BANCO DE DADOS

AULA 6

Prof. Ricardo Sonaglio Albano
Profª. Silvie Guedes Albano

CONVERSA INICIAL

Nesta etapa, iremos contemplar os temas mais avançados de um Banco de Dados, abordando a utilização de estruturas auxiliares, como os índices e as *views*, também conhecidas como *tabelas virtuais*.

Também estudaremos sobre as transações que atuam como um conjunto de operações que devem ser executadas como um bloco único dentro de um Banco de Dados.

Concluiremos este material tratando sobre a implementação de códigos no Banco de Dados, mergulhando no funcionamento das *triggers*, *stored procedures*, funções e cursores, os quais são recursos muito importantes que visam simplificar os procedimentos e as tarefas repetitivas que devem ser realizados no trabalho diário de manipulação e administração de um Sistema Gerenciador de Banco de Dados (SGBD). Bons estudos e sucesso!

TEMA 1 – ÍNDICE E VISÃO (*INDEX* E *VIEW*)

Antes de aprendermos o que é um índice, é necessário conhecermos o funcionamento de uma consulta tradicional no Banco de Dados.

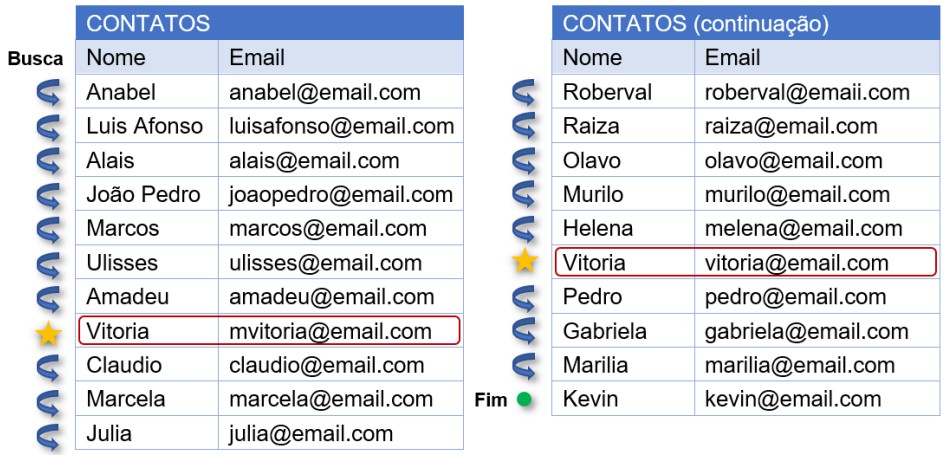
Ao executarmos um *select* com a declaração de um filtro (*where*), o Sistema Gerenciador de Banco de Dados (SGBD) realiza uma ação denominada *table scan* ou *full scan*, a qual faz uma varredura na tabela, ou seja, o SGBD percorre toda a tabela procurando as linhas que atendem ao filtro definido na consulta. Toda linha analisada que não atenda ao filtro será ignorada.

Vale ressaltar que também existe a tabela *heap*, que não apresenta nenhum índice. Contudo, na prática, sua existência é bastante rara, inclusive porque a própria *Structured Query Language* (SQL) cria índices implícitos automaticamente para colunas PK (chave primária) e FK (chave estrangeira).

Exemplo: em uma tabela que contém o contato de várias pessoas (nome e e-mail), vamos consultar a(s) pessoa(s) com o nome Vitória.

```
create table pessoa(
   nome varchar(50),
   email varchar(50));
select nome, email from pessoa where nome = 'Vitoria';
```

Figura 1 – Consulta na tabela Pessoa sem índice



Fonte: Albano; Albano, 2022.

Perceba que, na criação da tabela Pessoa, não houve a declaração de chave (PK ou FK) ou índice, o que caracteriza uma tabela *heap*.

Na busca pelo nome Vitória, o SGBD percorre a tabela e compara cada nome armazenado com o filtro realizado (`where nome = 'Vitoria'`). Se a linha corresponder ao filtro, será adicionada ao conjunto de dados que será retornado, caso contrário, será ignorada. Porém, mesmo que seja encontrada uma linha correspondente, o SGBD seguirá executando a varredura até o final da tabela, uma vez que pode haver mais linhas que atendam o filtro.

Também é importante salientar que esse exemplo é executado em uma tabela pequena e, dessa forma, o impacto desse processo não é muito grande. Contudo, imagine essa consulta em uma tabela que contenha milhares ou milhões de linhas. Qual seria o impacto de uma consulta que precisa ler toda a tabela procurando as linhas que correspondem ao filtro? Certamente, isso gera um alto custo computacional.

Para minimizarmos o impacto usamos os índices, que agilizam a localização e a recuperação das linhas que correspondem a cláusula *where*, ou seja, aumentam a velocidade do tempo de resposta de uma consulta.

Ao longo do nosso estudo, em diversos momentos, temos ressaltado a importância do uso de boas práticas. Agora, vamos mencionar mais alguns pontos relevantes, os quais são:

- Selecione apenas as colunas necessárias, pois isso reduzirá a quantidade de colunas que devem ser retornadas. Na maioria dos casos, não é necessário o uso do asterisco (*), que é equivalente a todos, em um *select*;

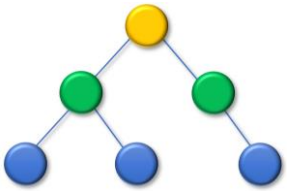
- Na cláusula *where*, procure usar filtros nesta ordem: números, datas e horas, textos e, por último, textos extensos;
- Quanto mais preciso e menos complexo for a declaração de um filtro, mais simples será a execução da consulta;
- Quando possível, use a comparação por igualdade e não por faixa de valores, reduzindo, assim, o número de comparações que o SGBD precisará realizar para filtrar os dados;
- Utilize índices, principalmente em campos numéricos. Tal recurso auxilia o SGBD a localizar os dados mais facilmente.

1.1 Índices

São estruturas que auxiliam na organização e na consulta de uma tabela, as quais têm por objetivo facilitar o acesso aos dados, otimizando o desempenho das consultas. Com sua utilização, inclusive, ocorre uma diminuição no número de operações de entrada e saída (I/O), o que acaba reduzindo o uso dos recursos do servidor.

Nos Sistemas Gerenciadores de Banco de Dados (SGBDs) o tipo mais usado de índice é denominado *Balance Tree* (*B-Tree*), sendo representado por estrutura em árvore balanceada, ou seja, é uma árvore de pesquisa binária em que um nó não pode possuir mais que dois filhos e, além disso, por ser uma árvore balanceada, significa que um lado da árvore não pode ter mais nós que o outro lado.

Figura 2 – Representação do índice *B-Tree*



Fonte: Albano; Albano, 2022.

O índice *B-Tree* pode ser classificado em *clustered* e *non-clustered*, sendo:

1. *B-Tree clustered* – A tabela de dados é organizada pela coluna indexada. Na *Structured Query Language* (SQL), ao criarmos uma PK (chave primária), automaticamente será um índice *clustered* e, sendo assim, as

linhas da tabela são organizadas pela chave primária e não pela ordem de inclusão. Porém, apenas um índice *clustered* pode ser declarado em cada tabela e, apesar de ser eficiente para agilizar as operações de localização de registros, possui um pequeno "*overhead*" (tempo de espera) nas operações de atualização, inserção e exclusão, isso porque a tabela tem que ser reorganizada pela ordem do índice.

2. *B-Tree non-clustered* – Diferente do *clustered*, os índices *non-clustered* contém apenas a indicação (ponteiros) para as páginas de dados onde estão as informações, ou seja, não há uma organização por ordem do índice. Além disso, toda coluna FK (chave estrangeira) é um índice *B-Tree non-clustered*.

Quando usar índices?

- Em tabelas que apresentem um grande volume de dados;
- Em colunas que armazenem uma grande variação de valores (faixa de valores), como por exemplo, uma coluna que contém salários ou data de nascimento;
- Quando uma ou mais colunas forem usadas frequentemente em filtros *where* (PK e FK) ou em *joins* (união de tabelas);
- Em colunas que são usadas com operadores de comparação e/ou com as cláusulas *between*, [*not*] *exists*, [*not*] *in*, *group by* e *order by*.

Quando não é aconselhável usar índices?

- Em tabelas com volume de dados reduzido;
- Em colunas pouco referenciadas em filtros nas consultas.

Criando um índice: é possível criar um índice na própria declaração de uma tabela, mas, geralmente, a necessidade de criação de um índice surge na medida que as tabelas crescem e, dessa forma, pode ser incluído posteriormente na tabela.

1. Criação da tabela – Usa-se a palavra reservada *index*, seguida do nome da coluna que será indexada.

Sintaxe:
```
create table nomeTabela(
        nomeColuna1 tipo_dado,
        ...,
        nomeColunaN tipo_dado,
```

```
                    index(nomeColuna));
```

Exemplo:
```
create table cliente(
                    codigo integer,
                    nome varchar(100),
                    email varchar(100),
                    index(codigo));
```

2. Alteração da tabela – Na definição de um índice em uma tabela existente usamos o comando *create index*.

Sintaxe:
```
create index nomeIndice
                    on tabela(colunas);
```

Exemplo:
```
create index idxContatoNome
                    on Contatos(nome);
```

Para demonstrar o uso do índice, vamos realizar algumas consultas com e sem a utilização de índices.

Exemplo: Projetar os funcionários que possuem salários superiores a R$ 5.000,00.

```
select * from funcionario where salarioFunc > 5000;
```

Figura 3 – Tabela Funcionário sem índice

matricula	nomeFunc	generoFunc	NascFunc	salarioFunc	departamento	cargo	emailFunc	cidadeId
1	Ana Rosa Silva	F	1996-12-31	8500.00	1	1	ana.rosa@email.com	1
2	Tales Heitor Souza	M	2000-10-01	7689.00	1	2	tales.heitor@email.com	1
3	Bia Meireles	F	2002-03-14	9450.00	1	2	bia.meireles@email.com	2
4	Pedro Filho	M	1998-05-22	12340.00	3	3	pedro.filho@email.com	2
5	Camila Fialho	F	1989-03-15	10450.00	2	3	camila.fialho@email.com	4
6	Ulisses Frota	M	1997-06-30	12340.00	1	4	ulisses.frota@email.com	7
7	Leonardo Timbo	NULL	2001-07-02	7850.00	2	3	leonardo.timbo@email.com	2
8	Lucas Goes	M	2002-03-02	8834.00	3	4	lucas.goes@email.com	5
9	Sofia Lima	NULL	1999-12-23	9578.00	4	4	sofia.lima@email.com	5
10	Nicolas figueira	M	1997-06-01	12340.00	3	2	nicolas.figueira@email.com	3
11	Helena Arcanjo	F	1998-11-20	6320.00	2	2	helena.arcanjo@email.com	7
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Fonte: Albano; Albano, 2022.

Figura 4 – Tempo de execução sem índice

	Action	Response	Duration / Fetch Time
1	select * from funcionario where salarioFunc > 5000 LIMIT 0, 1000	11 row(s) returned	0.0022 sec / 0.00002...

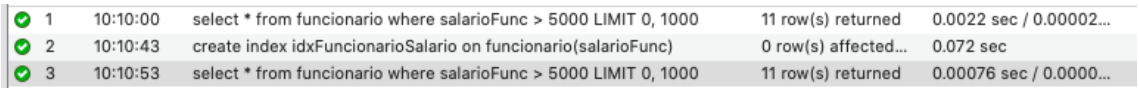
Fonte: Albano; Albano, 2022.

Perceba que o tempo gasto para a execução da consulta resultou em 0.0022 segundos.

A seguir iremos criar um índice para a coluna ‘salarioFunc’, realizando a mesma consulta feita anteriormente e verificando se houve alteração no tempo de execução.

```
create index idxFuncionarioSalario
   on funcionario(salarioFunc);
select * from funcionario where salarioFunc > 5000;
```

Figura 5 – Comparação de tempo de execução com e sem o uso de índice



1	10:10:00	select * from funcionario where salarioFunc > 5000 LIMIT 0, 1000	11 row(s) returned	0.0022 sec / 0.00002...
2	10:10:43	create index idxFuncionarioSalario on funcionario(salarioFunc)	0 row(s) affected...	0.072 sec
3	10:10:53	select * from funcionario where salarioFunc > 5000 LIMIT 0, 1000	11 row(s) returned	0.00076 sec / 0.0000...

Fonte: Albano; Albano, 2022.

Analisando o tempo de execução apresentado na figura, verificamos uma redução significativa (0.00076 segundos) em comparação a execução anterior, o que demonstra a melhoria no desempenho da consulta.

Na administração de um Banco de Dados, surge a necessidade de sabermos quais índices cada tabela possui e, para isso, usamos o comando *show*. Você se lembra dele?

Sintaxe: **`show index from nomeTabela;`**

Exemplo: **`show index from funcionario;`**

Figura 6 – Índices da tabela Funcionário

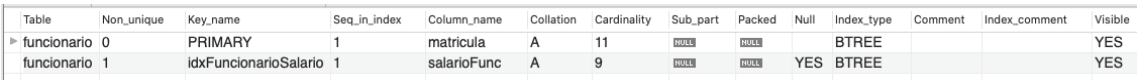


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
funcionario	0	PRIMARY	1	matricula	A	11	NULL	NULL		BTREE			YES
funcionario	1	idxFuncionarioSalario	1	salarioFunc	A	9	NULL	NULL	YES	BTREE			YES

Fonte: Albano; Albano, 2022.

1.2 View

Também conhecida como tabela virtual, uma *view* é um objeto que retorna um conjunto de dados de uma consulta armazenada em um Banco de Dados. Como possui colunas de tabelas “reais”, ela pode receber comandos de declarações *join*, *where* e funções, como se fosse uma tabela normal. Além

disso, pode ser usada em comandos *insert*, *update* e *delete* para atualização de dados. Onde aplicar?

- Para simplificar o acesso a várias tabelas relacionadas;
- Na segurança dos dados na tabela, limitando o acesso a determinadas colunas de uma tabela que poderão ser acessadas;
- Para isolar a estrutura da tabela de uma aplicação;
- Em um conjunto de tabelas que podem ser unidas a outros conjuntos de tabelas com a utilização de *join* ou *union*.

Sintaxe:
```
create [or replace] [algorithm = algorithm_type]
        view nomeView [(listaColuna)] as
        select [with [cascaded | local] check option];
```

Onde:

nomeView	Nome que damos ao objeto *view* que criarmos.
listaColuna	Recurso para sobrescrever os nomes das colunas recuperadas do *select*.
or replace	Indica a substituição da *view* existente. O uso do *alter table* gera o mesmo efeito.
algorithm	Define o algoritmo interno utilizado para processar a *view*.
with check option	Para realizar alterações nos dados de uma *view*, a cláusula *where* (da *view*) deve ser considerada (respeitada).

Alguns comandos úteis:

1. Excluir uma *view* através do comando *drop view*.

   Sintaxe:
   ```
   drop view nome_da_view;
   ```
2. Consultar as *views* existentes.

   Sintaxe:
   ```
   show full tables in nome_banco
        where table_type like 'view';
   select table_schema, table_name
        from information_schema.tables
        where table_type like 'view';
   ```

Exemplo: uma empresa necessita disponibilizar as informações (nome, gênero, salário e e-mail) dos funcionários para o Departamento de Marketing. Ocorre que essas informações, principalmente o salário, são restritas e, dessa

forma, seria mais seguro e apropriado criarmos uma *view* que conterá apenas as informações necessárias.

```
create view mostraFuncionario as
     select nomeFunc as 'Nome',
          generoFunc as 'Genero',
          salarioFunc as 'Salario',
          emailFunc as 'Email' from funcionario;
```

3. Usar a *view*.

```
select * from mostraFuncionario order by Nome;
```

Figura 7 – Select da view

Nome	Genero	Salario	Email
Ana Rosa Silva	F	8500.00	ana.rosa@email.com
Bia Meireles	F	9450.00	bia.meireles@email.com
Camila Fialho	F	10450.00	camila.fialho@email.com
Helena Arcanjo	F	6320.00	helena.arcanjo@email.com
Leonardo Timbo	NULL	7850.00	leonardo.timbo@email.com
Lucas Goes	M	8834.00	lucas.goes@email.com
Nicolas figueira	M	12340.00	nicolas.figueira@email.com
Pedro Filho	M	12340.00	pedro.filho@email.com
Sofia Lima	NULL	9578.00	sofia.lima@email.com
Tales Heitor Souza	M	7689.00	tales.heitor@email.com
Ulisses Frota	M	12340.00	ulisses.frota@email.com

Fonte: Albano; Albano, 2022.

1.3 Transações

Caracteriza-se por ser um conjunto de operações em sequência, as quais são executadas como um bloco único e indivisível, com início e fim (término), sendo o término definido pela presença de um dos seguintes comandos:

- *Commit* – Confirma a execução em definitivo do bloco de instruções executado, finalizando a transação com sucesso;
- *Rollback* – Descarta todas as instruções executadas, indicando que ocorreu um erro.

As transações baseiam-se no princípio do “tudo ou nada”, que, de forma geral, define que deve existir a garantia da integridade de todos os dados ou nenhum dado novo será adicionado ao Banco de Dados. Isso quer dizer que todas as operações declaradas na transação serão finalizadas com sucesso ou, então, descartadas. **Tipos de transações:**

- Implícitas – Ocorrem automaticamente quando enviamos os comandos *insert*, *update* ou *delete*;
- Explícitas – Seguem o conceito formal de transações, onde se deve indicar o início e o término.

Geralmente, nos Sistemas Gerenciadores de Banco de Dados (SGBDs), os comandos *commit* e *rollback* são definidos como padrão e estão implícitos. Porém, é possível desabilitá-los tornando-os explícitos, sendo, assim, necessária a declaração e a execução para confirmar ou descartar definitivamente uma transação.

Por definição, uma transação deve apresentar as seguintes propriedades (ACID):

- **A**tomicidade:
  - Bloco único (indivisível), sempre apresentando início e fim;
  - Princípio do “tudo ou nada”, evitando que falhas possam deixar a base de dados inconsistente.
- **C**onsistência:
  - Ao término de uma transaçãom todos os dados devem estar em um estado consistente em um Banco de Dados relacional, ou seja, todas as regras e a integridade referencial devem ter sido respeitadas.
- **I**solamento:
  - Garante que uma determinada transação não usará dados em um estado intermediário, ou seja, dados oriundos de instruções que tenham sido executadas, mas não confirmadas;
  - Mesmo que existam transações executadas concorrentemente, cada execução isolada apresentará o mesmo resultado.
- **D**urabilidade:
  - Uma vez encerrada a transação, as alterações por ela efetuadas são definitivas;
  - Significa que os resultados de uma transação, caso ela seja concluída com sucesso, devem ser persistentes, mesmo se depois houver uma falha no sistema.

Declarando uma transação:

Sintaxe:
```
start transaction
    [comandos]
```

```
[commit / rollback]
set autocommit = {0 ou 1} ou {OFF ou ON};
```

Onde:

start transaction	Inicia o bloco de transação.
commit	Confirma a transação que está sendo executada.
rollback	Desfaz ou reverte a transação atual.
set autocommit	Habilita ou desabilita o modo automático da transação. O valor zero ou OFF equivale a desabilitar.

Em um Banco de Dados, são executadas diversas transações ao mesmo tempo (execuções simultâneas), sendo esse processo denominado de concorrência. É importante ressaltar que processamentos simultâneos podem provocar alguns problemas, tais como:

- Leitura suja (*dirty read*) – Ocorre em uma transação que acessa dados não confirmados, ou seja, que estão sendo atualizados por outra transação, o que pode gerar dados incorretos;
- Leitura não repetida (*nonrepeatable read*) – A transação acessa o mesmo dado várias vezes e obtém valores diferentes em cada acesso;
- Leitura fantasma (*phantom read*) – Ao ler um conjunto de linhas da tabela, a transação obtém uma quantidade diferente de linhas em cada leitura;
- Atualização perdida (*lost update*) – Duas ou mais transações consultam a mesma linha e a atualizam com base no valor original.

Para evitar os problemas de concorrência, pode ser definido um nível de isolamento para as transações. Tal mecanismo de isolamento é conhecido como *lock* (bloqueio), onde é definido os níveis de isolamento que podem ser classificados em:

- *Read uncommited* – Permite a leitura de dados não atualizados;
- *Read commited* – Permite a leitura de dados atualizados. Os registros manipulados por comandos *insert*, *update* e *delete* permanecem bloqueados para outras sessões até que a transação seja concluída;
- *Repeatable read* – Não é possível, por uma transação, ler dados que não foram confirmados por outras transações e nenhuma outra transação pode modificar dados que foram lidos pela transação atual até que a transação atual seja concluída;

- *Serializable* – Nenhuma transação pode modificar dados lidos pela transação atual até sua conclusão, inclusive, também não é permitido que outras transações insiram novas linhas nesses mesmos dados, enquanto a transação atual não for concluída.

É importante salientar que quanto maior o nível de isolamento menor o número de transações que serão executadas ao mesmo tempo, o que impacta diretamente no desempenho do Banco de Dados.

Declarando o nível de isolamento da transação: a definição do nível de isolamento deverá ser declarada antes do início da transação.

Sintaxe:
```
set transaction isolation level
      {read uncommited, read commited,
       repeatable read, serializable};
```

Exemplo:
```
set transaction isolation level read commited;
start transaction … -- Início da transação
```

Para exemplificarmos o funcionamento de uma transação explícita, iremos incluir uma linha na tabela Funcionário, executando o comando *rollback* para descartar essa inclusão.

Script:
```
set autocommit = off; -- Desabilita o autocommit
start transaction; -- Inicia a transação
insert into funcionario values(
     12, 'Wilquison Fontes', 'M', '2001-04-01',
     9800, 3, 1, 'wilquison.fontes@email.com', 7);
select * from funcionário
     where matricula = 12; -- Pausa
```

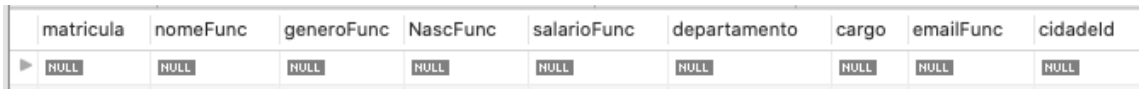
Figura 8 – *Select* antes do *rollback*

matricula	nomeFunc	generoFunc	NascFunc	salarioFunc	departamento	cargo	emailFunc	cidadeId
12	Wilquison Fontes	M	2001-04-01	9800.00	3	1	wilquison.fontes@email.com	7
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Fonte: Albano; Albano, 2022.

```
-- Continuação
rollback; -- Descarte
select * from funcionário
     where matricula = 12;
```

Figura 9 – *Select* após o *rollback*



matricula	nomeFunc	generoFunc	NascFunc	salarioFunc	departamento	cargo	emailFunc	cidadeId
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Fonte: Albano; Albano, 2022.

Ressaltamos a importância do uso de transações, uma vez que seu principal objetivo é manter a integridade das informações armazenadas no Banco de Dados. Ao executarmos um conjunto de operações encadeadas (transações), ou seja, que possuem dependência uma da outra, teremos a garantia, em caso de erro, que as mesmas serão descartadas, mantendo, assim, os dados íntegros.

Agora que você aprendeu sobre índices, *views* e transações, exercite esses conceitos nas tabelas do nosso estudo de caso. Dessa forma, você colocará em prática o conteúdo estudado. **Nota:** em cada seção dedicada a um novo comando, as tabelas que estão sendo utilizadas como base para a execução dos exemplos serão sempre atualizadas com os mesmos dados originais, a fim de evitar limitações nos exemplos.

TEMA 2 – *TRIGGER*

Em um Banco de Dados, é usual existirem ações que são disparadas automaticamente em resposta à outra ação. Tal ocorrência ou objeto é denominado de *trigger* ou gatilho e, normalmente, essas ações realizam validações, cálculos, registros de *logs*, entre outros.

Diretamente relacionada a uma tabela, a *trigger* é disparada em resposta a ação de um comando *insert*, *update* ou *delete*, podendo ser disparada antes ou depois da execução do comando que disparou a sua execução. O padrão SQL ANSI define dois tipos de *triggers*:

1. Em nível de linha – O disparo da *trigger* ocorre para cada linha inserida, excluída ou atualizada. Por exemplo, se 10 linhas forem excluídas em uma tabela, a *trigger* será executada 10 vezes.
2. Em nível de instrução (comando) – A *trigger* é executada uma vez por instrução. Considerando a mesma situação anterior, a *trigger* será disparada somente uma vez.

É importante ressaltar que o MySQL implementa somente a *trigger* em nível de linha.

Sintaxe:
```
create or [replace] trigger nomeTrigger
{momentoDisparado}{eventoDisparado}
on nomeTabela
for each row
begin
     /* Código da trigger */
end;
```

Onde:

create or replace	Indica a criação ou a alteração de uma *trigger*.
nomeTrigger	Nome da *trigger*.
momentoDisparado	Momento de execução da *trigger*, que pode ocorrer *before* (antes) ou *after* (depois) do comando.
eventoDisparado	Indica qual evento ou comando que irá disparar a *trigger* (*insert*, *update* ou *delete*).
nomeTabela	Nome da tabela que a *trigger* está associada. Não pode haver mais de uma *trigger* associada a mesma tabela ou ao mesmo comando.
for each row	Indica que a *trigger* é do tipo em nível de linha.

Referências *new* e *old*:

As *triggers* são executadas em conjunto com os comandos *insert*, *update* e *delete*. Para que uma *trigger* possa acessar as linhas (registros) que estão sendo manipuladas, é necessário fazer uso das referências *new* e *old*, as quais são declaradas conforme o tipo de comando utilizado, sendo:

- *New* – Utilizado nos comandos *insert* e *update*;
- *Old* – Utilizado nos comandos *delete* e *update*.

Comandos úteis:

1. Visualizar as *triggers* do Banco de Dados.

   Sintaxe:
   ```
   show triggers
   [{from | in} nomeBancoDados]
   [where = condição];
   ```

   Exemplos:
   ```
   -- Mostra todas as triggers do servidor
   ```

```
show triggers;
-- Mostra todas as triggers do Banco de Dados aula
show triggers from aula;
-- Mostra todas as triggers da tabela Funcionário
show triggers from aula where table = 'funcionario';
```

2. Exclusão de uma *trigger*.

   Sintaxe: **`drop trigger nomeTrigger;`**

   Exemplo: Primeiramente iremos implementar uma tabela Auditoria para as operações realizadas na tabela Funcionário. Nessa tabela serão armazenadas as seguintes informações:

- Inclusão – Matrícula, salário e data da inclusão;
- Exclusão – Matrícula e data da exclusão;
- Alteração – Matrícula, salário anterior, novo salário e data da alteração.

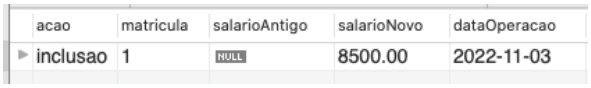
```
create table auditoria(
   acao char(10),
   matricula int,
   salarioAntigo decimal(10, 2),
   salarioNovo decimal(10, 2),
   dataOperacao date);

-- Triger disparada na inclusão
delimiter $$ -- Declaração de um delimitador de comandos
create trigger funcionarioInclusao after insert
on funcionario
for each row
begin
   insert into auditoria values
   ('inclusao', new.matricula, null, new.salarioFunc,
   curdate());
end$$
delimiter;
```

Perceba que estamos criando uma *trigger* que será disparada após (*after*) o comando *insert* na tabela Funcionário (**`on funcionario`**), inserindo os dados

na tabela Auditoria e buscando os dados na referência *new* das colunas do funcionário que está sendo incluído (**new.matricula** e **new.salarioFunc**).

```
-- Incluir funcionário
insert into funcionario values(
   1, 'Ana Rosa Silva', 'F', '1996-12-31', 8500, 1, 1,
   'ana.rosa@email.com', 1);
-- Mostra os dados da tabela Auditoria
select * from auditoria;
```

Figura 10 – *Select* na tabela Auditoria depois do comando *insert*



acao	matricula	salarioAntigo	salarioNovo	dataOperacao
inclusao	1	NULL	8500.00	2022-11-03

```
delimiter $$
create trigger funcionarioAlteracao after update
on funcionario
for each row
begin
   insert into auditoria values(
      'Alteracao', new.matricula, old.salarioFunc,
      new.salarioFunc, curdate());
end$$
delimiter;

update funcionario set salarioFunc = 10500
   where matricula = 1;
```

Nessa *trigger,* estamos utilizando as duas referências, *new* e *old*. A referência *new* armazena o novo salário do funcionário, enquanto que a *old* contém o salário antigo do funcionário. Assim, conseguimos trabalhar com as duas informações de forma simultânea.

```
-- Mostrar os dados da tabela Auditoria
select * from auditoria;
```

Figura 11 – *Select* na tabela Auditoria depois do comando *update*

acao	matricula	salarioAntigo	salarioNovo	dataOperacao
inclusao	1	NULL	8500.00	2022-11-03
Alteracao	1	8500.00	10500.00	2022-11-03

Fonte: Albano; Albano, 2022.

Agora nos falta desenvolver a *trigger* que irá fazer o controle de auditoria no momento da exclusão de uma linha.

```
delimiter $$
create trigger funcionarioExclusao after delete
on funcionario
for each row
begin
   insert into auditoria values(
      'Exclusao', old.matricula, old.salarioFunc,
      old.salarioFunc, curdate());
end$$
delimiter;

delete from funcionario where matricula = 1;
```

No caso do comando *delete*, o mesmo segue o princípio já comentado na *trigger* do comando *insert*.

```
-- Mostrar os dados da tabela Auditoria
select * from auditoria;
```

Figura 12 – *Select* na tabela Auditoria depois do comando *delete*

acao	matricula	salarioAntigo	salarioNovo	dataOperacao
inclusao	1	NULL	8500.00	2022-11-03
Alteracao	1	8500.00	10500.00	2022-11-03
Exclusao	1	10500.00	10500.00	2022-11-03

Exemplo de *trigger before*: vamos criar uma *trigger* para implementar o autoincremento na coluna 'matricula'. Nesse caso, é necessário calcular o número da matrícula antes da linha ser inserida. Assim, a melhor opção de *trigger* é a *before*.

```
delimiter $$
create trigger funcionarioNovoID before insert
on funcionario
for each row
begin
   set new.matricula = (select max(matricula) + 1
   from funcionario);
end$$
delimiter;

insert into funcionário
   (nomeFunc, generoFunc, NascFunc, salarioFunc,
   departamento, cargo, emailFunc, cidadeId)
   values ('Tales Heitor Souza', 'M', '2000-10-01', 7689,
   1, 2, 'tales.heitor@email.com', 1);
```

Figura 13 – *Select* depois do comando *insert*

matricula	nomeFunc	generoFunc	NascFunc	salarioFunc	departamento	cargo	emailFunc	cidadeId
2	Tales Heitor Souza	M	2000-10-01	7689.00	1	2	tales.heitor@email.com	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Fonte: Albano; Albano, 2022.

```
-- Mostra as triggers do Banco de Dados aula
show triggers from aula;
```

Figura 14 – Show triggers do Banco de Dados Aula

Trigger	Event	Table	Statement		Timing
funcionarioNovoID	INSERT	funcionario	begin	set new.matricula = (sel…	BEFORE
funcionarioInclusao	INSERT	funcionario	begin	insert into auditoria valu…	AFTER
funcionarioAlteracao	UPDATE	funcionario	begin	insert into auditoria valu…	AFTER
funcionarioExclusao	DELETE	funcionario	begin	insert into auditoria valu…	AFTER

Fonte: Albano; Albano, 2022.

Agora que você aprendeu sobre *trigger*, exercite esse conceito nas tabelas do nosso estudo de caso. Dessa forma, você colocará em prática o conteúdo estudado.

TEMA 3 – *STORED PROCEDURE*

Conjunto de instruções da *Structured Query Language* (SQL) a serem executadas pelo Sistema Gerenciador de Banco de Dados (SGBD) para realizar uma determinada tarefa. Em seu interior, são declaradas instruções de seleção, controle e repetição, podendo realizar alterações nos dados armazenados nas tabelas, porém sem retorno de valores. Também podem ou não receber parâmetros de entrada.

São usadas para realizar tarefas diárias, repetitivas em um Banco de Dados, como por exemplo, comunicação com as aplicações, realização de *backups*, entre outras.

Utiliza comandos de comparação, comandos de repetição (*loop*), variáveis e funções. Além disso, podem ser executadas pelas linguagens de programação. Algumas vantagens de usar *procedures* são:

- Várias aplicações escritas em diferentes linguagens, ou que rodam em plataformas diferentes, porém, que executam a mesma função;
- Prioridade a consistência e a segurança dos dados armazenados;
- Ocultar a complexidade de acesso ao Banco de Dados;
- Facilita e centraliza o gerenciamento de permissões;
- Melhora a velocidade de execução.

Sintaxe:

```
delimiter $$
create or replace procedure
   nomeProcedimento([tipo] parâmetro tipoDado,
   ..., [tipo] parâmetroN [tipoDadoN])
begin
   /* Bloco de comandos */
end $$
delimiter;
```

Onde:

create or replace	Indica a criação ou alteração da *procedure*.
nomeProcedimento	Nome da *procedure*.
parâmetro	Nome do parâmetro.
tipoDado	Tipo de dado do parâmetro.
tipo	Tipo de parâmetro, o qual pode ser [*in*, *out*, *inout*].

Tipos de parâmetros:

- *in* – Parâmetro de entrada, ou seja, o valor deve ser informado no momento da chamada da *procedure*. Tal valor pode sofrer alteração durante a execução da *procedure*;
- *out* – Indica um parâmetro de saída, ou seja, um valor retornado pela execução da *procedure*;
- *inout* – Combinação de *in* ou *out*, indicando que o parâmetro pode ser informado na chamada da *procedure* e na saída, retornando ao final da execução.

Executando uma *procedure*:

Sintaxe: `call nomeProcedimento(parâmetro);`

Comandos úteis:

1. Mostrar as *procedures*.

   Sintaxes:
   ```
   show procedure status
      where definer = 'root@localhost';
   ```
   ou
   ```
   select * from information_schema.routines
      where routine_schema = 'nomeBancoDados'
      and routine_type = 'PROCEDURE';
   ```

2. Excluir as *procedures*.

   Sintaxe:
   ```
   drop procedure nomeProcedure;
   ```

Comandos condicionais e de repetição:

1. *If*: Utilizado para testes condicionais.

   Sintaxe:
   ```
   if condição then comandos
      [elseif condição then comandos] ...
      [else comandos]
   end if
   ```

2. *Case*: Compara um valor com várias condições até encontrar uma verdadeira. Caso não exista correspondência, o comando *else* será executado.

   Sintaxe:
   ```
   case valor
      when condição then comandos
      [when condição then comandos] ...
   ```

```
        [else comandos]
    end case
```

3. *Loop*, *while-do* e *repeat-until*: Comandos que permitem a execução repetida de um bloco de comandos, sendo a instrução finalizada com ponto e vírgula (;). A sintaxe de cada comando é:

- *Loop* – Normalmente usamos os comandos *iterate* para retornar ao início do *loop* ou *leave* para finalizar o *loop*.

Sintaxe:
```
[identificador:] loop
    /* Bloco de comandos */
    if condição then
        iterate [identificador];
    end if;
    leave [identificador];
end loop [identificador];
```

- *While-do* – O bloco de comandos será executado enquanto (*while*) a condição for verdadeira. Esse comando pode não ser executado se a condição for falsa desde o início. O teste da condição é realizado no início do comando.

Sintaxe:
```
[identificador:] while condição do
    /* Bloco de comandos */
end while [identificador]
```

- *Repeat-until*: O bloco de comandos é executado até (*until*) que a condição se torne verdadeira. O teste da condição é realizado no final, assim, o comando é executado pelo menos uma vez.

Sintaxe:
```
[identificador] repeat
    /* Bloco de comandos */
    until condição
end repeat [identificador]
```

Exemplo: vamos criar uma *procedure* para gerar aleatoriamente 6 números para uma aposta de loteria. O intervalo dos números será informado como parâmetro na *procedure*. Os números gerados serão armazenados em uma tabela juntamente com o número do concurso da loteria.

```
-- Tabela que armazena o concurso e os números gerados
create table cartela(
   concurso int,
   numero int);

delimiter $$
create procedure geraCartela(
   nroInicial int,
   nroFinal int,
   nroConcurso int)
begin
   declare nroGerado int default 0;
   declare i int default 0;
   geraNumero: loop
      set nroGerado = (select floor(rand() * nroFinal)
         + nroInicial);
      if not exists (select * from cartela
         where numero = nroGerado) then
         insert into cartela values(
            nroConcurso, nroGerado);
         set i = i + 1;
      end if;
      if i < 6 then
         iterate geraNumero;
      end if;
      leave geraNumero;
   end loop geraNumero;
end $$
```

Perceba que utilizamos o comando *loop* para implementar o bloco de repetição, sendo que isso foi proposital. O *loop* é diferente dos comandos *while* e *repeat*, inclusive é necessário criar um identificador do bloco, nesse caso, o “geraNumero”.

Para cada repetição do bloco, é gerado um número aleatório entre o intervalo informado na chamada da *procedure* (‘nroInicial’ e ‘nroFinal’). Além disso, verificamos se o número gerado não existe e, somente nesse caso,

gravamos na tabela e incrementamos o contador (variável ‘i’). Enquanto o contador for inferior a 6, retornamos para o início do bloco (*iterate*) ou, caso contrário, encerramos o bloco de repetição (*leave*).

Executando a *procedure* desenvolvida:

```
call geraCartela(1, 60, 200);
select * from cartela;
```

Figura 15 – Números gerados

concurso	numero
200	4
200	54
200	18
200	46
200	57
200	25

Fonte: Albano; Albano, 2022.

Agora que você aprendeu sobre o comando *loop*, vamos mostrar um trecho de *procedure* com os comandos *while* e *repeat*, sendo:

- *Script while*:

Figura 16 – *Procedure* com *while*

```
delimiter $$
create procedure geraCartela(nroInicial int, nroFinal int, nroConcurso int)
begin
    declare nroGerado int default 0;
    declare i int default 0;
    while i < 6 do
        -- - gera o número aleatório
        set nroGerado =  (select floor(rand() * nroFinal) + nroInicial);
        -- verifica se o número já não foi gerado e grava na tabela
        if  not exists (select * from cartela where numero = nroGerado) then
            insert into cartela values (nroConcurso, nroGerado);
            set i = i + 1;
        end if;
    end while;
end $$
```

Fonte: Albano; Albano, 2022.

- *Script repeat*:

Figura 17 – *Procedure* com *repeat*

```
delimiter $$
create procedure geraCartelaRepeat(nroInicial int, nroFinal int, nroConcurso int)
begin
    declare nroGerado int default 0;
    declare i int default 0;
    repeat
        -- - gera o número aleatório
        set nroGerado =  (select floor(rand() * nroFinal) + nroInicial);
        -- verifica se o número já não foi gerado e grava na tabela
        if  not exists (select * from cartela where numero = nroGerado) then
            insert into cartela values (nroConcurso, nroGerado);
            set i = i + 1;
        end if;
     until i > 5
     end repeat;
end $$
```

Fonte: Albano; Albano, 2022.

Para mostrar as *procedures* do Banco de Dados, a sintaxe é:

```
show procedure status where definer = 'root@localhost';
```

Figura 18 – *Show procedure*

Db	Name	Type	Definer
aula	geraCartela	PROCEDURE	root@localhost

Fonte: Albano; Albano, 2022.

Agora que você aprendeu sobre *procedure*, exercite esse conceito no nosso estudo de caso. Dessa forma, você colocará em prática o conteúdo estudado.

TEMA 4 – *FUNCTION*

Uma função desenvolvida pelo usuário é sintaticamente semelhante a uma *stored procedure*, porém, ao final da execução, sempre retornará um valor ou um conjunto de valores de qualquer tipo de dado válido. Além disso, uma função tem como objetivo suprir tarefas específicas (como por exemplo, separar o ano de uma data), enquanto uma *procedure* é mais abrangente (como por exemplo, tratamento dos dados de uma planilha no Excel, realização de *backup*, entre outras tarefas).

Basicamente, uma função é um segmento de código (rotina em SQL) que recebe parâmetros de entrada, processa as instruções definidas e retorna ao ponto de chamada com um resultado.

Você pode estar se perguntando: qual o motivo de trabalharmos com esse recurso? Ao longo do seu trabalho na área de Banco de Dados, você irá perceber que existem situações que o MySQL não oferece suporte (de forma nativa) e, assim, surgirá a necessidade de criar suas próprias funções.

No MySQL, existem dois tipos de funções definidas pelo usuário, as quais são:

1. Escalares – Que retornam apenas um valor.
2. Composta – Que retorna um conjunto de valores como resultado.

Declarando uma função:

Sintaxe:

```
delimiter $$
create function funçãoNome(
   parâmetro1 tipo_dados, ...,
   parâmetroN tipo_dados)
returns tpDadoRetorno [not] deterministic
begin
   instruções
   return valor
end $$
delimiter;
```

Onde:

funçãoNome	Definição do nome da função.
parâmetro	Declaração de todos os parâmetros de entrada.
tpDadoRetorno	Tipo de dado do valor que será retornado.
deterministic	Uma função determinística retorna o mesmo resultado para os mesmos parâmetros de entrada. Já uma função não determinística retorna resultados diferentes para os mesmos parâmetros.
instruções	Código que a função deve executar, declarado entre o *begin* e o *end*.
return valor	Retorna o valor e encerra a função.

Apresentando funções:

Como uma função definida pelo usuário também é um objeto no MySQL, a partir de sua criação, a mesma pode ser visualizada na ferramenta que

estamos usando (MySQL Workbench, item ‘Funções’) ou acessando essa mesma informação por meio da seguinte consulta:

```
show function status where db = 'nomeBancoDados';
```

ou

```
select * from information_schema.routines
   where routine_schema = 'nomeBancoDados'
   and routine_type = 'FUNCTION';
```

Executando uma função:

Uma função implementada por um usuário pode ser utilizada com qualquer comando (*select*, *insert*, *update*, *delete*, entre outros) em conjunto com o nome da função desejada e a declaração dos parâmetros de entrada necessários, os quais são definidos na função.

Sintaxes:

```
select nomeFunção(parâmetro);
select nomeFunção(parâmetro1, ..., parâmetroN)
   from nomeTabela;
```

Exemplo 1: vamos criar uma função que retorne se um número é par ou ímpar. Para isso, usaremos a função *mod*, que retorna o resto da divisão. Assim, ao dividir um número por dois, se o resto resultar em zero indica que o número é par.

```
delimiter $$
create function parImpar(numero int)
returns char(5) deterministic
begin
   declare tipo char(5) default null;
   set tipo = 'impar';
   if numero mod 2 = 0 then
      set tipo = 'par';
   end if;
   return (tipo);
end $$
delimiter;
```

Executando a função desenvolvida:

```
select parImpar(10), parimpar(5);
```

Figura 19 – Resultado da função ‘parImpar’

parImpar(10)	parimpar(5)
par	impar

Fonte: Albano; Albano, 2022.

Exemplo 2: Vamos criar agora uma outra função que retorne a idade de um funcionário. Para isso, usaremos a data atual subtraindo da data de nascimento e retornaremos à idade do funcionário.

```
delimiter $$
create function CalculaIdade(nascimento date)
returns int deterministic
begin
   declare dataAtual date;
   declare idade int default 0;
   set dataAtual = (select curdate());
   set idade = year(dataAtual) - year(nascimento);
   return idade;
end $$
delimiter;
```

Executando a função desenvolvida:

```
select nomeFunc, nascFunc, CalculaIdade(nascFunc)
   as idade from funcionario;
```

Figura 20 – Resultado da função ‘CalculaIdade’

nomeFunc	nascFunc	idade
Ana Rosa Silva	1996-12-31	26
Tales Heitor Souza	2000-10-01	22
Bia Meireles	2002-03-14	20

Fonte: Albano; Albano, 2022.

Lembra da *procedure* que criamos anteriormente? A *procedure* que gera os números para um concurso da loteria? Dentro da *procedure* existe uma instrução para gerar os números de forma aleatória, instrução essa que

poderíamos retirar da *procedure*, criando uma função contendo essa instrução e, assim, na *procedure* apenas chamaríamos a função. Vamos ver como se faz?

Criando a função de número aleatório:

```
delimiter $$
create function numeroAleatorio(inicio int, final int)
returns int deterministic
begin
    declare numero int default 0;
    set numero = (select floor(rand() * final) + inicio);
    return numero;
end $$
delimiter;
```

Logo, a *procedure* ficaria assim:

Figura 21 – *Procedure* chamando uma função

```
delimiter $$
create procedure geraCartela(nroInicial int, nroFinal int, nroConcurso int)
begin
    declare nroGerado int default 0;
    declare i int default 0;
    while i < 6 do
        -- - gera o número aleatório
        set nroGerado =  (select numeroAleatorio(nroInicial, nroFinal));
        -- verifica se o número já não foi gerado e grava na tabela
        if  not exists (select * from cartela where numero = nroGerado) then
            insert into cartela values (nroConcurso, nroGerado);
            set i = i + 1;
        end if;
    end while;
end $$
```

Fonte: Albano; Albano, 2022.

Mostrando as *procedures* e funções do Banco de Dados:

```
show procedure status where definer = 'root@localhost';
```

Figura 22 – *Show procedure*

SPECIFIC_NAME	ROUTINE_CATALOG	ROUTINE_SCHEMA	ROUTINE_NAME	ROUTINE_TYPE
CalculaIdade	def	aula	CalculaIdade	FUNCTION
numeroAleatorio	def	aula	numeroAleatorio	FUNCTION
parImpar	def	aula	parImpar	FUNCTION

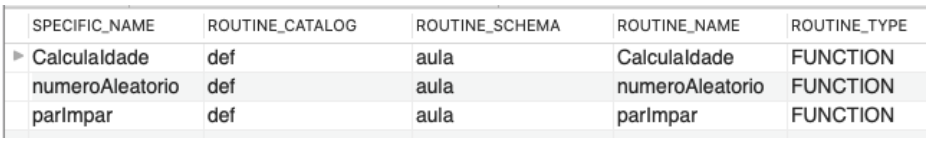
Fonte: Albano; Albano, 2022.

```
show function status where db = 'aula';
```

ou

```
select * from information_schema.routines
    where routine_schema = 'aula'
    and routine_type = 'FUNCTION';
```

Figura 23 – *Show function*



SPECIFIC_NAME	ROUTINE_CATALOG	ROUTINE_SCHEMA	ROUTINE_NAME	ROUTINE_TYPE
CalculaIdade	def	aula	CalculaIdade	FUNCTION
numeroAleatorio	def	aula	numeroAleatorio	FUNCTION
parImpar	def	aula	parImpar	FUNCTION

Fonte: Albano; Albano, 2022.

Agora que você aprendeu sobre *function*, exercite esse conceito no nosso estudo de caso. Dessa forma, você colocará em prática o conteúdo estudado.

TEMA 5 – CURSOR

Ao executarmos um comando *select*, *update* ou *delete* em uma tabela do Banco de Dados, várias linhas são acessadas de uma só vez. Obviamente, o conjunto de registros retornados depende do tamanho da tabela e da forma como foi consultada, juntamente com a cláusula *where*, que filtra os dados selecionados.

Existem situações em que trazer os registros de uma só vez não é uma forma conveniente ou possível para realizar certos tipos de operações, uma vez que pode existir a necessidade do resultado linha a linha.

Nesses casos, os Sistemas Gerenciadores de Banco de Dados (SGBDs) fornecem um recurso bastante interessante, chamado de cursor, o qual consiste de uma instrução *select* por meio de um laço (*loop*, *while* ou *repeat*) e comandos específicos para cursores, realizando o acesso linha a linha ao resultado obtido na consulta.

Tal instrução manipula o resultado do *select*, alocando uma variável na memória com o resultado obtido por meio da consulta. Dessa forma, permite a realização de operações (atualização, exclusão ou movimentação de dados) sobre linhas individuais de um resultado. Geralmente, são utilizados em *procedures*. Características dos cursores:

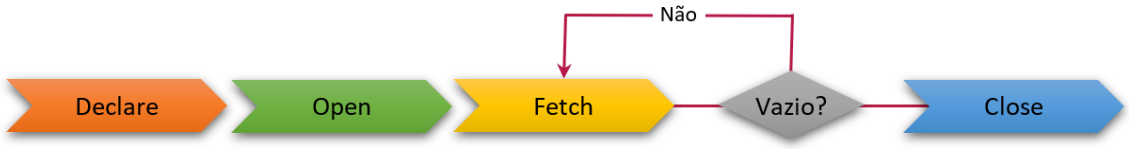
- Um cursor é somente para leitura, não sendo possível realizar diretamente atualização ou exclusão de dados no conjunto de dados obtido no resultado (*read only*);

- A leitura dos dados por meio do cursor ocorre sequencialmente, isto é, não é possível pular, e sempre na ordem definida pelo *select* (*non_scrollable*);
- No MySQL, o cursor pode carregar os dados na memória criando uma tabela temporária (cópia) ou, ainda, apontar diretamente para o conjunto de dados obtido na consulta (*asensitive*).

Etapas para utilização do cursor: para o pleno funcionamento do cursor, existe um ciclo de vida, ou seja, uma sequência de etapas que devem ser criteriosamente obedecidas, as quais são:

1. Declaração do cursor (*declare*, alocação de memória);
2. Abertura do cursor (*open*);
3. Execução do cursor (*fetch*);
4. Fechamento e liberação do cursor (*close*).

Figura 24 – Ciclo de vida do cursor



Fonte: Albano; Albano, 2022.

Declarando um cursor: na primeira etapa do ciclo de vida do cursor utilizamos o comando *declare* acompanhado da palavra-chave *cursor*.

Sintaxe: `declare cursorNome cursor for expressãoSelect;`

Onde:

cursorNome	Nome definido para o cursor.
expressãoSelect	Declaração da instrução *select* que é associada ao cursor.

Nota: Como boa prática, recomenda-se que a criação de um cursor no MySQL seja sempre realizada por último, ou seja, declarando todas as variáveis que serão necessárias antes do cursor.

Abrindo um cursor: depois que criamos um cursor, para utilizá-lo é necessário requerer a abertura do cursor, identificando o mesmo por um nome.

Sintaxe: `open cursorNome;`

Executando um cursor: com o cursor aberto podemos acessar os dados armazenados em *procedures*, *triggers* ou funções. Para tanto, usamos o comando *fetch* em conjunto com a cláusula *into*.

Sintaxe: **`fetch cursorNome into listaVariáveis;`**

Recorde que para evitar erros devem ser declaradas todas as variáveis antes de realizar esse procedimento, uma vez que o número de colunas recuperadas pela instrução *select* deve corresponder ao número de variáveis de saída especificadas na instrução *fetch*.

Fechando e liberando um cursor: após a conclusão de todas as instruções que utilizam o cursor, devemos encerrar o uso do cursor e liberar a memória que foi alocada.

Vale salientar que mesmo que um cursor tenha sido fechado, não será necessária uma nova declaração para ativá-lo novamente, sendo apenas necessário executar o comando *open* para reabrir o cursor.

Sintaxe: **`close cursorNome;`**

Tratamento de erros: como já comentado, no MySQL o cursor lê o conjunto de resultados de forma sequencial. Porém, pode tornar-se inconveniente se, por algum motivo, uma linha não estiver disponível. Caso isso ocorra e não houver uma ação predefinida, o cursor fechará e não retornará os dados.

Para minimizar esse tipo de erro, é importante tratarmos de forma apropriada com a predefinição de uma ação, a qual será executada se uma linha estiver indisponível. Para isso, utilizamos a instrução *declare handler*, que trata de avisos ou exceções.

Sintaxe:
```
declare ação handler for not found
        set varFinal = true;
```

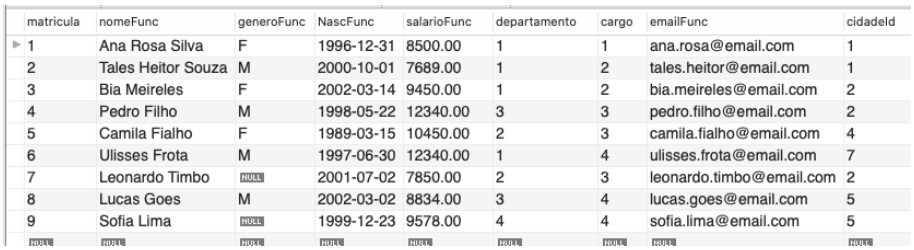
Onde:

- A ação pode assumir dois estados, isto é, manter a execução com a ação *continue* ou finalizar a execução com a ação *exit*;
- A variável final define que o cursor chegou ao final do conjunto de resultados.

Exemplo: A empresa solicitou uma simulação de reajuste para salários dos funcionários e, sendo assim, criaremos um cursor que possibilite fazer essa simulação, cujas regras são:

- Funcionários do departamento 1 terão 10% de aumento;
- Funcionários do departamento 2 terão 12% de aumento;
- Demais funcionários terão 8% de aumento.

Figura 25 – *Select* na tabela Funcionário



matricula	nomeFunc	generoFunc	NascFunc	salarioFunc	departamento	cargo	emailFunc	cidadeId
1	Ana Rosa Silva	F	1996-12-31	8500.00	1	1	ana.rosa@email.com	1
2	Tales Heitor Souza	M	2000-10-01	7689.00	1	2	tales.heitor@email.com	1
3	Bia Meireles	F	2002-03-14	9450.00	1	2	bia.meireles@email.com	2
4	Pedro Filho	M	1998-05-22	12340.00	3	3	pedro.filho@email.com	2
5	Camila Fialho	F	1989-03-15	10450.00	2	3	camila.fialho@email.com	4
6	Ulisses Frota	M	1997-06-30	12340.00	1	4	ulisses.frota@email.com	7
7	Leonardo Timbo	NULL	2001-07-02	7850.00	2	3	leonardo.timbo@email.com	2
8	Lucas Goes	M	2002-03-02	8834.00	3	4	lucas.goes@email.com	5
9	Sofia Lima	NULL	1999-12-23	9578.00	4	4	sofia.lima@email.com	5
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Fonte: Albano; Albano, 2022.

Normalmente, o cursor é declarado dentro de uma *procedure* e, por esse motivo, criaremos uma *procedure* que conterá o cursor com os dados dos funcionários (nome, departamento e salário atual). Após calcularmos o novo salário do funcionário, armazenaremos em uma nova tabela o nome, o salário atual e o novo salário.

```
-- Tabela para armazenar a simulação do reajuste
create table simulacao(
   nome varchar(100),
   salario decimal(10, 2),
   novoSalario decimal(10, 2));


-- Procedure para calcular o reajuste
delimiter $$
create procedure simulaReajuste()
begin
   -- Variável para identificar o final do cursor
   declare done boolean default false;
   declare vnome varchar(100);
   declare vsalario decimal(10, 2);
   declare vnovoSalario decimal(10, 2);
   declare vdepartamento int;
```

```
        declare cursorFuncionario cursor for
            select nomeFunc, departamento, salarioFunc
                from funcionario;
        -- Altera o status da variável de fim do cursor
        declare continue handler for not found set done = true;
        open cursorFuncionario;
        leCursor: loop
            fetch cursorFuncionario into
                vnome, vdepartamento, vsalario;
            if done then -- Testa se o cursor chegou ao final
                leave leCursor; -- Sai do loop no final
            end if;
            if vdepartamento = 1 then
                set vnovoSalario = vsalario * 1.10;
            elseif vdepartamento = 2 then
                set vnovoSalario = vsalario * 1.12;
            else
                set vnovoSalario = vsalario * 1.08;
            end if;
            insert into simulacao values(
                vnome, vsalario, vnovoSalario);
        end loop;
        close cursorFuncionario;
    end $$
    delimiter;
```

Perceba que declaramos um cursor (**cursorFuncionario**) que contém três colunas no *select* (**nome**, **departamento** e **salario**). Dessa forma, necessitamos ter o mesmo número de variáveis (**vnome**, **vdepartamento** e **vsalario**) para receber esses valores no momento da leitura do cursor (*fetch*).

Também declaramos uma variável 'done' do tipo *boolean*, a qual é usada pelo *handler* para controlar o final do cursor. Após a leitura linha a linha, quando chegar ao final do cursor, o *handler* irá alterar o valor da variável 'done' para verdadeiro. Assim, saberemos que chegamos ao fim do cursor, saindo do bloco de leitura do mesmo.

Como já comentado, apenas após a declaração do cursor é possível a sua abertura (*open*) para que possamos manipular os dados que estão armazenados nele. Também já sabemos que um cursor pode possuir zero ou várias linhas e, assim, devemos usar um comando de repetição para realizar a leitura linha a linha até terminar. O comando para realizar essa leitura é o *fetch*.

Mostrando o resultado da simulação:

```
select * from simulacao;
```

Figura 26 – Resultado da simulação

nome	salario	novoSalario
Ana Rosa Silva	8500.00	9350.00
Tales Heitor Souza	7689.00	8457.90
Bia Meireles	9450.00	10395.00
Pedro Filho	12340.00	13327.20
Camila Fialho	10450.00	11704.00
Ulisses Frota	12340.00	13574.00
Leonardo Timbo	7850.00	8792.00
Lucas Goes	8834.00	9540.72
Sofia Lima	9578.00	10344.24

Fonte: Albano; Albano, 2022.

Concluímos essa seção sugerindo novamente que você utilize os conhecimentos adquiridos, exercitando esse novo conceito no nosso estudo de caso. Dessa forma, você colocará em prática o conteúdo estudado.

FINALIZANDO

Ao longo deste estudo, aprendemos muito sobre Banco de Dados. Percorremos diversos temas, estudando e aplicando conceitos de forma prática, simulando situações do cotidiano, discutindo sobre assegurar a integridade e a segurança dos dados, tanto na aplicação de protocolos e restrições quanto também com boas práticas, ressaltando sempre a necessidade de estarmos atentos ao manipularmos um Banco de Dados.

Gostaríamos de expressar o grande prazer que foi poder compartilhar conhecimento e fazer parte desse momento de aprendizado. Esperamos que de forma simples e honesta tenhamos contribuído em seu caminho, fornecendo subsídios para que você possa se tornar um profissional com espírito crítico e em constante desenvolvimento.

Para você que respira informação, perceba que está ingressando ou já atua no melhor segmento do mercado, sendo responsável por fornecer a

estrutura e suporte para todas as demais áreas. Então, sinta-se orgulhoso de fazer parte desse seleto grupo de pessoas tão necessárias para o perfeito equilíbrio entre a sociedade e a informação. Seja muito feliz e sucesso!

REFERÊNCIAS

BEIGHLEY, L. **Use a Cabeça SQL**. Rio de Janeiro: Alta Books, 2010.

HEUSER, C. A. **Projeto de Banco de Dados**. Porto Alegre: Bookman, 2009.

MYSQL DOCUMENTATION. Disponível em: <https://dev.mysql.com/doc>. Acessado em: 12 jan. 2023.

SETZER, V. W. **Banco de Dados:** Conceitos, modelos, gerenciadores, projeto lógico, projeto físico. 3. ed. São Paulo: Edgard Blücher, 1986.