email) VALUES
(120, 'Aluno Teste Gerar', '2000-01-01', '12012012000', 'teste@gerar.com');
INSERT INTO alunos (aluno_id, nivel_proeficiencia_inicial) VALUES (120, 'Básico');
```

```
INSERT INTO matriculas (matricula_id, aluno_id, turma_id, data_matricula, valor_total_curso, status_mat) VALUES
(50, 120, 11, '2025-11-20', 900.00, 'Ativa');
CALL sp_gerar_mensalidades(50);
-- Verificação (Deve retornar 3 linhas, todas com status 'Pendente')
SELECT numero_parcela, data_vencimento, valor_nominal, status_pag FROM mensalidades WHERE matricula_id = 50;
```

- TESTE DE FALHA (CENÁRIO NEGATIVO - Matrícula Inexistente). O teste chama a procedure com uma Matrícula ID 999 que não existe no banco de dados. O SELECT INTO dentro da procedure não encontrará registros e retornará NULL para a variável v_duracao_meses. O procedimento deve ser bloqueado pela cláusula IF (v_duracao_meses IS NULL) e retornar o erro 'Curso não possui duração válida para geração de mensalidades.' O retorno esperado é o erro.

```
-- Preparação (garantindo que Aluno exista para não falhar na FK matriculas_alunos)
SET search_path TO escola_idiomas;  -- EXECUTAR APENAS NO POSTGRESQL
INSERT INTO pessoas (pessoa_id, nome, data_nasc, cpf, email) VALUES
(121, 'Aluno Teste Falha', '2000-01-01', '12112112100', 'teste@falha.com');
INSERT INTO alunos (aluno_id, nivel_proeficiencia_inicial) VALUES (121, 'Básico');
 -- ESTE COMANDO DEVE GERAR O ERRO: 'Curso não possui duração válida para geração de mensalidades.'
CALL sp_gerar_mensalidades(999);
-- Verificação (Deve retornar 0 linhas)
SELECT * FROM mensalidades WHERE matricula_id = 999;
```

4.2. Procedure 2: Aplicação de Bolsa/Desconto

4.2.1. Enunciado ECA

Evento (Entrada): A secretaria decide conceder uma bolsa ou desconto a um aluno já matriculado (chamada da procedure passando id_matricula e percentual_desconto).

Condição (Lógica): O desconto deve ser aplicado apenas sobre as mensalidades cujo status_pag seja 'Pendente'. Parcelas 'Pagas' ou 'Vencidas' devem ser ignoradas para manter a integridade do histórico financeiro e evitar recálculos indevidos de dívidas passadas.

Ação (Saída): Atualizar (UPDATE) o campo valor_nominal das parcelas elegíveis, reduzindo o valor original conforme o percentual informado. Caso o percentual seja inválido (<=0 ou >100), a operação deve ser abortada.

4.2.2. Decisões de Implementação (Documentação)

Validação de Percentual: A cláusula IF inicial impõe regras de negócio rigorosas para o percentual de desconto (p_percentual_desconto). Ela impede que descontos sejam nulos, negativos ou maiores que 100%. A falha nessa condição dispara um SIGNAL SQLSTATE, abortando a operação antes de qualquer alteração nos dados financeiros.

Aplicação Seletiva (status_pag = 'Pendente'): A atualização do valor nominal (UPDATE mensalidades) é filtrada de forma restritiva pela condição status_pag = 'Pendente'. Esta decisão é crucial para a integridade contábil, pois impede a alteração de parcelas que já foram pagas ('Pago') ou que já foram identificadas como dívida consolidada ('Vencido').

Cálculo com Fator Multiplicador: Em vez de subtrair o valor do desconto (valor_nominal - (valor_nominal * p_percentual_desconto / 100)), o cálculo utiliza o fator multiplicador (1 - p_percentual_desconto / 100). Isso simplifica a expressão do UPDATE (valor_nominal * v_fator_multiplicador), tornando o código mais limpo e menos propenso a erros de precedência matemática.

Feedback de Operação (ROW_COUNT): O comando SELECT ROW_COUNT() AS parcelas_atualizadas é utilizado para retornar ao chamador (o usuário ou a aplicação) o número exato de linhas que foram afetadas pelo UPDATE. Isso oferece um feedback claro e auditável sobre o resultado da operação de desconto.

4.2.3. Transcrição do Código SQL

4.2.3.1. MySQL

```
DELIMITER //
CREATE PROCEDURE sp_aplicar_desconto(IN p_matricula_id INT, IN p_percentual_desconto DECIMAL(5, 2))
BEGIN
   DECLARE v_fator_multiplicador DECIMAL(10, 6);

IF p_percentual_desconto IS NULL OR p_percentual_desconto <= 0 OR p_percentual_desconto > 100 THEN
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Percentual de desconto inválido (deve ser entre 0.01 e 100.00). Operação abortada.';
ELSE SET v_fator_multiplicador = 1 - (p_percentual_desconto / 100);
UPDATE mensalidades SET valor_nominal = valor_nominal * v_fator_multiplicador
WHERE matricula_id = p_matricula_id AND status_pag = 'Pendente';

SELECT ROW_COUNT() AS parcelas_atualizadas;
END IF;
```

```
END //
DELIMITER ;
```

4.2.3.2. PostgreSQL

```
CREATE OR REPLACE PROCEDURE escola_idiomas.sp_aplicar_desconto(
   p_matricula_id INT,
   p_percentual_desconto DECIMAL
)
LANGUAGE plpgsql
AS $$
DECLARE
   v_fator_multiplicador DECIMAL;
   v_linhas_afetadas INT;
BEGIN
   -- 1. Verificação de Segurança (Percentual de Desconto)
IF p_percentual_desconto IS NULL OR p_percentual_desconto <= 0 OR p_percentual_desconto > 100 THEN
RAISE EXCEPTION 'Percentual de desconto inválido (deve ser entre 0.01 e 100.00). Operação abortada.';
ELSE v_fator_multiplicador := 1 - (p_percentual_desconto / 100);
UPDATE escola_idiomas.mensalidades SET valor_nominal = valor_nominal * v_fator_multiplicador
WHERE matricula_id = p_matricula_id AND status_pag = 'Pendente';

GET DIAGNOSTICS v_linhas_afetadas = ROW_COUNT;
RAISE NOTICE 'Total de % parcelas pendentes atualizadas com desconto de %%%', v_linhas_afetadas, p_percentual_desconto;
   END IF;
END;
$$;
```

4.2.4. Casos de Teste (Descrição)

- TESTE DE SUCESSO (CENÁRIO POSITIVO). A Matrícula ID 2 tem 8 parcelas pendentes de 300.00. Aplicaremos um desconto de 10% (Multiplicador = 0.90). O valor nominal deve ser atualizado de 300.00 para 270.00 (300.00 * 0.90). O retorno esperado é uma tabela com o valor 8 (o número de parcelas atualizadas).

```
CALL sp_aplicar_desconto(2, 10.00);
-- Verificação 1: Confirma o número de linhas atualizadas
-- Retornará 'parcelas_atualizadas' = 8
-- Verificação 2: Confirma que os valores foram atualizados para 270.00
```

SELECT matricula_id, numero_parcela, valor_nominal, status_pag FROM mensalidades WHERE matricula_id = 2;

- TESTE DE FALHA (CENÁRIO NEGATIVO). Tentativa de aplicar um desconto de 120% (Inválido). O procedimento deve ser bloqueado pela cláusula IF inicial e retornar o erro 'Percentual de desconto inválido...'. O retorno esperado é o erro e 0 linhas afetadas.

```
-- ESTE COMANDO DEVE GERAR O ERRO: 'Percentual de desconto inválido (deve ser entre 0.01 e 100.00). Operação abortada.'
CALL sp_aplicar_desconto(2, 120.00);
-- Verificação (Confirma que NENHUMA parcela foi atualizada, permanecendo 270.00)
SELECT matricula_id, numero_parcela, valor_nominal, status_pag FROM mensalidades WHERE matricula_id = 2;
```

5. GATILHOS (Triggers)

5.1. Trigger 1: Guardião de Lotação

5.1.1. Enunciado ECA

Evento (Entrada): Tentativa de inserir uma nova linha na tabela matriculas (Momento: BEFORE INSERT).

Condição (Lógica): O sistema verifica se a contagem atual de alunos na turma de destino (NEW.turma_id) já atingiu ou ultrapassou a capacidade da sala associada àquela turma.

Ação (Saída): Se a sala estiver cheia: Bloquear a inserção e lançar um erro no banco de dados (ex: SIGNAL SQLSTATE no MySQL ou RAISE EXCEPTION no PostgreSQL) com a mensagem "Sala Lotada". Se houver vaga: Permitir a inserção normalmente.

5.1.2. Decisões de Implementação (Documentação)

Momento de Execução (BEFORE INSERT): O Trigger é executado antes que o registro seja fisicamente inserido na tabela. Isso permite o bloqueio da operação sem a necessidade de um ROLLBACK custoso (melhor performance).

Filtro de Status (status_mat = 'Ativa'): A contagem de alunos é rigorosa, considerando apenas matrículas com status 'Ativa'. Isso evita que alunos com matrículas 'Concluídas' ou 'Inativas' sejam contados indevidamente para o cálculo da lotação atual.

Verificação Preditiva: A condição 'if(alunos_ativos + 1) > capacidade_sala' prevê o estado futuro. O Trigger compara a lotação ANTES da inserção (alunos_ativos) com a capacidade, mas contando o novo aluno (+1), garantindo que a regra de lotação não seja violada.

5.1.3. Transcrição do Código SQL

5.1.3.1. SQL

```
drop trigger if exists tg_verificar_lotacao;
delimiter //
create trigger tg_verificar_lotacao
before insert on matriculas
for each row
begin
declare alunos_ativos int;
declare capacidade_sala int;
select s.capacidade into capacidade_sala from turmas t join salas s on s.sala_id = t.sala_id where turma_id = new.turma_id;
```

```
select count(matricula_id) into alunos_ativos from matriculas where turma_id = new.turma_id and status_mat = 'Ativa';
if(alunos_ativos + 1) > capacidade_sala then signal sqlstate '45000' set message_text = 'Sala Lotada. Matrícula bloqueada.';
end if;
end //
delimiter ;
```

5.1.3.2. PostgreSQL

```
CREATE OR REPLACE FUNCTION escola_idiomas.fn_verificar_lotacao()
RETURNS TRIGGER AS $$
DECLARE
   alunos_ativos INTEGER;
   capacidade_sala INTEGER;
BEGIN
   SELECT s.capacidade INTO capacidade_sala
   FROM escola_idiomas.turmas t
   JOIN escola_idiomas.salas s ON s.sala_id = t.sala_id
   WHERE t.turma_id = NEW.turma_id;

   SELECT COUNT(matricula_id) INTO alunos_ativos
   FROM escola_idiomas.matriculas
   WHERE turma_id = NEW.turma_id AND status_mat = 'Ativa';

   IF (alunos_ativos + 1) > capacidade_sala THEN
      RAISE EXCEPTION 'Sala Lotada. Matrícula bloqueada.';
   END IF;

   RETURN NEW; -- Retorna o registro para ser inserido (se não bloqueado)
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS tg_verificar_lotacao ON escola_idiomas.matriculas;
CREATE TRIGGER tg_verificar_lotacao
BEFORE INSERT ON escola_idiomas.matriculas
FOR EACH ROW
EXECUTE FUNCTION escola_idiomas.fn_verificar_lotacao();
```

5.1.4. Casos de Teste (Descrição)

- Este teste valida o cenário POSITIVO (Permissão). Usaremos a Turma ID 17 (ALE-A1-T02-2025) que tem Capacidade 15 e atualmente possui 1 aluno ativo (Matrícula ID 37). O sistema deve permitir a inserção de um novo aluno (Aluno

ID 29), resultando em 2/15. O retorno esperado é a Matrícula 118 ou superior, com status 'Ativa'.

```
INSERT INTO matriculas (aluno_id, turma_id, data_matricula, valor_total_curso, status_mat) VALUES
(29, 17, current_date(), 1800.00, 'Ativa');
-- Verificação (Deve retornar o novo registro)
SELECT M.matricula_id, P.nome AS Aluno, T.nome_turma FROM matriculas M JOIN pessoas P ON
M.aluno_id = P.pessoa_id JOIN turmas T ON M.turma_id = T.turma_id WHERE M.aluno_id = 29 AND M.turma_id = 17;
```

- Este caso testa o BLOQUEIO (Cenário Crítico). A Turma ID 19 (ITA-RAP-T02-2025) foi populada até sua capacidade máxima de 18 alunos. Este INSERT de um novo aluno (Aluno ID 30) DEVE FALHAR, pois (18 alunos ativos + 1 novo aluno) > 18 (Capacidade). O retorno esperado é o erro 'Sala Lotada. Matrícula bloqueada.' e o registro não deve existir.

```
-- ESTE COMANDO DEVE GERAR O ERRO: "Sala Lotada. Matrícula bloqueada."
INSERT INTO matriculas (aluno_id, turma_id, data_matricula, valor_total_curso, status_mat) VALUES
(30, 19, current_date(), 900.00, 'Ativa');
-- Verificação (Deve retornar 0 linhas, pois o INSERT foi bloqueado)
SELECT * FROM matriculas WHERE aluno_id = 30 AND turma_id = 19;
```

5.2. Trigger 2: Automação de Status Financeiro

5.2.1. Enunciado ECA

Evento (Entrada): Atualização de dados na tabela mensalidades (Momento: BEFORE UPDATE).

Condição (Lógica): O sistema detecta que o campo valor_pag foi preenchido pelo usuário (era NULL e agora (NEW.valor_pag) possui valor), indicando que um pagamento foi processado.

Ação (Saída): Alterar automaticamente o campo status_pag para 'Pago' e, caso a data_pag não tenha sido informada manualmente, definir a data atual como data de pagamento. Isso garante consistência mesmo que o operador esqueça de mudar o status manualmente.

5.2.2. Decisões de Implementação (Documentação)

Sinalização de Pagamento (NEW.valor_pag): A decisão de disparo do Trigger foca no preenchimento do campo 'valor_pag'. Isso garante que o status financeiro seja automatizado no momento em que a informação contábil mais relevante (o valor recebido) é registrada.

Controle de Status (OLD.status_pag): A cláusula "AND OLD.status_pag != 'Pago'" impede que o Trigger seja executado desnecessariamente em updates posteriores (como uma correção na forma de pagamento) de uma parcela que já foi quitada. Isso garante eficiência.

Consistência de Data (CURDATE()): O uso do condicional IF NEW.data_pag IS NULL em conjunto com CURDATE() (ou CURRENT_DATE no PostgreSQL) assegura que todo registro com status 'Pago' tenha uma data de pagamento associada, garantindo consistência mesmo que o operador esqueça de mudar o status manualmente.

5.2.3. Transcrição do Código SQL

5.2.3.1. SQL

```
drop trigger if exists tg_atualizar_status_pagamento;
delimiter //
create trigger tg_atualizar_status_pagamento
before update on mensalidades
for each row
begin
if new.valor_pag is not null and old.status_pag != 'Pago' then
        set new.status_pag = 'Pago';
        if new.data_pag is null then set new.data_pag = current_date();
        end if;
end if;
end //
delimiter ;
```

5.2.3.2. PostgreSQL

```
CREATE OR REPLACE FUNCTION escola_idiomas.fn_atualizar_status_pagamento()
RETURNS TRIGGER AS $$
BEGIN
   IF NEW.valor_pag IS NOT NULL AND OLD.status_pag <> 'Pago' THEN

      NEW.status_pag := 'Pago'; -- Em PL/pgSQL, usa-se := para atribuição

      IF NEW.data_pag IS NULL THEN
         NEW.data_pag := CURRENT_DATE;
      END IF;

   END IF;

   RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS tg_atualizar_status_pagamento ON escola_idiomas.mensalidades;
CREATE TRIGGER tg_atualizar_status_pagamento
BEFORE UPDATE ON escola_idiomas.mensalidades
FOR EACH ROW
EXECUTE FUNCTION escola_idiomas.fn_atualizar_status_pagamento();
```

5.2.4. Casos de Teste (Descrição)

- Este teste valida o cenário positivo de automação. A mensalidade ID 1, Parcela 4 está com status 'Pendente' (data de vencimento 2025-05-20). O sistema insere apenas o valor pago (300.00). O Trigger deve automaticamente mudar 'status_pag' para 'Pago' e preencher 'data_pag' com a data atual. O retorno esperado é status_pag = 'Pago' e data_pag preenchida.

```
UPDATE mensalidades SET valor_pag = 300.00 WHERE matricula_id = 1 AND numero_parcela = 4;
SELECT matricula_id, numero_parcela, status_pag, data_pag, valor_pag FROM mensalidades WHERE matricula_id = 1 AND numero_parcela = 4;
```

- Este caso testa a robustez do Trigger contra reprocessamento. A mensalidade ID 1, Parcela 4, já foi paga no teste anterior. Tentamos um update secundário (ex: mudar a forma de pagamento). O sistema não deve reverter ou mudar o status 'Pago', provando que a condição OLD.status_pag != 'Pago' funcionou.

```
UPDATE mensalidades SET forma_pagamento_id = 3 WHERE matricula_id = 1 AND numero_parcela = 4;
SELECT matricula_id, numero_parcela, status_pag, data_pag, valor_pag FROM mensalidades WHERE matricula_id = 1 AND numero_parcela = 4;
```