Atividade 2

Aluno: Max Humberto Recuero Jr.

1 – Introdução

Para colocar em prática os conceitos discutidos na disciplina de Banco de Dados 2, é proposto uma atividade prática à ser desenvolvida seguindo algumas especificações. O projeto a ser desenvolvido será um banco de dados para uma aplicação de vendas *mobile*. O banco será desenvolvido em PostgreSQL cumprindo alguns requisitos estabelecidos no projeto.

2 - Modelo Entidade Relacionamento

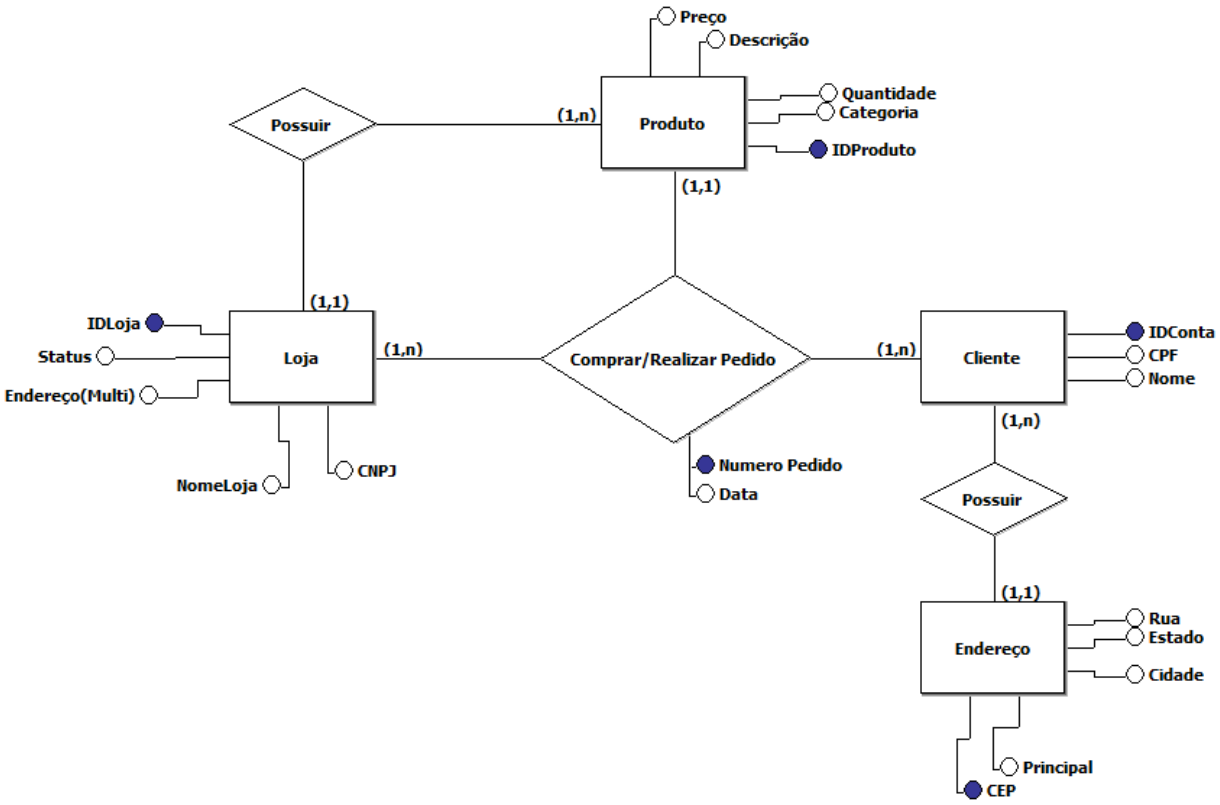

Figura 2.1: Construção do modelo Entidade-Relacionamento com base nos requisitos da aplicação em desenvolvimento.

3 - Populando as tabelas

A população das tabelas será realizada através do site *mockaroo*, gerando os dados necessários para realizar as consultas e analisar as tabelas.

4 - Criação de Índices

Realizando análises na aplicação, as consultas que mais devem ser realizadas são buscas por lojas abertas/fechadas, buscas por produtos contidos em uma loja e buscas por pedidos realizados.

```
1  Select nomeloja, status, idloja from Lojas
2  where status = true
3  order by nomeloja
```

Data Output | Explain | Messages | Notifications

	nomeloja character varying (100)	status boolean	idloja [PK] integer
1	Abbott-Huels	true	226
2	Ankunding, Ledner and Becht...	true	910
3	Aufderhar-Bechtelar	true	34
4	Aufderhar, Kuphal and Mueller	true	717
5	Bahringer, Kihn and Brown	true	645

Figura 4.1: Exemplo de busca de lojas com 'status = 'true'' onde o status é se a loja encontra-se aberta(true) ou fechada(false).

```
1  SELECT PRODUTOS.NOME, IDLOJA, PRODUTOS.QUANTIDADE, PRODUTOS.CATEGORIA,
2  PRODUTOS.IDPRODUTO, PRODUTOS.PRECO FROM PRODUTOS
3  NATURAL JOIN LOJAS
4  ORDER BY IDLOJA
```

Data Output | Explain | Messages | Notifications

	nome character varying (100)	idloja integer	quantidade integer	categoria character varying (16)	idproduto [PK] integer	preco character varying (8)
1	Juice - Orange	1	10	Comidas	3074	$279.27
2	Taylor County Goldaster	1	15	plantas e flores	931	£394.64
3	Dwarf Woodsorrel	1	3	plantas e flores	253	¥508.74
4	Winter Squash	2	0	plantas e flores	885	¥283.09
5	Taurus	2	4	Carros	1601	¥723.51

Figura 4.2: Exemplo de busca de produtos pertencentes à cada loja de acordo com a chave estrangeira 'idloja' na tabela Produtos, referente à chave primária de Lojas

```
1  SELECT P.CODIGOPEDIDO, P.DATAPEDIDO, P.IDLOJA, P.IDCONTA, P.IDPRODUTO, C.NOME FROM PEDIDOS P
2  NATURAL JOIN CLIENTES C
3  ORDER BY C.NOME
```

Data Output | Explain | Messages | Notifications

	codigopedido integer	datapedido date	idloja integer	idconta integer	idproduto integer	nome character varying (30)
1	1948	2019-12-17	55	742	1736	aahreniuskl
2	426	2020-08-07	839	645	301	aaiskrigghw
3	1761	2019-11-25	674	191	583	aashe5a
4	1403	2020-08-22	446	191	527	aashe5a
5	1681	2020-01-13	920	421	963	abaildonbo

Figura 4.3: Exemplo de busca de pedidos com base na chave estrangeira 'idconta' na tabela Pedidos, referente à chave primária de Clientes.

Agora será analisado o tempo de cada uma das consultas para realizar ou não indexação nestas consultas. Começando pela consulta de produtos, utilizando '*explain analyze*' na consulta realizada, tem-se um tempo de execução de 0.499 milissegundos sem a criação do índice. Para criar o índice será executado o comando na linha dois da figura 4.4 e analisado novamente os resultados. Na figura 4.5, o resultado da análise da consulta mostra um tempo de execução de 0.112 milissegundos, ou seja, com o índice criado obtém-se uma resposta quase cinco vezes mais rápido do que a consulta sem índice.

```
1  DROP INDEX IDX_PROD_PKIDLOJA
2  CREATE INDEX IDX_PROD_PKIDLOJA ON PRODUTOS(IDLOJA)
3  EXPLAIN ANALYZE
4  SELECT PRODUTOS.NOME, IDLOJA, PRODUTOS.QUANTIDADE, PRODUTOS.CATEGORIA,
5  PRODUTOS.IDPRODUTO, PRODUTOS.PRECO FROM PRODUTOS
6  NATURAL JOIN LOJAS
7  WHERE IDLOJA = 432
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.28..126.84 rows=5 width=45) (actual time=0.085..0.481 rows=8 loops=1)
2	-> Index Only Scan using pkloja on lojas (cost=0.28..8.29 rows=1 width=4) (actual time=0.014..0.015 rows=1 loops=1)
3	Index Cond: (idloja = 432)
4	Heap Fetches: 1
5	-> Seq Scan on produtos (cost=0.00..118.50 rows=5 width=45) (actual time=0.070..0.464 rows=8 loops=1)
6	Filter: (idloja = 432)
7	Rows Removed by Filter: 4992
8	Planning Time: 0.103 ms
9	Execution Time: 0.499 ms

Figura 4.4: Realização dos comandos '*explain analyze*' para análise de tempos de execução e custos da consulta de produtos em uma loja específica.

```
1  DROP INDEX IDX_PROD_PKIDLOJA
2  CREATE INDEX IDX_PROD_PKIDLOJA ON PRODUTOS(IDLOJA)
3  EXPLAIN ANALYZE
4  SELECT PRODUTOS.NOME, IDLOJA, PRODUTOS.QUANTIDADE, PRODUTOS.CATEGORIA,
5  PRODUTOS.IDPRODUTO, PRODUTOS.PRECO FROM PRODUTOS
6  NATURAL JOIN LOJAS
7  WHERE IDLOJA = 432
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
3	Index Cond: (idloja = 432)
4	Heap Fetches: 1
5	-> Bitmap Heap Scan on produtos (cost=4.32..19.90 rows=5 width=45) (actual time=0.022..0.033 rows=8 loops=1)
6	Recheck Cond: (idloja = 432)
7	Heap Blocks: exact=7
8	-> Bitmap Index Scan on idx_prod_pkidloja (cost=0.00..4.32 rows=5 width=0) (actual time=0.016..0.016 rows=8 loops=1)
9	Index Cond: (idloja = 432)
10	Planning Time: 0.167 ms
11	Execution Time: 0.112 ms

Figura 4.5: Execução da análise da consulta de produtos em loja específica após criação do índice.

Será verificado a consulta na tabela de pedidos, no mesmo formato da busca anterior. Realizando a análise presente na figura 4.6, percebe-se que o tempo de execução é de 0.284 milissegundos. Realizando varias consultas após a criação do índice, foi notado que algumas consultas com o índice aplicado podem demorar mais que a consulta sem índice, porém, em média, a maioria das consultas acaba sendo executada em média na metade do tempo. Na figura 4.7 mostra que a consulta tem tempo de execução de 0.059 milissegundos, o que mostra que a aplicação do índice beneficia o banco de dados.

```
1  EXPLAIN ANALYZE
2  SELECT P.CODIGOPEDIDO, P.DATAPEDIDO, P.IDLOJA, P.IDCONTA, P.IDPRODUTO, C.NOME FROM PEDIDOS P
3  NATURAL JOIN CLIENTES C
4  WHERE IDCONTA = 532
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=0.28..46.31 rows=2 width=30) (actual time=0.054..0.262 rows=2 loops=1)
2	-> Index Scan using pkcliente on clientes c (cost=0.28..8.29 rows=1 width=14) (actual time=0.012..0.014 rows=1 loops=1)
3	Index Cond: (idconta = 532)
4	-> Seq Scan on pedidos p (cost=0.00..38.00 rows=2 width=20) (actual time=0.040..0.244 rows=2 loops=1)
5	Filter: (idconta = 532)
6	Rows Removed by Filter: 1998
7	Planning Time: 0.128 ms
8	Execution Time: 0.284 ms

Figura 4.6: Execução da análise da consulta de pedidos com cliente específico antes de criar o índice.

```
1  CREATE INDEX IDX_PEDIDOS_FKIDCONTA ON PEDIDOS(IDCONTA)
2  EXPLAIN ANALYZE
3  SELECT P.CODIGOPEDIDO, P.DATAPEDIDO, P.IDLOJA, P.IDCONTA, P.IDPRODUTO, C.NOME FROM PEDIDOS P
4  NATURAL JOIN CLIENTES C
5  WHERE IDCONTA = 532
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=4.57..18.28 rows=2 width=30) (actual time=0.027..0.029 rows=2 loops=1)
2	-> Index Scan using pkcliente on clientes c (cost=0.28..8.29 rows=1 width=14) (actual time=0.010..0.011 rows=1 loops=1)
3	Index Cond: (idconta = 532)
4	-> Bitmap Heap Scan on pedidos p (cost=4.29..9.96 rows=2 width=20) (actual time=0.010..0.012 rows=2 loops=1)
5	Recheck Cond: (idconta = 532)
6	Heap Blocks: exact=2
7	-> Bitmap Index Scan on idx_pedidos_fkidconta (cost=0.00..4.29 rows=2 width=0) (actual time=0.006..0.006 rows=2 loops=1)
8	Index Cond: (idconta = 532)
9	Planning Time: 0.127 ms
10	Execution Time: 0.059 ms

Figura 4.7: Execução da análise da consulta de pedidos com cliente específico após criação do índice.

Finalmente, para a consulta de 'status' das lojas, onde 'A' representa a loja estar aberta e 'F' representa a loja estar fechada, será aplicado um índice *bitmap*, defido ao fato de existir apenas dois estados que 'status' possa assumir. Antes de criar o índice, é analisado o tempo de execução da consulta na figura 4.8, sendo este tempo de 3.994 milissegundos. Assim, aplicando o índice e fazendo a análise de várias consultas iguais tem que, em média, o tempo de execução caiu para 2.500 milissegundos.

```
4  EXPLAIN ANALYZE
5  SELECT NOMELOJA, STAT, IDLOJA FROM LOJAS
6  WHERE STAT = 'A'
7
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on lojas (cost=0.00..376.00 rows=5868 width=24) (actual time=0.364..3.796 rows=5868 loops=1)
2	Filter: ((stat)::text = 'A'::text)
3	Rows Removed by Filter: 6132
4	Planning Time: 0.083 ms
5	Execution Time: 3.994 ms

Figura 4.8: Análise da consulta na tabela Lojas por lojas que estejam abertas ('stat' = 'A').

```
3  CREATE INDEX IDX_STATUS_LOJAS ON LOJAS USING GIN (STAT)
4  EXPLAIN ANALYZE
5  SELECT NOMELOJA, STAT, IDLOJA FROM LOJAS
6  WHERE STAT = 'A'
7
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Bitmap Heap Scan on lojas (cost=61.48..360.83 rows=5868 width=24) (actual time=1.053..2.167 rows=5868 lo...
2	Recheck Cond: ((stat)::text = 'A'::text)
3	Heap Blocks: exact=82
4	-> Bitmap Index Scan on idx_status_lojas (cost=0.00..60.01 rows=5868 width=0) (actual time=1.032..1.032 ro...
5	Index Cond: ((stat)::text = 'A'::text)
6	Planning Time: 0.155 ms
7	Execution Time: 2.540 ms

Figura 4.9: Realização da consulta de lojas abertas após criação do índice.

5 - Criação de funções

Para a aplicação em desenvolvimento, não há necessidade de um cliente acessar os dados de todos os produtos para visualização dele, somente quando o cliente acessar um produto que todos os dados do produto devem ser carregados. Assim, para não executar a mesma *query* várias vezes, cria-se uma função para isto, representada na figura 5.1. Com isto, a consulta fica armazenada e pode-se ser chamada pela função.

```
1  CREATE or REPLACE FUNCTION GETPRODUTO (pid int)
2  RETURNS TABLE(QUANTIDADE INT, CATEGORIA VARCHAR(16), LOJAID int, PRECO VARCHAR(8), NOME VARCHAR(100)) AS $$
3  BEGIN
4      RETURN QUERY
5          SELECT P.quantidade, P.categoria, P.idloja, P.Preco, P.Nome FROM PRODUTOS as P
6          WHERE IDPRODUTO = PID;
7  END;
8  $$ LANGUAGE PLPGSQL
```

Figura 5.1: Código de criação da função para buscar dados de produtos armazenados no banco.

```
10  select getproduto(400)
```

Data Output | Explain | Messages | Notifications

	getproduto record
1	(13,"plantas e flores",598,$444.51,"Diaz Bluestem")

Figura 5.2: Execução da função criada para busca de dados de produtos. Na figura acima, o produto buscado é referente à chave primária de valor 400.

Para um usuário atualizar seu endereço como principal ou não, pode-se implementar uma função que realize esta tarefa no banco. Na criação da tabela endereços, foi criada uma variável 'principal' de valor *boolean*, que será para definir se um usuário gostaria de colocar o endereço como principal para caso ele utilize mais de um endereço (a funcionalidade de vários endereços não será implementada no trabalho). Para realizar esta tarefa, será declarado duas variáveis, uma para armazenar o valor da chave primária do endereço referente a conta passada como parâmetro da função e outra para

armazenar o valor da coluna ‘principal’ e realizar o tratamento da atualização.

```
1  CREATE OR REPLACE FUNCTION atualizarPrincipalEnd (iduser int)
2  returns VOID AS $$
3  DECLARE
4  CEPATT VARCHAR(8);
5  STAT bool;
```

Figura 5.3: Criação da função e etapa de declaração de variáveis para realizar a tarefa de alterar o endereço principal.

Com isto realizado, implementa-se na figura 5.4 o armazenamento dos valores que desejamos alterar e a tratativa da tarefa, onde o valor *boolean* selecionado na busca será levado para tratamento da tarefa, definindo se será atualizado para *false* ou *true*. Ainda na figura 5.5, foi realizado o tratamento para caso uma conta inválida seja passada nos parâmetros da função.

```
6  BEGIN
7      SELECT CEP INTO CEPATT FROM CLIENTES
8      WHERE IDCONTA = IDUSER;
9      SELECT PRINCIPAL INTO STAT FROM ENDERECOS
10     WHERE CEP = CEPATT;
11     IF STAT = TRUE THEN
12         UPDATE ENDERECOS SET PRINCIPAL = FALSE
13         WHERE CEP = CEPATT;
14     ELSE
15         UPDATE ENDERECOS SET PRINCIPAL = TRUE
16         WHERE CEP = CEPATT;
17     END IF;
18     RAISE NOTICE 'ENDERECO PRINCIPAL ATUALIZADO';
```

Figura 5.4: Etapa de busca e tratamento da tarefa da função de atualizar endereço.

```
19 EXCEPTION
20     WHEN NO_DATA_FOUND THEN
21         RAISE NOTICE 'NÃO EXISTE ESTE CLIENTE';
22 END;
23 $$ language plpgsql
```

Figura 5.5: Tratamento de exceção para a função de alterar endereço principal, onde é tratado o caso da inserção inválida de uma conta.

Outra função de consulta corriqueira que se pode implementar é a visualização do numero de pedidos que uma conta já realizou.

```
3  CREATE OR REPLACE FUNCTION GETPEDIDOS (IDACC INT)
4  RETURNS int AS $$
5  DECLARE
6  NUMPEDIDOS INT;
7  BEGIN
8      SELECT COUNT(*) INTO NUMPEDIDOS FROM PEDIDOS
9      WHERE IDCONTA = IDACC;
10     RETURN NUMPEDIDOS;
11 END;
12 $$ LANGUAGE PLPGSQL
```

Figura 5.6: Criação de função para realizar a contagem do número total de pedidos de um cliente.

6 - Visões Comuns

Na etapa de criação de visualizações comuns de tabelas, será criado visualizações que busquem apenas os dados necessários para cada consulta. Por exemplo, um cliente abre a aplicação e deseja comprar algo, porém primeiro ele deve selecionar a loja à qual deseja realizar as compras. Assim, não há necessidade de carregar todos os dados de uma loja, apenas seu nome e 'id' da loja. Na figura está representada a criação desta *view*.

```
1  CREATE VIEW LOJASABERTAS AS
2  select nomeloja, idloja from lojas
3  where stat = 'A'
4  order by idloja
```

Figura 6.1: Criação da visualização de dados necessários para a consulta de lojas abertas.

Em caso de um cliente estar carregando seus dados para visualizar, pode-se buscar todos os dados relacionados à este cliente, de forma a carregar o endereço em conjunto sem necessidade de carregar chaves primárias. Neste caso, o cliente pode verificar seus dados e possivelmente requerer uma alteração neles. Este tipo de visualização pode ser aplicado para todas as lojas com o mesmo objetivo.

```
1  CREATE VIEW CLIENTESDATA AS
2  SELECT C.NOME, C.CPF, E.CIDADE, E.ESTADO, E.RUA
3  FROM CLIENTES C, ENDERECOS E
4  WHERE C.CEP = E.CEP
```

Figura 6.2: Criação da *view* de dados de Clientes.

Outra *view* que podemos utilizar é para a tabela produtos, onde podemos armazenar os produtos e filtrar por produtos pertencente à apenas uma loja. Assim, criando a *view* para a tabela inteira, armazena-se os produtos que tem no banco e depois, utilizando a clausula *where*, filtrar os produtos de acordo com a loja que o usuário selecionar. Na figura 6.3, está representada a criação da *view* para os produtos e a consulta de filtragem de produtos.

```
1  CREATE VIEW produtosdata as
2  SELECT QUANTIDADE, IDLOJA, PRECO, NOME FROM PRODUTOS
3
4  SELECT * FROM PRODUTOSDATA
5  WHERE IDLOJA = 3
```

Figura 6.3: Criação da visualização dos dados da tabela produtos com a consulta de filtragem dos dados.

7 - Tabela de Auditoria

Na aplicação proposta no trabalho, será adicionada uma tabela de auditoria para a tabela de pedidos, de forma a manter um controle sobre esta tabela e ter um registro dos pedidos realizados na aplicação. Isto serve como um controle, ou histórico, de alterações que possam ocorrer na tabela, deixando-a menos vulnerável à inserções, atualizações e remoções de dados por conta de terceiros.

```
1  CREATE TABLE PEDIDOS_AUDITORIA(
2      OPERACAO CHAR, DATA_OP TIMESTAMP, USUARIO VARCHAR,
3      CODIGOPEDIDO INTEGER, DATAPEDIDO DATE, IDLOJA INTEGER,
4      IDCONTA INTEGER, IDPRODUTO INTEGER
5  )
6
```

Figura 7.1: Criação e estrutura da tabela de auditoria para pedidos.

Nas figuras 7.2 e 7.3, está representado os códigos utilizados para criar o trigger e o seus tratamentos na tabela de pedidos.

```
1  CREATE OR REPLACE FUNCTION process_ped_audit()
2  RETURNS TRIGGER AS $pedid_audit$
3  BEGIN
4  IF (TG_OP = 'DELETE') THEN
5      INSERT INTO PEDIDOS_AUDITORIA SELECT 'D', now(), user, OLD.*;
6      RETURN OLD;
7  ELSEIF (TG_OP = 'UPDATE') THEN
8      INSERT INTO PEDIDOS_AUDITORIA SELECT 'U', now(), user, NEW.*;
9      RETURN NEW;
10 ELSEIF (TG_OP = 'INSERT') THEN
11     INSERT INTO PEDIDOS_AUDITORIA SELECT  'I', now(), user, NEW.*;
12     RETURN NEW;
13 END IF;
14 RETURN NULL;
15 END;
16 $pedid_audit$ LANGUAGE PLPGSQL;
```

Figura 7.2: Criação da função que trata as atividades realizadas na tabela pedidos.

```
19 CREATE TRIGGER pedid_audit
20 AFTER INSERT OR UPDATE OR DELETE ON PEDIDOS
21 FOR EACH ROW EXECUTE PROCEDURE process_ped_audit();
22
```

Figura 7.3: Criação do trigger na tabela de pedidos para tratar e armazenar na tabela de auditoria dos pedidos.

Após criação da tabela de auditoria e suas funções, será realizado algumas tarefas e depois visualizado a tabela de auditoria que foi criada para conferir se todos os resultados estão corretos. Na figura 7.4 temos as tarefas realizadas e os resultados na tabela de auditoria.

```
25 INSERT INTO PEDIDOS (datapedido, idloja, idconta, idproduto)
26 VALUES ('31/5/2020', 600, 400, 2000),
27 ('18/10/2020', 450, 350, 1005)
28
29 UPDATE PEDIDOS SET IDLOJA = 341
30 WHERE CODIGOPEDIDO = 500
31
32 DELETE FROM PEDIDOS
33 WHERE CODIGOPEDIDO = 400 OR CODIGOPEDIDO = 410
34
```

Data Output | Explain | Messages | Notifications

	operacao character (1)	data_op timestamp without time zone	usuario character varying	codigopedido integer	datapedido date	idloja integer	idconta integer
1	I	2020-10-25 23:29:45.204203	postgres	2001	2020-05-31	600	400
2	I	2020-10-25 23:29:45.204203	postgres	2002	2020-10-18	450	350
3	U	2020-10-25 23:29:48.922422	postgres	500	2019-11-09	341	509
4	D	2020-10-25 23:29:52.043292	postgres	400	2020-07-30	699	175
5	D	2020-10-25 23:29:52.043292	postgres	410	2019-11-08	814	926

Figura 7.4: Tarefas realizadas na tabela de pedidos e os resultados armazenados na tabela de auditoria após estas tarefas serem completas

8 - Criação de Triggers

A utilização de *triggers* na aplicação pode ser utilizada para o caso de usuários efetuarem compras em um produto cuja quantidade seja zero, além de ser utilizado para diminuir a quantidade quando este

produto for comprado. Assim, nas figuras 8.1 e 8.2, temos a criação deste *trigger* e sua função que trata os casos.

```
1  CREATE FUNCTION quantidade_av() returns TRIGGER AS $$
2  DECLARE
3  QNT INT;
4  BEGIN
5      SELECT QUANTIDADE INTO QNT FROM PRODUTOS
6      WHERE IDPRODUTO = NEW.IDPRODUTO;
7
8      IF(QNT = 0) THEN
9      RAISE EXCEPTION 'PRODUTO INDISPONIVEL';
10     END IF;
11
12     UPDATE PRODUTOS SET QUANTIDADE = QUANTIDADE - 1
13     WHERE IDPRODUTO = IDPROD;
14     RETURN NEW;
15 END;
16 $$ LANGUAGE PLPGSQL;
```

Figura 8.1: Criação da função que verifica se o produto tem quantidade maior que zero para realizar a compra. 'IDPROD' foi substituído por 'NEW.IDPRODUTO'.

```
19 CREATE TRIGGER CHECK_QNT
20 BEFORE INSERT ON PEDIDOS
21 FOR EACH ROW EXECUTE PROCEDURE QUANTIDADE_AV();
```

Figura 8.2: Criação do *trigger* que leva à verificação da quantidade do produto disponível.

Outra situação em que um cliente não deve efetuar uma compra será quando a loja estiver fechada, por tanto, mesmo se o cliente acesse a loja e visualize seus produtos, ele deve ser incapaz de comprar ou efetivar compras naquela loja. Caso o *front-end* e *back-end* da aplicação estivesse sendo desenvolvida, esta verificação seria implementada lá, porém, caso ocorra algum erro, o banco deve entender a situação e também realizar a verificação. Assim, cria-se um *trigger* para tratar este evento.

```
1  CREATE FUNCTION LOJA_STAT() RETURNS TRIGGER AS $$
2  DECLARE
3  S VARCHAR(2);
4  BEGIN
5      SELECT STAT INTO S FROM LOJAS
6      WHERE IDLOJA = NEW.IDLOJA;
7      IF(S = 'F') THEN
8          RAISE EXCEPTION 'A LOJA ENCONTRA-SE FECHADA';
9      END IF;
10     RETURN NEW;
11 END;
12 $$ LANGUAGE PLPGSQL;
13
14 CREATE TRIGGER VER_STAT
15 BEFORE INSERT ON PEDIDOS
16 FOR EACH ROW EXECUTE PROCEDURE LOJA_STAT();
```

Figura 8.3: Criação do trigger para verificar se o pedido realizado está em uma loja 'aberta', caso não esteja, este pedido é invalidado.

Como na aplicação, na tabela de produtos, está sendo utilizado dados de quantidade, cujo valor deve ser maior ou igual a zero, não faz sentido armazenar valores negativos para este campo. Por isso, pode-se criar um *trigger* para tratar inserções e atualizações destes valores, assim impedindo que a

quantidade de um produto seja negativa.

```
CREATE FUNCTION QNT_VER_INSorUP () RETURNS TRIGGER AS $$
DECLARE
Q int;
BEGIN
    SELECT NEW.QUANTIDADE INTO Q;
    IF(Q<0) THEN
        RAISE EXCEPTION 'QUANTIDADE DEVE SER POSITIVA OU ZERO';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER QNT_INS_UP
BEFORE INSERT OR UPDATE ON PRODUTOS
FOR EACH ROW EXECUTE PROCEDURE QNT_VER_INSorUP()
```

Figura 8.4: Criação do *trigger* para tratar quantidades negativas de produtos durante inserção ou atualização de dados.

Seguindo a mesma lógica aplicada para a quantidade, pode-se aplicar para a coluna de preços do produto, restringido a inserção ou atualização de produtos com preço menor ou igual a zero. Por isso, cria-se a função representada na figura 8.5 para realizar a verificação do valor do preço.

```
CREATE FUNCTION VER_PRECO() RETURNS TRIGGER AS $$
DECLARE
PR VARCHAR;
PRNUM DECIMAL(8,2);
BEGIN
    SELECT NEW.PRECO INTO PR;
    SELECT SUBSTRING(PR FROM 2)::DECIMAL(8,2) INTO PRNUM;
    IF(PRNUM <= 0) THEN
        RAISE EXCEPTION 'PRECO DEVE SER MAIOR QUE ZERO';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE PLPGSQL

CREATE TRIGGER PRECO_VALIDO
BEFORE INSERT OR UPDATE ON PRODUTOS
FOR EACH ROW EXECUTE PROCEDURE VER_PRECO()
```

Figura 8.5: Criação da função e trigger que verificam se o preço a ser alterado ou inserido é válido.

Ainda sim, o valor do preço está sujeito a inserção de preços com vírgula separando os centavos, sendo que o banco, já que não foi alterado, reconhece apenas com ponto. Com isto, ainda deve-se verificar como esta sendo a inserção e realizar uma normalização dos dados para não ocorrem erros no futuro.