Competências, Habilidades e Bases Tecnológicas da disciplina de Banco de Dados III.

III.2 BANCO DE DADOS III					
Função: Otimização da busca de informações no banco de dados Classificação: Execução					
Atribuições e Responsabilidades					
• Realizar gestão de bancos de dados.					
Valores e Atitudes					
• Incentivar a criatividade. • Desenvolver a criticidade. • Fortalecer a persistência e o interesse na resolução de situações-problema.					
Competências			Habilidades		
1. Otimizar a linguagem de consulta estruturada como forma de informação relevante para a tomada de decisão.			1.1 Executar linguagem de consulta estruturada objetivando melhor desempenho. 1.2 Compilar relatórios analíticos a partir dos dados coletados.		
Orientações					
• Detalhamento das Bases Tecnológicas - Anexo I					
Bases Tecnológicas					
Blocos de linguagem de consulta estruturada *(SQL)* Exceções (tratamentos de erros) Funções Gatilhos Visões Controladas Índices Merge e Permissões					
Carga horária (horas-aula)					
Teórica	00	Prática em Laboratório*	60	Total	60 Horas-aula
Teórica (2,5)	00	Prática em Laboratório* (2,5)	50	Total (2,5)	50 Horas-aula
* Possibilidade de divisão de classes em turmas, conforme o item 4.8 do Plano de Curso. * Todos os componentes curriculares preveem prática, expressa nas habilidades, relacionadas às competências. Para este componente curricular está prevista divisão de classes em turmas.					
Para ter acesso às titulações dos Profissionais habilitados a ministrarem aulas neste componente curricular, consultar o site: http://www.cpscetec.com.br/crt/					

1-) Comunicação de alunos com alunos e professores:

- Um e-mail para a sala é de grande valia para divulgação de material, notícias e etc.
- Criação de grupo nas redes sociais também é interessante.

2-) Uso de celulares:

Para o bom andamento das aulas, recomendo que utilizem os celulares em vibracall, para não atrapalhar o andamento da aula.

3-) Material das aulas:

A disciplina trabalha com NOTAS DE AULA que são disponibilizadas ao final de cada aula.

4-) Prazos de trabalhos e atividades:

Toda atividade solicitada terá uma data limite de entrega, de forma alguma tal data será postergada, ou seja, se não for entregue até a data limite a mesma receberá menção I, isso tanto para atividades entregues de forma impressa ou enviadas ao e-mail da disciplina. No caso de PTCC o calendário será seguido, e no dia marcado de determinada atividade só receberão menção os alunos **presentes**, caso não estejam a menção para aquela atividade será I.

5-) Qualidade do material de atividades:

- Impressas ou manuscritas:
  Muita atenção na qualidade do que será entregue, atividades sem grampear, faltando nome e número de componentes, rasgadas, amassadas, com rebarba de folha de caderno e etc. serão desconsiderados por mim.

- Digitais:
  Ao enviarem atividades para o e-mail da disciplina, SEMPRE no assunto deverá ter o nome da atividade que está sendo enviada, e no corpo do e-mail deverá ter o(s) nome(s) do(s) integrante(s) da atividade, sem estar desta forma a atividade será DESCONSIDERADA.

6-) Menções e critérios de avaliação:

Na ETEC os senhores serão avaliados por MENÇÃO, onde temos:
- MB - Muito Bom;
- B - Bom;
- R - Regular;
- I - Insatisfatório.

Cada trimestres teremos as seguintes formas de avaliação:
- 1 avaliação teórica;
- 1 avaliação prática (a partir do 2º trimestre, e as turmas do 2º módulo em diante);
- Seminários;
- Trabalhos teóricos e/ou práticos;
- Assiduidade;
- Outras que se fizerem necessário.

Caso o aluno tenha alguma menção I em algum dos trimestres será aplicada uma recuperação, que poderá ser em forma de trabalho prático ou teórico.

1. Revisão SQL

1.1 O MySQLWorkBench.

No decorrer desse curso veremos uma série de recursos de gerenciamento ligados ao Sistema Gerenciador de Banco de Dados Relacionais (**SGBDR**) "**Mysql**", esses recursos visam desde a manutenção dos bancos de dados e suas tabelas através de recursos como "**triggers**", "**stored procedures**", "**views**", "**index**" recursos esses fundamentais para a boa "**saúde**" de um banco de dados no que se refere a consistência dos dados e performance, passando também pela parte de segurança referente a criação e gestão de usuários e atribuição e revogação de direitos a esses usuários além de formas de realizar backups das bases de dados existentes.

Tais tarefas podem se mostrar desafiadoras ao "**Database Administrator**" (**DBA**) profissional responsável por tais tarefas, logo se faz necessário o uso de uma ou mais

ferramentas que permita um alto grau de produção para esse profissional. Existe uma gama enorme de ferramentas capazes de fornecer esse auxilio para os mais diversos SGBDR corporativos existentes hoje no mercado. No nosso caso estamos trabalhando com o "**MySQL**" e vamos utilizar uma "**Integrated Development Envriomento**" (**IDE**) ou "**Ambiente Integrado de Desenvolvimento**" conhecido como "**MySQL Workbench**".

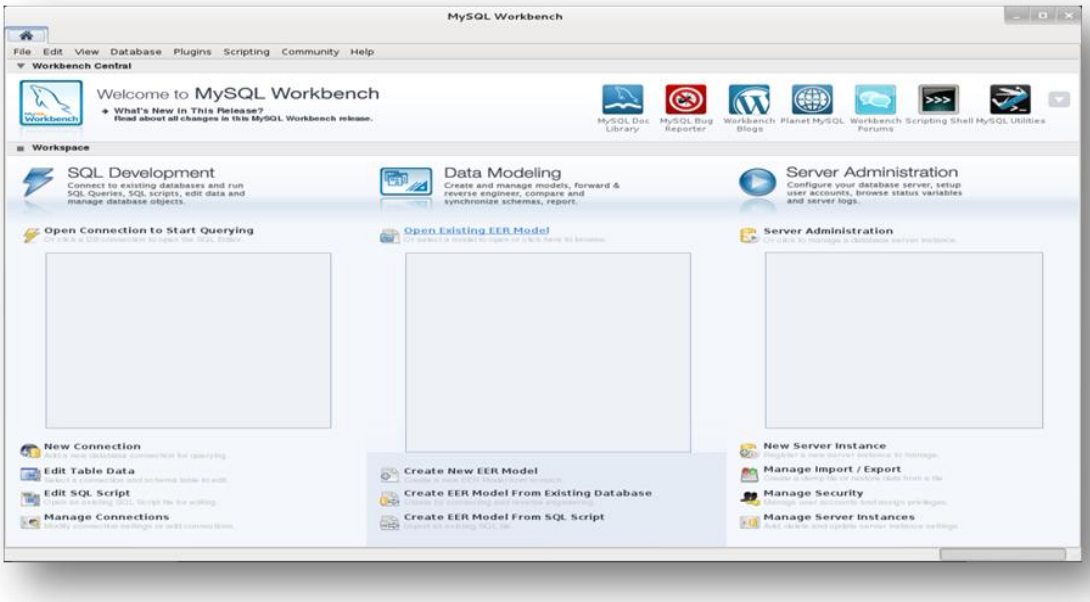
Essa ferramenta reúne dentro de si recursos divididos em três grupos básicos são eles:

- **SQL Development –** Reúne ferramentas ligadas a codificação de comandos **SQL Data Manipulation Language ou Linguagem de Manipulação de Dados** (**DML**), **Data definition Language** ou **Linguagem de Definição de Dados** (**DDL**), **Data Control Language** ou **Linguagem de controle de Dados** (**DCL**), **Data Transaction Language** ou **Linguagem de Transação de Dados** (**DTL**) e **Data Query Language ou Linguagem de Consulta de dados** (**DQL**).

- **Data Modeling –** Reúne ferramentas ligadas a modelagem de banco de dados permitindo a criação de diagramas/modelos de entidade relacionamento, além disso é possível converter esses diagramas em projetos físicos (bancos de dados) através de interface gráfica e podemos também realizar processos de engenharia reversa onde podemos "apontar" para um projeto físico de um determinado banco de dados e converter esse em diagramas/modelos de entidade relacionamento isso é muito útil principalmente quando estamos trabalhando em projetos que foram legados (já existiam e nos foram passados a título de continuação) e por alguma razão qualquer a equipe anterior não documentou a base de dados ou até mesmo essa documentação foi perdida.

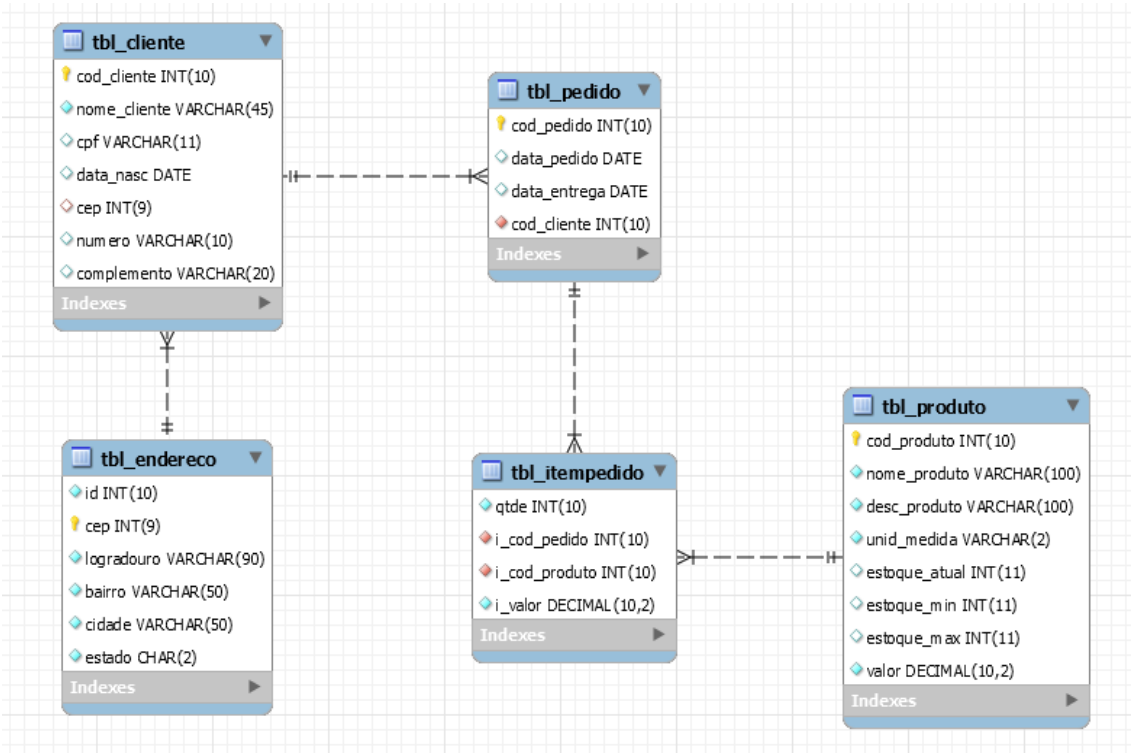
- **Server Administrator –** Reúne ferramentas uteis para a gerencia de backups e da "saúde" das bases de dados criadas na instancia do "**MySQL**" que se está gerenciando, permite por exemplo verificar quantas conexões estão ativas nas bases de dados existentes, verificar a "saúde" da memória principal do servidor que hospeda a instancia do "**MySQL**", o trafego de dados, índice de eficiência das chaves, criar e associar a usuários diretos ou remover usuários e direitos, realizar backups entre outras ações.

A imagem abaixo ilustra a tela inicial do "**MySQL WorkBench**" onde podemos escolher entre os grupos de ferramentas citadas:



1.2 SQL

Vamos aplicar uma revisão de BD II, fazendo um banco de dados de pedidos onde iremos seguir o DER abaixo e criar as tabelas com seus relacionamentos, veja:



Mas iremos fazer esse banco de dados através de comandos SQL, primeiramente vamos organizar as coisas, iremos criar um Script SQL chamado “bd_vendas DDL” onde irá conter a criação do banco de dados e as tabelas mostradas no DER anterior.

- Criando e habilitando a base de dados:

```
/*
CRIAÇÃO DO BANCO DE DADOS BD_VENDAS - TLBD III
*/
create database bd_vendas;
-- -----------------------------------------------------

/*
HABILITANDO O BANCO DE DADOS PARA USO
*/
use bd_vendas;
-- -----------------------------------------------------
```

- Criando a tabela de produtos:

```
/*
CRIAÇÃO DA TABELA DE PRODUTOS
*/
create table tbl_produto (
  cod_produto   int unsigned auto_increment,
  nome_produto  varchar(100) not null,
  desc_produto  varchar(100) not null,
  unid_medida   varchar(2) not null,
  estoque_atual int default 0,
  estoque_min   int default 0,
  estoque_max   int default 0,
  valor         decimal(10,2) not null,
  primary key (cod_produto));
-- -----------------------------------------------------
```

- Criando a tabela de endereços para armazenar os CEPs:

```
/*
CRIAÇÃO DA TABELA DE ENDEREÇO - CEP
*/
create table tbl_endereco (
  id int(10) not null,
  cep int(9) not null,
  logradouro varchar(90) not null,
  bairro varchar(50) not null,
  cidade varchar(50) not null,
  estado char(2) not null,

  constraint pk_endereco primary key (cep)
);
```

- Criando a tabela de clientes:

```
/*
CRIAÇÃO DA TABELA DE CLIENTES
*/
create table tbl_cliente (
  cod_cliente  int unsigned auto_increment,
  nome_cliente varchar(45) not null,
  cpf          varchar(11) default '',
  data_nasc    date,
  cep          int(9) default 0,
  numero       varchar(10) default '',
  complemento  varchar(20) default '',

  primary key (cod_cliente),
  constraint foreign key fk_cliencep (cep) references tbl_endereco(cep)
  );

-- ----------------------------------------------------------------------
```

- Criando a tabela de pedidos:

```
/*
CRIAÇÃO DA TABELA DE PEDIDOS
*/
create table tbl_pedido (
   cod_pedido   int unsigned auto_increment,
   data_pedido  date,
   data_entrega date,
   cod_cliente  int unsigned not null,
   primary key (cod_pedido),
   constraint fk_cliente foreign key (cod_cliente)
   references tbl_cliente(cod_cliente));
-- ------------------------------------------------------------
```

- Criando a tabela de itens do pedido:

```
/*
CRIAÇÃO DA TABELA DE ITENS DO PEDIDO
*/
create table tbl_itempedido (
   qtde          int unsigned not null,
   i_cod_pedido  int unsigned not null,
   i_cod_produto int unsigned not null,
   i_valor         decimal(10,2) not null,
   constraint fk_pedido1
     foreign key (i_cod_pedido)
     references tbl_pedido (cod_pedido),
   constraint fk_tbl_produto1
     foreign key (i_cod_produto)
     references tbl_produto(cod_produto));
-- ------------------------------------------------------------
```

Feito isso salvem esse arquivo e mantenha sempre com vocês em nossas aulas, pois iremos incrementar mais tabelas no decorrer do curso.

Caso queiram para verificar se a construção do seu banco de dados ficou igual a mencionada inicialmente, façam a engenharia reversa (*Reverse Engineer*) no menu *DATABASE* para verificar se ficou igual.

Popular as tabelas

Iremos agora inserir dados nas tabelas para que possamos dar seguimento aos nossos estudos, mas a primeira pergunta que vem a nossa mente é: "Por qual tabela devo começar?". Para essa resposta precisamos analisar o DER e verificar quais tabelas não possuem dependências de chaves estrangeiras e começar pelas mesmas, assim as tabelas de endereço e produto devem ser as primeiras a serem populadas, pois são tabelas que fornecem suas chaves para o preenchimento de outras, se fizermos ao contrário, a inserção dará erro.

Para isso, vamos criar outro Script SQL chamado "bd_vendas DML", onde iremos colocar todos os *INSERTs* que iremos fazer para popular essas tabelas, esse também deverá acompanhar os senhores para nossas aulas.

Para darmos início, vamos popular a tabela de endereço com os CEPs, vou disponibilizar um Script com esses *inserts* com as cidades de Taboão da Serra e Embu das Artes, darão cerca de 2000 registros nessa tabela.

Na sequência vamos inserir os produtos:

```
insert into tbl_produto (nome_produto, desc_produto, unid_medida,
estoque_atual,estoque_min, estoque_max,valor) values
 ('Arroz', 'Arroz agulhinha tipo 1','SC', 10,2,20, 12.50),
 ('Feijão', 'Feijão carioquinha com casca','SC', 25,5,60, 7.50),
 ('Macarrão', 'Macarrão Adria espaguete','PC', 50,10,80, 5.50),
 ('Óleo', 'Óleo Lisa','LT', 15,10,45, 6.50),
 ('Vinagre', 'Vinagre Castelo','GR', 30,10,50, 7.89),
 ('Batata', 'Batata lavada','KG', 100,50,200, 4.50),
 ('Tomate', 'Tomate vermelho','KG', 80,8,160, 6.90),
 ('Cebola', 'Cebola com casca','KG', 50,5,100, 6.99),
 ('Leite', 'Leite Leco','CX', 25,10,90, 2.50),
 ('Café', 'Café do Ponto','SC', 500,100,200, 11.50);
```

Feito isso, podemos realizar as inserções dos clientes, pois sem os dados da tabela de CEPs não seria possível chegar a essa etapa. **Só lembrando que para a inserção dessa tabela, o CEP informado deverá estar cadastrado na tabela de CEPs.**

```
/*
   Clientes
*/
insert into tbl_cliente(nome_cliente,cpf,data_nasc,cep,numero,complemento) values
('Marcos Costa de Sousa','12345678901','1981-02-06',6768100,'1525','apto 166C'),
('Zoroastro Zoando','01987654321','1989-06-15',6757190,'250',''),
('Idelbrandolâncio Silva','54698721364','1974-09-27',6753001,'120',''),
('Cosmólio Ferreira','41368529687','1966-12-01',6753020,'25','apto 255 F'),
('Conegunda Prado','54781269501','1950-10-06',6753020,'50','apto 166C'),
('Brogundes Asmônio','41256398745','1940-05-10',6753400,'100',''),
('Iscruência da Silva','12457965823','1974-11-25',6803040,'5',''),
('Zizafânio Zizundo','54123698562','1964-08-14',6803140,'25',''),
('Ricuerda Zunda','21698534589','1934-10-14',6803045,'123',''),
('Aninoado Zinzão','25639856971','1976-12-25',6803070,'50','');
```

Exercício

"Populem" as tabelas de acordo com o MER desenvolvido por vocês, com uma massa de dados consistente e concisa seguindo os seguintes critérios:

1º - Criem no mínimo 5 pedidos;
2º - Para cada pedido, pelo menos 3 itens em cada;
3º - Para evidenciar, em um arquivo .doc enviem:

- INSERT da tabela de pedido;
- INSERT da tabela de itempedido;

Para a próxima aula dia 08/08.

1. 3 *Transactions* em banco de dados

Transação ou *Transaction* é uma única unidade

de trabalho processada pelo Sistema de Gerenciamento de Banco de Dados (SGBD). Imagine que precisamos realizar em uma única transação duas operações de manipulação de dados (DML, do inglês Data Manipulation Language), como *INSERT, DELETE* e *UPDATE*. Estas operações só podem se tornar permanentes no banco de dados se todas forem executadas com sucesso. Em caso de falhas em uma das duas é possível cancelar a transação, porém todas modificações realizadas durante a mesma serão descartadas, inclusive a que obtive sucesso. Assim, os dados permanecem íntegros e consistentes.

Podemos caracterizar as transações conforme abaixo:

O *BEGIN TRANSACTION* indica onde ela deve começar, então os comando SQL a seguir estarão dentro desta transação.

O *COMMIT TRANSACTION* indica o fim normal da transação, o que tiver de comando depois já não fará parte desta transação. Neste momento tudo o que foi manipulado passa fazer parte do banco de dados normalmente e operações diversas passam enxergar o que foi feito.

O *ROLLBACK TRANSACTION* também fecha o bloco da transação e é a indicação que a transação deve ser terminada, mas tudo que tentou ser feito deve ser descartado porque alguma coisa errada aconteceu e ela não pode terminar normalmente. Nada realizado dentro dela será perdurado no banco de dados.

Ao contrário do que muita gente acredita *rollback* no contexto de banco de dados não significa reverter e sim voltar ao estado original. Um processo de reversão seria de complicadíssimo à impossível.

A maioria dos comandos SQL são transacionais implicitamente, ou seja, ele por si só já é uma transação. Você só precisa usar esses comandos citados quando precisa usar múltiplos comandos.

2. Comandos DML

Vamos neste momento alternar nossas aulas de acordo com o que foi visto até agora, neste capítulo iremos ver *UPDATE, INSERT, DELETE* e principalmente o *SELECT*, nos aprofundando na DQL, que é uma ramificação da DML, tudo isso para melhor fixação do conteúdo, que é vital neste momento do nosso curso. Apesar de termos as tabelas criadas e populadas conforme aula anterior, neste primeiro momento iremos nos ater em trabalhar somente com a tabela de Clientes **TBL_CLIENTE** e Endereço **TBL_ENDERECO**, em seguida passamos para as outras tabelas até finalizarmos fazendo os todos os procedimentos vistos com as chaves estrangeiras criadas nestas tabelas.

2.1 – SELECT

De acordo com a tabela “populada”, podemos retirar informações ricas da mesma, para isso iremos ver a DQL, que nos ajudará muito nesta extração de informações.

2.1.1 – CLÁUSULA DISTINCT

A cláusula DISTINCT permite que os dados repetidos de uma tabela sejam exibidos uma única vez, vamos observar o exemplo abaixo:

```
select distinct bairro
from tbl_endereco;
```

Neste caso somente serão mostrados os bairros distintos existentes na **TBL_ENDERECO,** ou seja, aparecerá no resultado 202 linhas, então podemos dizer que a tabela de endereço da nossa base de dados possui 202 bairros diferentes em seu cadastro.

Como qualquer SELECT, podemos refinar mais a consulta, por exemplo se quisermos ver a quantidade distintas de bairro por município por exemplo:

```
select distinct bairro
from tbl_endereco
where cidade = 'Taboão da Serra';
```

Nesse outro exemplo filtramos pelo município de Taboão da Serra, onde foram apresentadas 98 linhas, ou seja, em Taboão da Serra a nossa tabela possui 98 bairros diferentes.

2.1.2 – CLÁUSULA LIMIT

A cláusula LIMIT é usada para determinar um limite para a quantidade e para a faixa de linhas que podem ser retornadas, vamos neste nosso exemplo mostrar as linhas referentes aos clientes ISCRUÊNCIA e ZIZAFÂNIO observar o exemplo abaixo:

```
select *
from tbl_cliente
limit 6,2
```

Observe que o primeiro parâmetro de limite especificado (6) refere-se ao deslocamento, ou seja, ao ponto inicial, ao passo que o segundo parâmetro (2) refere-se a quantidade de linhas a serem retornadas.

Quando um único parâmetro é informado, este se refere a quantidade de linhas retornadas.

2.1.3 – CLÁUSULA ORDER BY

A cláusula ORDER BY **pode** ser usada juntamente com a cláusula LIMIT quando for importante que as linhas sejam retornadas de acordo com uma determinada ordem, observar o exemplo abaixo:

```
select *
from tbl_cliente
order by nome_cliente
limit 6,2;
```

Esta ordem também poderá ser ASCENDENTE ou DESCENDENTE dependendo da minha necessidade, veja:

```
select *
from tbl_cliente
order by nome_cliente desc
limit 6,2;
```

Ou

```
select *
from tbl_cliente
order by nome_cliente asc
limit 6,2;
```

Quando não informamos a classificação depois da cláusula ORDER BY por default é considerado ASC.

2.1.4 – CLÁUSULA WHERE

A cláusula *WHERE* serve para determinar quais os dados de uma tabela que serão afetados pelos comandos *SELECT, UPDATE* e *DELETE*, podemos dizer então que esta cláusula determina o escopo de uma consulta a algumas linhas, realizando uma filtragem dos dados que estejam de acordo com as condições definidas, vamos observar o exemplo abaixo:

```
select *
from tbl_cliente
where cpf = '12345678901';
```

Neste caso trará os dados do cliente “MARCOS COSTA DE SOUSA” que possui o CPF 12345678901.

```
select *
from tbl_endereco
where bairro = 'Jardim Caner'
```

Neste outro caso os dados de endereço que são do Bairro JD CANER serão mostrados.

```
select *
from tbl_endereco
where cidade = 'Taboão da Serra'
```

Neste outro caso os dados de endereço que são da Cidade de TABOÃO DA SERRA serão mostrados.

2.1.5 – OPERADORES AND E OR

Os operadores lógicos AND e OR são utilizados junto com a cláusula WHERE quando for necessário utilizar mais de uma condição de comparação, vamos observar o exemplo abaixo:

```
select * from tbl_endereco
where estado = 'SP'
  and cidade = 'Taboão da Serra'
```

Neste caso todos endereços da UF de SP e a CIDADE de TABOÃO DA SERRA serão mostrados.

```
select * from tbl_endereco
where estado = 'SP'
   or cidade = 'Taboão da Serra';
```

Neste caso todos os endereços da UF de SP ou da CIDADE de TABOÃO DA SERRA serão mostrados, em nosso exemplo irão aparecer todos os dados cadastrados, em caso de uma base de dados de todo o Brasil por exemplo, se existisse a cidade de Taboão da Serra na Bahia por exemplo, seria mostrada no resultado.

2.1.6 – OPERADOR IN

O operador IN permite checar se o valor de uma consulta está presente em uma lista de elementos.

```
select * from tbl_endereco
where bairro in ('Jardim Kuabara','Jardim Pazini');
```

Neste caso todos os Endereços dos BAIRROS Jardim Kuabara e Jardim Pazini serão mostrados.

Com o NOT antecedido do IN, faz a operação inversa, ou seja não trará os endereços dos BAIRROS Jardim Kuabara e Jardim Pazini.

```
select * from tbl_endereco
where bairro not in ('Jardim Kuabara','Jardim Pazini');
```

2.1.7 – OPERADOR LIKE

O operador LIKE é empregado quando a base para realizar pesquisa forem as colunas que estão no formato char ou varchar, vamos observar o exemplo abaixo:

```
select * from tbl_cliente
where nome_cliente like 'M%'
```

Neste caso todos os clientes que possuem a letra M no início do seu nome serão mostrados.

```
select * from tbl_cliente
where nome_cliente like '%M%';
```

Neste caso todos os clientes que possuem a letra M em qualquer parte do seu nome serão mostrados.

```
select * from tbl_cliente
where nome_cliente like '%M';
```

Neste caso todos os clientes que possuem a letra M no fim do seu nome serão mostrados.

Com o NOT antecedido do LIKE, faz a operação inversa, ou seja não trará os clientes dos nas situações acima, veja:.

```
select * from tbl_cliente
where nome_cliente not like 'M%';
```

Neste caso todos os clientes que possuem a letra M no início do seu nome não serão mostrados.

```
select * from tbl_cliente
where nome_cliente not like '%M%';
```

Neste caso todos os clientes que possuem a letra M em qualquer parte do seu nome não serão mostrados.

```
select * from tbl_cliente
where nome_cliente not like '%M';
```

Neste caso todos os clientes que possuem a letra M no fim do seu nome não serão mostrados.

```
select * from tbl_cliente
where nome_cliente like 'M_R__S%';
```

Neste caso todos os clientes que possuem na 1ª posição de seus nomes letra M, na 3ª posição a letra R e na 5ª posição a letra S e não importando o restante do nome serão mostrados, nesse caso adivinhem o nome de quem aparecerá!!!

2.1.8 – OPERADOR BETWEEN

O operador BETWEEN tem a função de permitir a consulta de uma determinada faixa de valores. Este operador permite checar se o valor de uma coluna encontra-se em um determinado intervalo, vamos observar o exemplo abaixo:

```
select * from tbl_produto
where estoque_min between 10 and 100
```

Neste caso serão mostrados todos os registros que possuem o campo ESTOQUE_MIN entre 10 e 100.

Outro exemplo, podemos usar o NOT BETWEEN

```
select * from tbl_produto
where estoque_min not between 10 and 100
```

Neste caso é feito o contrário do exemplo dado, ele trará os registro que NÃO estejam com o campo ESTOQUE_MIN entre 10 e 100.

2.1.9 – CLÁUSULA COUNT

A cláusula *COUNT* serve para contarmos quantas ocorrências existe um determinado *SELECT*, vamos observar o exemplo a seguir:

```
select count(*) from tbl_produto
where estoque_min between 10 and 100
```

Neste caso é realizada a contagem de quantas ocorrências existem de acordo com o campo ESTOQUE_MIN entre 10 e 100, ele mostrará o número 6.

2.1.10 – CLÁUSULA GROUP BY

A cláusula *GROUP BY* serve para agrupar diversos registros com base em uma ou mais colunas da tabela, vamos observar o exemplo abaixo:

```
select count(*), unid_medida
from tbl_produto
group by unid_medida
```

Neste caso serão mostrados os registros agrupados pelo campo UNID_MEDIDA, aparecerão CX = 1, GR = 1, KG = 3, LT = 1, PC = 1 e SC = 3.

```
select count(*), month(data_nasc)
from tbl_cliente
where year(data_nasc) > 1930
group by month(data_nasc)
order by 2
```

Neste caso será mostrada a quantidade de ocorrências existentes de acordo o mês do campo DATA_NASC da tabela TBL_CLIENTE, pois estamos contando e agrupando pelo mês, com o ano do mesmo campo maior que 1930.

```
select count(*) as qtde, month(data_nasc) as mes,
year(data_nasc) as ano
from tbl_cliente
where year(data_nasc) > 1930
group by month(data_nasc), year(data_nasc)
order by 2
```

Neste caso será mostrada a quantidade de ocorrências existentes igualmente ao caso anterior, só com a diferença de colocarmos o ano e agrupando o mesmo, isso nos trará mais uma linha no resultado.

2.2 Funções agregadas

Podemos resumir as informações dos dados de uma tabela através de funções agregadas, que são aplicadas sobre as colunas de uma tabela. Devemos passar como argumento de entrada uma coluna que precisamos extrair umas simples informação da mesma, um único resultado, para que assim, não seja preciso analisar informação por informação, com uso da nossa força bruta.

Você sabe como obter um valor resultante de uma simples soma dos valores de uma coluna específica?

Sabe extrair qual é o maior ou menor valor entre muitos em um conjunto?

Sabe quantas linhas uma tabela possui?

Sabe fazer a média entre um conjunto de valores?

Iremos ver 4 diferentes funções agregadas que geram diferentes tipos de resultado, são elas:

- SUM(): computa e retorna o resultado da soma de todos os valores de uma coluna informada;
- AVG(): retorna a média dos valores de uma coluna;
- MIN(): encontra o menor valor dentre aqueles contidos na coluna de entrada;
- MAX(): encontra o maior dentre todos os valores de um conjunto;

2.2.1 Função SUM()

A função soma computa a soma dos valores de uma coluna. Essa função serve para manipularmos dados numéricos. O tipo numérico do dado pode ser *integer e decimal*. O resultado da função *SUM()* será do mesmo tipo da coluna que está sendo utilizada por esta função, vamos observar o exemplo abaixo:

```
select sum(estoque_min) as soma_estoque
from tbl_produto;
```

Neste caso será mostrada a soma do campo ESTOQUE_MIN existente na tabela.

2.2.2 Função AVG()

A função *AVG()* computa a média entre os valores de uma coluna de dados. Assim como a função *SUM()*, esta função deve ser aplicada sobre dados numéricos. Porque esta função direciona a soma dos valores, da coluna informada, para uma coluna temporária e divide o resultado da soma pela quantidade de valores que foram usadas na soma. Esse resultado final pode ser de um tipo diferente do tipo da coluna usada. Por exemplo, se você aplicar AVG em uma coluna onde só contenha inteiros, o resulta será *decimal*, isso porque ocorre uma divisão antes de finalizar o processo desta função, vamos observar o exemplo abaixo:

```
select avg(estoque_max) as media_qtdemax
from tbl_produto;
```

Neste caso será mostrada a média do campo ESTOQUE_MAX existente na tabela.

2.2.3 Função MIN()

A função MIN serve para obtermos o valor mínimo existente em uma determinada coluna, vamos observar o exemplo abaixo:

```
select min(valor) as vlrmin_preco
from tbl_produto;
```

Neste caso será mostrado o mínimo valor do campo VALOR existente na tabela.

2.2.4 Função MAX()

A função MAX serve para obtermos o valor máximo existente em uma determinada coluna, vamos observar o exemplo abaixo:

```
select max(valor) as vlrmax_preco
from tbl_produto;
```

Neste caso será mostrado o máximo valor do campo VALOR existente na tabela.

2.3 Operações matemáticas no *SELECT*

Podemos realizar “contas” no próprio SELECT para agilizar o processo de consulta, vejamos os exemplos:

```
select qtde * i_valor as valor_produto
from tbl_itempedido
where i_cod_produto = 2
  and i_cod_pedido = 1
```

Neste caso será mostrado o resultado da multiplicação dos dados das colunas VALOR e QTDE.

2.4 Joins de tabelas

Quando queremos relacionar mais de uma tabela no *SELECT*, podemos contar com alguns tipos de JOINS que veremos a seguir:

2.4.1 *Inner Join*

É o método de junção mais conhecido e, pois retorna os registros que são comuns às duas tabelas, precisamos fazer a amarração das tabelas com as chaves respectivas, veja os exemplos:

```
select *
from tbl_cliente as c
inner join  tbl_pedido as p
on c.cod_cliente = p.cod_cliente
```

Ou

```
select *
from tbl_cliente as c, tbl_pedido as p
where c.cod_cliente = p.cod_cliente
```

Ambos darão o resultado:

cod_cliente	nome_cliente	cpf	data_nasc	cep	numero	complemento	cod_pedido	data_pedido	data_entrega	cod_cliente
1	Marcos Costa de Sousa	12345678901	1981-02-06	6768100	1525	apto 166C	1	2019-01-01	2019-01-04	1
2	Zoroastro Zoando	01987654321	1989-06-15	6757190	250		2	2019-01-05	2019-01-07	2
3	Idelbrandolâncio Silva	54698721364	1974-09-27	6753001	120		3	2019-01-12	2019-01-15	3
5	Conegunda Prado	54781269501	1950-10-06	6753020	50	apto 166C	4	2019-01-15	2019-01-16	5
7	Iscruência da Silva	12457965823	1974-11-25	6803040	5		5	2019-01-20	2019-01-22	7

2.4.2 *Left Join*

Tem como resultado todos os registros que estão na tabela A (mesmo que não estejam na tabela B) e os registros da tabela B que são comuns à tabela A, precisamos fazer a amarração das tabelas com as chaves respectivas, veja os exemplos:

```
select *
from tbl_cliente as c left join tbl_pedido as p
on c.cod_cliente = p.cod_cliente
```

O resultado será esse:

cod_cliente	nome_cliente	cpf	data_nasc	cep	numero	complemento	cod_pedido	data_pedido	data_entrega	cod_cliente
1	Marcos Costa de Sousa	12345678901	1981-02-06	6768100	1525	apto 166C	1	2019-01-01	2019-01-04	1
2	Zoroastro Zoando	01987654321	1989-06-15	6757190	250		2	2019-01-05	2019-01-07	2
3	Idelbrandolâncio Silva	54698721364	1974-09-27	6753001	120		3	2019-01-12	2019-01-15	3
5	Conegunda Prado	54781269501	1950-10-06	6753020	50	apto 166C	4	2019-01-15	2019-01-16	5
7	Iscruência da Silva	12457965823	1974-11-25	6803040	5		5	2019-01-20	2019-01-22	7
4	Cosmólio Ferreira	41368529687	1966-12-01	6753020	25	apto 255 F	NULL	NULL	NULL	NULL
6	Brogundes Asmônio	41256398745	1940-05-10	6753400	100		NULL	NULL	NULL	NULL
8	Zizafânio Zizundo	54123698562	1964-08-14	6803140	25		NULL	NULL	NULL	NULL
9	Ricuerda Zunda	21698534589	1934-10-14	6803045	123		NULL	NULL	NULL	NULL
10	Aninoado Zinzão	25639856971	1976-12-25	6803070	50		NULL	NULL	NULL	NULL

2.4.3 *Right Join*

Teremos como resultado todos os registros que estão na tabela B (mesmo que não estejam na tabela A) e os registros da tabela A que são comuns à tabela B, precisamos fazer a amarração das tabelas com as chaves respectivas, veja os exemplos:

```
select *
from tbl_pedido as p right join tbl_cliente as c
on p.cod_cliente = c.cod_cliente
```

O resultado será:

cod_pedido	data_pedido	data_entrega	cod_cliente	cod_cliente	nome_cliente	cpf	data_nasc	cep	numero	complemento
1	2019-01-01	2019-01-04	1	1	Marcos Costa de Sousa	12345678901	1981-02-06	6768100	1525	apto 166C
2	2019-01-05	2019-01-07	2	2	Zoroastro Zoando	01987654321	1989-06-15	6757190	250	
3	2019-01-12	2019-01-15	3	3	Idelbrandolâncio Silva	54698721364	1974-09-27	6753001	120	
4	2019-01-15	2019-01-16	5	5	Conegunda Prado	54781269501	1950-10-06	6753020	50	apto 166C
5	2019-01-20	2019-01-22	7	7	Iscruência da Silva	12457965823	1974-11-25	6803040	5	
NULL	NULL	NULL	NULL	4	Cosmólio Ferreira	41368529687	1966-12-01	6753020	25	apto 255 F
NULL	NULL	NULL	NULL	6	Brogundes Asmônio	41256398745	1940-05-10	6753400	100	
NULL	NULL	NULL	NULL	8	Zizafânio Zizundo	54123698562	1964-08-14	6803140	25	
NULL	NULL	NULL	NULL	9	Ricuerda Zunda	21698534589	1934-10-14	6803045	123	
NULL	NULL	NULL	NULL	10	Aninoado Zinzão	25639856971	1976-12-25	6803070	50	

3. VIEWS (Visualizações).

3.1 Introdução.

"**Views**" também conhecidas em português como visualizações nada mais são do que tabelas virtuais que são criadas a partir de seleções de dados, essas tabelas virtuais reúnem apenas os campos de uma ou mais tabelas físicas que são listados em uma instrução "**SELECT**" associada a "**VIEW**", diferente das tabelas físicas "reais" que são armazenadas em disco e são permanentes, ou seja se o servidor de dados for reinicializado tanto as tabelas quanto o seus dados permanecerão em disco as "**VIEWS**" são, como já foi dito virtuais, logo seus dados permanecem armazenados na memória principal do servidor de dados, isso traz

algumas vantagens a mais óbvia é o acesso extremamente rápido a esses dados, logo podemos afirmar que que as "**VIEWS**" são um recurso altamente recomendado em casos onde essas seleções de dados são muito requisitadas o caso mais comum de aplicação é em relatórios, e sendo consideradas como "tabelas" tudo que vimos em DQL pode ser usado em VIEWS.

Abaixo podemos ver a sintaxe de como criar uma "**view**":

CREATE VIEW <NOME_DA_VIEW> **AS** <CONSULTA>;

Para praticarmos um pouco vamos criar três "**views**" em nossa base de dados "**VENDAS**" é necessário que a massa de dados que pedi no início do semestre esteja feita. Uma vez feito isso vamos criar três "views", observe as imagens abaixo:

A primeira "**view**" lista clientes e pedidos:

```
#1ª VIEW - RELAÇÃO DE CLIENTE COM PEDIDO
create view vw_cliped as
   select c.cod_cliente as codigo, c.nome_cliente as nome, p.cod_pedido as pedido,
          p.data_pedido as data_requisicao
   from tbl_cliente c, tbl_pedido p
   where c.cod_cliente = p.cod_cliente;
```

A segunda "**view**" lista clientes, pedidos e itens dos pedidos:

```
#2ª VIEW - RELACIONA CLIENTES, PEDIDOS E ITEM DOS PEDIDOS
create view vw_clipedprod as
   select c.cod_cliente as codigo, c.nome_cliente as nome, p.cod_pedido as pedido,
          p.data_pedido as data_requisicao,
          i.i_cod_produto as produto, i.qtde
   from tbl_cliente c, tbl_pedido p, tbl_itempedido i
   where c.cod_cliente = p.cod_cliente
     and i.i_cod_pedido = p.cod_pedido;
```

A terceira "**view**" lista clientes, pedidos, itens dos pedidos e descrição dos produtos:

```
#3ª VIEW - RELACIONA CLIENTES, PEDIDOS, ITENS DOS PEDIDOS E PRODUTOS
create view vw_prodtudototal as
   select c.cod_cliente as codigo, c.nome_cliente as nome, p.cod_pedido as pedido,
          p.data_pedido as data_requisicao,
          i.i_cod_produto as produto, pr.nome_produto as descricao, i.qtde,
          pr.valor, i.qtde * pr.valor as totalcomprado
   from tbl_cliente c, tbl_pedido p, tbl_itempedido i, tbl_produto pr
   where c.cod_cliente = p.cod_cliente
     and i.i_cod_pedido = p.cod_pedido
     and i.i_cod_produto = pr.cod_produto;
```

Para visualização de "VIEWS" usamos o comando SELECT convencional, e para apagar uma "VIEW" usamos o comando DROP VIEW.

3.2 VIEWS com posição consolidada

Uma VIEW consolidada nada mais é que uma consulta que agrupa informações de vários registros de uma só vez, bastante utilizada para relatórios.

A view irá mostrar os produtos dos pedidos já calculados (quantidade X valor unitário), vejam o seguinte exemplo para explicar:

```
create view vw_consolidavenda (cod_pedido, cod_produto, valor) as
   select p.cod_pedido,
          i.i_cod_produto, i.qtde * pr.valor
   from tbl_pedido p, tbl_itempedido i, tbl_produto pr
   where i.i_cod_pedido = p.cod_pedido
     and i.i_cod_produto = pr.cod_produto;
```

Na sequencia se fizermos um *SELECT* nesta view teremos:

COD_PEDIDO	COD_PRODUTO	VALOR
11	1	22.50
11	2	9.60
11	3	2.50
12	4	17.50
13	5	18.00

Outro exemplo seria agrupar o total dos pedidos, vejam:

```
create view vw_vendatotal (cod_pedido, total_valor) as
   select p.cod_pedido, sum(i.qtde * pr.valor)
   from tbl_pedido p, tbl_itempedido i, tbl_produto pr
   where i.i_cod_pedido = p.cod_pedido
     and i.i_cod_produto = pr.cod_produto
   group by p.cod_pedido;
```

Foram agrupados os pedidos com seus respectivos totais.

Exercícios para fixação:

De acordo com o DER já confeccionado para nossa aula, criem as seguintes views:

a) Que enxerguem os dados do cliente (código e nome) e pedidos (número do pedido, data do pedido e data de entrega), onde a data do pedido seja superior a 30/01/2014;
b) Que enxerguem os dados do cliente (código do cliente e nome), dados do pedido (código do pedido, data do pedido e data da entrega), os dados do item do pedido (quantidade e código do produto), onde a quantidade destes produtos sejam maiores de 25;
c) Que enxerguem os dados do pedido (código do pedido, código do cliente), os dados do item do pedido (quantidade, código do produto e descrição do produto);
d) Que enxerguem os produtos reajustados em 11,2 %, onde deverá ser mostrado o código e a descrição do produto, o valor atual e o valor reajustado.

Esta tarefa deverá ser realizada para o dia 29/08/2022, verificarei as soluções em aula, atividade em duplas.

4. Triggers (Gatilhos)

4.1 Introdução

Uma "**trigger**" também conhecida em português como "**gatilho**" é uma sub-rotina alocada dentro de uma base de dados assim como os procedimentos armazenados (**stored procedure**), porem diferentes desses que precisam ser chamados para serem executados uma "**trigger**" é executada automaticamente (**disparada**) perante uma ação sofrida em uma tabela especifica dentro da base de dados onde a "**trigger**" foi criada, esse disparo pode ocorrer antes ou depois da ação em questão, logo podemos afirmar que "**triggers"** são criadas em um determinado banco de dados ficando associadas a uma tabela especifica e que sua execução depende de uma ação especifica na tabela em questão podendo a execução da "**trigger**" ser antes ou depois da ação que tabela sofre.

A sintaxe de um "**trigger**" pode ser observada abaixo:

DELIMITER [define um caractere delimitador]
CREATE TRIGGER <nome> <momento> <evento> ON <tabela>
FOR EACH ROW
BEGIN
<ação a ser executada pela trigger>
END

Para tornar o entendimento mais fácil vamos criar uma "**trigger**" na base de dados "**BD_VENDAS**", a ideia da mesma é que sempre que um registro na tabela "**TBL_CLIENTE**" for inserido será gravado na tabela chamada de "**TBL_LOG**" o usuário logado no banco de dados, assim como a data e a hora em que a inclusão ocorreu para que esses dados possam ser usados posteriormente para uma possível auditoria.

Para tanto vamos antes de qualquer coisa criar a tabela "**TBL_LOG**" que vai armazenar esses dados de auditoria a imagem abaixo traz o código de criação da tabela:

```
create table tbl_log(
    id_log int not null auto_increment primary key,
    usuario varchar(50) not null,
    dt_log date not null,
    hora time not null
);
```

Agora vamos cria a "**trigger**", essa por sua vez deve saber o momento em que deve ser executada sendo esses antes (**BEFORE**) ou depois (**AFTER**) de uma ação, além disso, devemos também definir em qual tabela a "**trigger**" vai atuar nesse caso, a tabela "**TBL_CLIENTE**" e, seguida devemos definir qual ação será responsável pela execução da "**trigger**" no caso uma ação de "**DELETE**", porem vale ressaltar aqui que poderia ser qualquer outra ação que a tabela possa vir a sofrer, observe a imagem abaixo:

```
delimiter $
create trigger trg_log before delete
on tbl_cliente
for each row
begin
    insert into tbl_log
           (usuario, dt_log, hora)
    values (user(), curdate(), curtime());

end $
```

Exibindo triggers.

Assim como os procedimentos armazenados também podemos listar as "**triggers**" de nossa base de dados, observe as imagens abaixo:

```
show triggers from bd_vendas;
```

Lista todas as "**triggers**" de uma base de dados especifica, nesse caso a base de dados "**BD_VENDAS**" e novamente o usuário que está operando deve ter esse direito.

Excluindo triggers.

Assim como os procedimentos armazenados nos podemos também excluir "**triggers**" que tenham sido criadas, observe a imagem abaixo:

```
drop triggers trg_log;
```

Exercícios para fixação:

a) Modifique a tabela tbl_log acrescentando um campo onde armazene o tipo de operação realizada, sendo: “INSERÇÃO”, “ATUALIZAÇÃO” ou “EXCLUSÃO” e outro campo que armazene a tabela que está sendo realizadas as ações.
b) De acordo com o exercício A crie uma trigger que ao atualizar e antes de qualquer ação na tabela de Pedidos;
c) De acordo com o exercício A crie uma trigger que ao excluir e antes de qualquer ação na tabela de Produtos;
d) De acordo com o exercício A crie uma trigger que ao inserir e depois de qualquer ação na tabela de Clientes.

Essa atividade deverá ser desenvolvida em duplas e apresentada até a próxima aula.

5. Stored Procedures (Procedimentos armazenados).

5.1 Introdução.

Procedimentos armazenados é um conjunto de comandos SQL que podem ser armazenados dentro de uma base de dados e através desses podemos realizar praticamente qualquer tarefa dentro de um SGBDR, geralmente usamos procedimentos armazenados para tarefas como inserção, seleção, alteração e exclusão de dados.

Adotar procedimentos armazenados na manipulação de bancos de dados aumenta o desempenho de uma aplicação, pois os procedimentos ficam armazenados dentro da base de dados e são compilados no momento de sua criação e isso reduz a carga de processamento e consumo de memória principal no servidor de dados.

Além de aumentar o desempenho os procedimentos armazenados também aumentam a segurança, pois toda a lógica da base de dados fica dentro dessa, observe que isso também permite um acesso transparente ao programador na hora de fazer inserções, consultas, alterações e exclusões na base, pois para realizar tais tarefas, basta apenas chamar o procedimento armazenado responsável pela tarefa desejada, isso também permite ganho de tempo na migração de plataformas de desenvolvimento ou na criação de novas aplicações que acessam a mesma base de dados sem que haja necessidade de refazer “strings” SQL.
Por último o fato das “strings” SQL estarem no lado do servidor de dados diminui muito o trafego de dados no momento da execução dessas, pois é passado apenas o nome do procedimento ao invés de um longo comando SQL.

Dado todo o quadro descrito acima fica aqui um alerta, procedimentos armazenados não são a “bala mágica” que vai resolver todos os problemas de lógica de negócios e de carga de processamentos em uma aplicação quando criamos procedimentos armazenados em excesso transferindo toda a responsabilidade da lógica do negócio de uma aplicação para base de dados estamos na verdade criando procedimentos armazenados que possuem algoritmos complexos que exigem grande carga de processamento do servidor de dados que geralmente não é concebido para esse tipo de tarefa, logo devemos ponderar o uso de procedimentos armazenados dividindo a carga de responsabilidades do processamento que está ligado à lógica do negócio da aplicação.

Procure pensar da seguinte forma quando estiver tentando definir essas responsabilidades, procedimentos armazenados servem para tornar a manipulação da base de dados transparente para a aplicação e eventualmente essas vão carregar parte da lógica de negócios, mas não são feitas para concentrar a lógica de negócios na base de dados.

5.2 Criando procedimentos armazenados.

Sintaxe:

```
#DELIMITADOR
delimiter $
#DECLARAÇÃO DA SOTRED PROCEDURE.
create procedure nome_da_procedure()
#INICIO DO CORPO DE CODIGO DA STORED PROCEDURE
begin
    #CÓDIGO SQL QUE A PROCEDURE VAI EXECUTAR

end $ #FIM DO CORPO DE CÓDIGO DA STORED PROCUDURE
```

Podemos observar na imagem acima a sintaxe de criação de um procedimento armazenado bastante simples, tudo começa com a declaração de um delimitador na linha dois (2) através do comando "**DELIMITER**" seguido pelo caractere delimitador que nesse caso foi usado o "**$**" poderia ter sido qualquer outro delimitador, essa técnica é interessante para mostrar para a base de dados onde o procedimento armazenado começa e onde termina.

Logo em seguida vem à declaração do procedimento armazenado propriamente dito através dos comando "**CREATE**" (criar) "**PROCEDURE**" (procedimento) e o nome que se deseja dar a ao procedimento em questão, vale lembrar aqui que o ideal é usar um nome que seja relevante com a tarefa que o procedimento realiza no fim dessa linha temos uma abertura e fechamento de parênteses por enquanto não vamos utiliza-los para nada, mas é ali que são informados os parâmetros de entrada e saída de um procedimento, veremos como explorar esse recurso mais adiante.

O "**BEGIN**" e o "**END**" marcam o inicio e o fim do corpo do procedimento que é onde vamos colocar o código SQL para executar a tarefa desejada.

Exemplo prático.

```
delimiter $
create procedure prc_lista_prod()
    begin
        select * from tbl_produto
        where cod_produto in (1,3,5,7);
    end $
delimiter ;
```

Na imagem acima podemos ver o código de um procedimento armazenado responsável por fazer uma seleção sem critério em uma tabela chamada tbl_produto, observe que esse código será executado apenas uma vez na base de dados e após isso vamos chamar o procedimento armazenado através do nome que associamos a ele no momento de sua criação.

Chamando um procedimento armazenado.

```
call prc_lista_prod();
```

Para chamar um procedimento armazenado basta utilizar o comando "call" seguido do nome do procedimento em questão. Assim aparecendo o resultado da procedure.

cod_produto	nome_produto	desc_produto	unid_medida	estoque_atual	estoque_min	estoque_max	valor
1	Arroz	Arroz agulhinha tipo 1	SC	10	2	20	12.50
3	Macarrão	Macarrão Adria espaguete	PC	50	10	80	5.50
5	Vinagre	Vinagre Castelo	GR	30	10	50	7.89
7	Tomate	Tomate vermelho	KG	80	8	160	6.90

Excluindo um procedimento armazenado.

Uma que tenhamos criado um procedimento armazenado é possível excluir tais estruturas para tal podemos lançar mão do domando "**DROP**" seguido do nome do procedimento armazenado, observe a imagem abaixo:

```
drop procedure prc_lista_prod;
```

5.2.1 Procedimentos Armazenados com parâmetros.

Muitas vezes dentro de uma aplicação precisamos manipular dados em uma base de dados onde não sabemos exatamente quais parâmetros **(dados)** o usuário deseja utilizar, para casos como esses devemos criar procedimentos armazenados mais flexíveis que são capazes de receber valores variáveis e a partir desses valores devolver uma massa de dados para o usuário. Imagine por exemplo realizar uma instrução "**DELETE**" sem um valor para a clausula "**WHERE**", perderíamos todos os dados da tabela.

Existem três tipos de declaração de parâmetros para procedimentos armazenados:

IN – Esse é o modo padrão, quando declaramos um parâmetro como "**IN**" seu valor é protegido, ou seja, quando o programa de chamada passa um valor e esse sofrer alteração no decorrer da execução do procedimento o valor original da variável que representa o parâmetro permanece sem sofrer alteração.

OUT – Aqui é o contrario caso a variável que representa o valor do parâmetro sofrer uma alteração com relação ao valor passado pelo programa de chamada essa alteração se reflete no valor original do parâmetro (a variável que continha o valor original) descartando o antigo pelo novo.

INOUT – Essa variação é a fusão das duas anteriores onde um programa de chamada pode passar um valor e o procedimento armazenado pode alterar esse valor e no final será devolvido o valor alterado para o programa de chamada.

Sintaxe de declaração de procedimentos armazenados com parâmetros.

```
delimiter $
create procedure nome_da_procudure(mode nome_parametro tipo_parametro (tamanho_parametro))
#INICIO DO CORPO DE CÓDIGO DA STORED PROCEDURE
begin

    #CÓDIGO SQL QUE A PROCEDURE VAI EXECUTAR

end $ #FIM DO CORPO DE CÓDIGO DA STORED PROCUDURE
delimiter;
```

Observe o código entre os parenteses:

MODE – É o tipo do comportamento que o parâmetro vai utilizar (IN, OUT ou INOUT).

NOME_PARAMETRO – É o nome que a variavel que representa o parametro vai utilizar.

TIPO_PARAMETRO – É o tipo de dado que o parametro vai receber e pode ser qualquer tipo de dado valido no MySQL.

TAMANHO_PARAMETRO – É o tamanho que o tipo de dado vai suportar.

Exemplo prático de procedimento armazenado com parametro IN.

```
delimiter $
create procedure prc_prod_param_in (in nome_prod varchar(50))
    begin
        select * from tbl_produto where nome_produto = nome_prod;
    end $
delimiter ;
```

Depois executamos a chamada da procedure com o parâmetro

```
call prc_prod_param_in('arroz');
```

E aparece o resultado:

cod_produto	nome_produto	desc_produto	unid_medida	estoque_atual	estoque_min	estoque_max	valor
1	Arroz	Arroz agulhinha tipo 1	SC	10	2	20	12.50

Exemplo prático de procedimento armazenado com parametro OUT.

```
delimiter $
create procedure lista_produto_parametro_out (out total decimal(10,2))
    begin
        select sum(valor) into total
        from tbl_produto;
    end $
delimiter ;
```

Depois executamos a chamada da procedure parametrizando uma variável de saída, porém a procedure será compilada no banco e não trará resultado algum.

```
call prc_prod_param_out(@total);
```

O que a procedure faz é devolver o valor associado a variável parâmetro TOTAL.

```
select @total;
```

Trazendo o resultado:

Result Grid

@total
72.28

Exemplo prático de procedimento armazenado com parametro INOUT.

```
delimiter $
create procedure prc_prod_param_inout
                (in codprod int, inout nome_prod char(15),
                 inout valor_prod decimal(10,2))
    begin
        select nome_produto, valor
        from tbl_produto
        where cod_produto = codprod;
    end $
delimiter ;
```

Depois executamos a chamada da procedure passando o parâmetro de entrada, e as variáveis de saída.

```
call prc_prod_param_inout(10,@nomeprod, @valorprod);
```

O resultado sai assim:

nome_produto	valor
Café	11.50

5.3 A estrutura de decisão IF...ELSE.

Assim como em qualquer linguagem de programação dentro da SQL temos também a estrutura de decisão "**IF..ELSE"** que tem como finalidade testar um determinado valor ou condição e tomar uma decisão quanto a qual direção fluxo da execução do código deve seguir.

A sintaxe da estrutura "**IF..ELSE":**

IF <(condição)> THEN
Instruções a serem executadas caso a condição seja verdadeira.
ELSEIF <(condição)> THEN
Instruções a serem executadas caso a condição seja verdadeira.
ELSE
Instruções a serem executadas caso a condição seja falsa.
END IF;

Vamos agora fazer uma procedure para fazer a inserção de dados na TBL_PRODUTO, fazendo consistência no valor do produto, vejam:

```
delimiter $
create procedure prc_ins_prod (in vnomeprod char(100),
                               in vvalor decimal(10,2),
                               out msg varchar(100))
    begin
        declare valor decimal(10,2);
        declare erro bool;

        set erro = true;

        if (vvalor > 0) then
            set valor = vvalor;
        else
            set erro = false;
            set msg = "valor zerado, verifique!";
        end if;

        if (erro) then
            insert into tbl_produto (nome_produto, valor)
            values (vnomeprod, vvalor);
            set msg = "incluido com sucesso!";
        end if;
    end $
delimiter ;
```

Esta procedure verifica o valor do produto e dependendo da situação é dado uma situação de erro ou acerto na mesma. Abaixo executamos a procedure fazendo uma inserção:

```
call prc_ins_prod('ovo',2.55,@msg);
```

E agora verificamos a resposta da execução da procedure:

```
select @msg;
```

Exercícios para fixação:

a) Crie um procedimento armazenado que é passado o código do produto na tabela de produtos e um percentual para calcular o acréscimo ao valor desse mesmo produto, o retorno deverá ser uma mensagem informando se a operação foi feita de forma correta.
b) Crie um procedimento armazenado que grave na tabela de log (Exercício d) da atividade anterior, no campo tipo de operação, informem “INS_TRIGGER” o registro de auditoria;
c) **DESAFIO:** agora remova a TRIGGER que você criou conforme o exercício anterior - d), montando uma nova TRIGGER, só que agora, a mesma deverá chamar a execução do procedimento armazenado criado nessa atividade no item b).

Essa atividade deverá ser entre na aula do dia 26/09/2022.

5.4 Cursor (Cursores)

Cursor é um recurso bastante interessante em bancos de dados, pois permite que seus códigos SQL façam uma varredura de uma tabela ou consulta linha-por-linha, realizando mais de uma operação se for o caso.

Na maioria das vezes, um simples SELECT exibe na tela esta varredura, trazendo todos os registros da consulta em questão. A vantagem de usar um cursor é quando, além da exibição dos dados, queremos realizar algumas operações sobre os registros. Se o volume de

operações for grande, fica muito mais fácil, limpo e prático escrever o código utilizando cursor, do que uma consulta SQL.

Por exemplo: É muito mais vantajoso criar um cursor que faça a análise de cada produto de estoque, conferindo seu histórico, calculando sua previsão de vendas para o próximo mês, capturando o melhor cliente que já o comprou, etc, do que criar um SELECT absurdamente grande que talvez não consiga ainda todas as informações de forma simples.

Abaixo vamos fazer duas PROCEDURES, uma criando uma tabela temporária, e outra efetivamente usando o Cursor, iremos fazer a chamada de uma PROCEDURE dentro da outra, mostrando esse recurso, mas antes vamos verificar o conceito de tabelas temporárias:

- Tabelas temporárias -> Como o próprio nome sugere, são tabelas utilizadas para armazenamento provisório de dados, não são geradas fisicamente e sim em memória, ou seja, se fecharmos o Workbench ou desligarmos nossa máquina, ela deixará de existir, sua utilização é muito válida para armazenamento de dados temporários para processamentos pontuais. A sintaxe é igual a criação de tabelas físicas, só acrescenta a palavra TEMPORARY. Veja:

```
CREATE TEMPORARY TABLE [Nome da Tabela]
(
        [Campos]
)
```

Para “dropar” a tabela temporária, o comando é:

```
DROP TEMPORARY TABLE [Nome da Tabela]
```

Antes de trabalharmos em nossa práxis com cursores, vamos ver estruturas de repetição no MySQL, abaixo temos exemplos de funcionalidades dessas estruturas, caso queiram podem implementá-las a parte, nesse momento iremos captar somente o conceito das mesmas para prosseguirmos nossa aula, vejam:

5.4.1 Estrutura de repetição (WHILE, LOOP, REPEAT)

Assim como nas linguagens de programação, em PROCEDURES também temos estruturas de repetição.

- WHILE

O comando WHILE somente executa as declarações contidas em seu corpo se a condição testada retornar o valor TRUE (verdadeiro).

A sintaxe:

```
[<rótulo>:] WHILE condição DO
        declarações
END WHILE [<rótulo>];
```

Agora para exemplificar nossa abordagem, vamos criar a tabela temporária para enfatizar nosso exemplo, ela terá o objetivo de armazenar os resultados das estruturas de repetição:

```
/* TABELA TEMPORÁRIA */
create temporary table tbl_aux (
      num_exec integer not null,
      seq integer not null,
      descricao varchar(20),
      tipo_repet varchar(10),
      primary key(num_exec, seq)
);
```

Agora vamos montar a procedure com a estrutura WHILE

```
/* ESTRUTURA DE REPETIÇÃO WHILE  */
delimiter $$
create procedure acum_while (in limite int)
begin
    declare contador int default 0;
    declare soma int default 0;
    declare result   varchar(20);
    declare num_exec1 integer;

    select ifnull(max(num_exec) + 1,1) num_exec into num_exec1
    from tbl_aux;

    while contador < limite do
        set contador = contador + 1;
        set soma = soma + contador;
        insert into tbl_aux(num_exec, seq, descricao, tipo_repet)
                    values (num_exec1, contador, soma,'while');
    end while;

    select * from tbl_aux;

end $$
delimiter ;
```

Após isso vamos fazer a chamada da procedure:

```
call acum_while(12);
```

O Resultado será:

Result Grid | Filter Rows: | Export:

NUM_EXEC	SEQ	DESCRICAO	TIPO_REPET
1	1	1	WHILE
1	2	3	WHILE
1	3	6	WHILE
1	4	10	WHILE
1	5	15	WHILE
1	6	21	WHILE
1	7	28	WHILE
1	8	36	WHILE
1	9	45	WHILE
1	10	55	WHILE
1	11	66	WHILE
1	12	78	WHILE

- LOOP

O comando LOOP sofre uma consistência dentro da rotina e a mesma finaliza através do comando LEAVE, conforme abaixo, vejam:

```
/* ESTRUTURA DE REPETIÇÃO LOOP  */
delimiter $$
create procedure acum_loop (in limite int)
begin
    declare contador int default 0;
    declare soma int default 0;
    declare result   varchar(20);
    declare num_exec1 integer;

    select ifnull(max(num_exec) + 1,1) num_exec into num_exec1
    from tbl_aux;

    loop_acum: loop

        set contador = contador + 1;
        set soma = soma + contador;

        if contador >= limite then
            leave loop_acum;
        else
            insert into tbl_aux(num_exec, seq, descricao, tipo_repet)
                    values (num_exec1, contador, soma,'loop');
        end if;

    end loop loop_acum;
    select * from tbl_aux;

end $$
delimiter ;
```

Após isso vamos fazer a chamada da procedure:

```
call acum_loop(5);
```

O resultado será:

Result Grid | Filter Rows: | Expo

NUM_EXEC	SEQ	DESCRICAO	TIPO_REPET
2	1	1	LOOP
2	2	3	LOOP
2	3	6	LOOP
2	4	10	LOOP

- REPEAT

O comando REPEAT executa as declarações contidas em seu corpo e a condição é testada somente no final da estrutura, isso quer dizer que a mesma executará pelo menos 1 vez. Vejam:

```
/* ESTRUTURA DE REPETIÇÃO REPEAT  */
delimiter $$
create procedure acum_repeat (in limite int)
begin
    declare contador int default 0;
    declare soma int default 0;
    declare result   varchar(20);
    declare num_exec1 integer;

    select ifnull(max(num_exec) + 1,1) num_exec into num_exec1
    from tbl_aux;

    repeat

        set contador = contador + 1;
        set soma = soma + contador;
        insert into tbl_aux(num_exec, seq, descricao, tipo_repet)
                    values (num_exec1, contador, soma,'repeat');

        until contador >= limite
    end repeat;
    select * from tbl_aux;

end $$
delimiter ;
```

Após isso vamos fazer a chamada da procedure:

```
call acum_repeat(4);
```

O resultado será:

Result Grid | Filter Rows:

NUM_EXEC	SEQ	DESCRICAO	TIPO_REPET
3	1	1	REPEAT
3	2	3	REPEAT
3	3	6	REPEAT
3	4	10	REPEAT

Após esses exemplos e explicações das estruturas de repetição, iremos dar continuidade com nossa práxis fazendo uma procedure de ajuste de estoque, onde faremos a chamada de outra procedure para criação de tabela temporária e assim seguiremos com novos conceitos.

```
#PROCEDURE PARA CRIAÇÃO DE TABELAS TEMPORÁRIAS
delimiter @@
create procedure cria_tabela()
    begin
        create temporary table if not exists tbl_auxprod (
            aux_prod   integer not null,
            aux_desc   varchar(20),
            qtde_ajust varchar(10),
            data_hora   datetime default now(),
            primary key(aux_prod, data_hora)
        );

        delete from tbl_auxprod;

    end @@
delimiter;
```

Essa primeira PROCEDURE terá a função de criar tabela temporária e limpar a mesma, a limpeza se faz necessária porque se a mesma já existir, não será criada novamente, nesse caso só deverá ser limpa.

Na procedure a seguir temos as 4 instruções básicas:

1. Declare o cursor: Nomeando-o e definindo a estrutura da consulta a ser executada dentro dele;

2. Abra o cursor: A instrução **OPEN** executa a consulta e vincula as variáveis que estiverem referenciadas. As linhas identificadas pela consulta são chamadas de conjunto ativo e estão agora disponíveis para extração;

3. Extrair dados do cursor: Após cada extração você testa o cursor para qualquer linha existente. Se não existirem mais linhas para ser processada, você precisará fechar o cursor;

4. Feche o cursor: A instrução **CLOSE** libera o conjunto ativo de linhas;

Você utiliza as instruções **OPEN**, **FETCH** e **CLOSE** para controlar um cursor. A instrução OPEN executa a consulta associada ao cursor, identifica o conjunto ativo e posiciona o cursor, um indicador antes da primeira linha. A instrução FETCH recupera a linha atual e avança o cursor para a próxima linha. Após o processamento da última linha, a instrução CLOSE desativa o cursor. Veja a PROCEDURE a seguir:

```
#PROCEDURE PARA AJUSTE DE ESTOQUE
delimiter $$
create procedure ajusta_estoque (in qtde int, out msg char(100))
    begin
        -- definição de variáveis utilizadas na procedure
        declare p_linha   int default 0;
        declare p_codigo  int default 0;
        declare p_descri  varchar(100);
        declare p_estoque int default 0;
        declare p_status  int default 0;

        -- definição do cursor
        declare meucursor cursor for
                        select cod_produto, nome_produto, estoque_atual from tbl_produto;

        -- definição da variável de controle de looping do cursor
        declare continue handler for not found set p_linha = 1;

        -- abertura do cursor
        open meucursor;

        -- chamada da procedure de criação da tabela temporária
        call cria_tabela();

        -- looping de execução do cursor
        meuloop: loop
            fetch meucursor into p_codigo, p_descri, p_estoque;

            -- controle de existir mais registros na tabela
            if p_linha = 1 then
                select count(*) into p_status from tbl_auxprod;
                if p_status > 0 then

                    -- seleciono a tabela temporária
                    select * from tbl_auxprod;
                    leave meuloop;
                else

                    -- a procedure rodou mas sem nenhum processamento
                    set msg = "nada processado!";
                    select msg;
                    leave meuloop;
                end if;
             elseif p_estoque = qtde then

                  -- atualizo o estoque
                  update tbl_produto set estoque_atual = qtde + 2
                  where cod_produto = p_codigo;

                  -- insere os dados na tabela temporária
                  insert into tbl_auxprod (aux_prod,aux_desc, qtde_ajust)
                  values (p_codigo, p_descri, p_estoque + 2);
             end if;

         end loop meuloop;

         -- fechamento do cursor
         close meucursor;
    end $$
delimiter ;
```