UniATENEU
Centro Universitário

Banco de Dados de Alta Performance

1ª Edição

UniATENEU
Centro Universitário

BANCO DE DADOS

Autoria:
Geraldo Barbosa do Amarante

ATENEU Online
EAD
CONECTANDO VALORES

UniATENEU
Centro Universitário

Expediente

Reitor:
Prof. Cláudio Ferreira Bastos

Pró-reitor administrativo financeiro:
Prof. Rafael Rabelo Bastos

Pró-reitor de relações institucionais:
Prof. Cláudio Rabelo Bastos

Pró-reitor acadêmico:
Prof. Herbert Gomes Martins

Direção EAD:
Prof. Ricardo Zambrano Júnior

Coordenação EAD:
Profa. Luciana Rodrigues Ramos

Ficha Técnica

Autoria:
Geraldo Barbosa do Amarante

Supervisão de produção EAD:
Francisco Cleuson do Nascimento Alves

Design instrucional:
Antonio Carlos Vieira

Projeto gráfico e capa:
Francisco Erbínio Alves Rodrigues

Diagramação e tratamento de imagens:
Ismael Ramos Martins

Revisão textual:
Antonio Carlos Vieira

Ficha Catalográfica
Catalogação na Publicação
Biblioteca Centro Universitário Ateneu

AMARANTE, Geraldo Barbosa do. Banco de dados. Geraldo Barbosa do Amarante. – Fortaleza: Centro Universitário Ateneu, 2020.

172 p.

ISBN: 978-65-88268-36-0

1. Sistemas Gerenciadores de Banco de Dados (SGBD). 2. Linguagem SQL. 3. Técnicas avançadas de dados. 4. Bancos de dados em alta disponibilidade e alta performance. Centro Universitário Ateneu. II. Título.

SEJA BEM-VINDO!

Caro estudante, este é o material da disciplina *Banco de dados*, que pretende ser o guia para o estudo e entendimento da tecnologia de banco de dados, tecnologia esta que é a base para todo e qualquer sistema computadorizado. De nada adiantaria sistemas sofisticados de software se não tivéssemos onde guardar as informações produzidas por eles, de uma forma segura e disponível.

Podemos dizer que o banco de dados é a "alma da empresa", não importando o seu porte no mundo dos negócios, seja uma empresa gigantesca, como a Amazon ou a Google, ou uma microempresa familiar, a questão da tomada de decisões baseada em dados independe do seu tamanho, é vital para a sua sobrevivência e crescimento no mercado.

A revista *The Economist*, em 2017, trouxe uma frase que transmite bem o quanto é importante gerir os dados da empresa, a frase dizia que *"o recurso mais valioso do mundo não é o petróleo, mas sim os dados"*. Isto significa que é crucial as empresas manterem as informações de qualidade sobre os seus clientes, fornecedores, suas transações contábeis, processos etc. de forma segura e disponível.

Este livro está dividido em quatro unidades. A primeira unidade tem o objetivo de dar uma visão geral dos sistemas gerenciadores de banco de dados (SGBD), fornecendo uma base conceitual. Também discutiremos o modelo relacional que é mais amplamente utilizado na maioria dos sistemas gerenciadores de banco de dados comerciais. Veremos também tópicos relacionados a um projeto de um pequeno banco de dados a ser implementado nas unidades subsequentes. Na segunda unidade, aprenderemos a instalar e configurar o SGBD MySQL, assim como a ferramenta MySQL Workbench para que seja possível desenvolver uma aplicação prática dos conceitos aprendidos utilizando a linguagem SQL. Na terceira unidade, aprofundaremos mais nas características e recursos do SGBD MySQL, para aprendermos a tornar os nossos bancos de dados mais eficientes e com alto desempenho. Na quarta e última unidade, discutiremos assuntos relacionados à distribuição de banco de dados, seu monitoramento e como melhorar o desempenho desta estrutura.

Bons estudos!

Estes ícones aparecerão em sua trilha de aprendizagem e significam:

ANOTAÇÕES
Espaço para anotar suas ideias.

CONECTE-SE
Convocar o estudante para interagir no fórum tira-dúvidas.

CURIOSIDADE
Informação curiosa relacionada ao conteúdo.

EXERCÍCIO RESOLVIDO
Atividade explicativa para guiar o estudante.

FIQUE ATENTO
Informação complementar ao texto principal.

LINK WEB
Indicação de sites.

MATERIAL COMPLEMENTAR
Texto ou mídias complementares ao assunto da aula.

MEMORIZE
Tópico ou fato importante de lembrar.

OBJETIVOS DE APRENDIZAGEM
Objetivos de estudo do capítulo ou unidade.

PRATIQUE
Exercícios para fixar os conteúdos.

REFERÊNCIAS
Fontes de pesquisa citadas no texto.

RELEMBRE
Resumo do conteúdo estudado.

SUMÁRIO

01

SISTEMAS GERENCIADORES DE BANCO DE DADOS (SGBD)

1. Visão geral sobre sistemas gerenciadores de banco de dados 8
2. Tipos de banco de dados 14
3. O modelo relacional 14
4. Projeto de banco de dados - normalização 20
5. As formas normais mais utilizadas 23
6. O dicionário de dados 32
7. Aplicando a normalização 37
8. O diagrama de entidades e relacionamentos 42

Referências 47

02

O SISTEMA GERENCIADOR DE BANCO DE DADOS MYSQL E A LINGUAGEM SQL

1. O sistema gerenciador de banco de dados MySQL 50
   - 1.1. Instalação do MySQL 51
   - 1.2. Configuração 53
2. A linguagem SQL (*structure query language*) 59
   - 2.1. Utilizando o MySQL para a criação e manipulação de banco de dados 61
   - 2.2. Criação das tabelas do banco de dados 65
   - 2.3. Criação das chaves primárias 68
   - 2.4. Implementando a propriedade de AUTO INCREMENTO 69
   - 2.5. Implementando os relacionamentos entre as tabelas 71
   - 2.6. Manipulação dos dados 75
3. Tabelas temporárias 84
   - 3.1. Vantagens das tabelas temporárias 85
   - 3.2. Criação de tabelas temporárias 85
   - 3.3. Criação de tabelas temporárias com informações vindas de um comando *SELECT* 86
   - 3.4. Exclusão das tabelas temporárias 86

Referências 90

03

TÉCNICAS AVANÇADAS DE DADOS

1. Particionamento 94
1.1. Particionamento tipo *RANGE* 95
1.2. Particionamento tipo *LIST* 97
1.3. Particionamento tipo *COLUMN* 98
1.4. Particionamento do tipo *HASH* 102
1.5. Particionamento do tipo *KEY* 103
2. Indexação 104
2.1. Classificação dos índices conforme sua estrutura 106
2.2. Criação de índices no MySQL 106
3. Objetos avançados de banco de dados 107
3.1. Visões (*Views*) 107
3.2. Funções (*Functions*) 112
3.3. Gatilhos (*Triggers*) 116
4. Procedimentos armazenados (*stored procedures*) 121
Referências 129

04

BANCOS DE DADOS EM ALTA DISPONIBILIDADE E ALTA PERFORMANCE

1. A alta disponibilidade e a alta performance 132
2. A replicação 133
2.1. Teoria da replicação 135
2.2. Replicação no MySQL 145
3. *Cluster* 147
4. Monitoramento 151
4.1. Descrição dos processos de monitoramento e suporte 153
4.2. Ferramentas de monitoramento no MySQL . 161
5. Técnicas e dicas para alta performance de banco de dados 166
Referências 171

Unidade 03

Técnicas avançadas de dados

Apresentação

Nesta unidade, aprenderemos a utilizar a técnica de particionamento de tabelas para aumentar a velocidade das consultas. Iremos utilizar também as técnicas de indexação que são fundamentais para o desempenho das buscas nas tabelas de um banco de dados.

Aprenderemos também a criar e utilizar os objetos avançados de dados que exigem o conhecimento da linguagem SQL e são objetos que contribuem fortemente para o aumento da segurança e disponibilidade de um sistema de banco de dados.

Objetivos de aprendizagem

- *Dar o conhecimento teórico e prático através do uso da linguagem SQL, que permita ao aluno utilizar os objetos avançados de banco de dados no seu trabalho diário de desenvolvedor de banco de dados;*
- *Transformar conceitos aprendidos sobre otimização de banco de dados em uma experiência prática através da implementação de técnicas e do desenvolvimento de objetos avançados de dados que irão para o alto desempenho do banco de dados;*
- *Entender e saber aplicar as técnicas do particionamento e da indexação e como podem contribuir para o aumento do desempenho de um banco de dados, tornando-o um banco de dados de alto desempenho.*

1. Particionamento

O **particionamento** é um recurso disponibilizado pelos sistemas gerenciadores de banco de dados com a finalidade de permitir uma maior eficiência nas tarefas de **consulta**, **inclusão**, alteração e exclusão de dados, além de auxiliar nas tarefas de **manutenção** do banco de dados. É um recurso que deve ser bem planejado e dimensionado com rigor, pois do contrário, ao invés de otimizar o desempenho do banco de dados, pode chegar a comprometê-lo.

O conceito de particionamento está intimamente ligado ao **armazenamento físico** dos dados no banco de dados, uma vez que esta técnica permite que as tabelas sejam divididas em partes, e estas partes sejam direcionadas para locais diferentes no sistema de arquivos, ou seja, posso ter uma parte de uma tabela gravada em um disco e outra parte em um outro disco.

Para que o particionamento de uma tabela seja possível serão definidas **regras específicas** que servirão de base para a separação dos registros. Estas regras, geralmente, na literatura de banco de dados, são conhecidas como **função de particionamento** ou função de partição. Como exemplo, poderemos criar uma função de particionamento de uma tabela,

que define que todas as informações de vendedores da loja matriz, cujos registros estão em uma tabela individual, sejam gravados em um disco de um servidor da rede, enquanto que os demais vendedores das lojas filiais, também com informações constantes na mesma tabela, sejam direcionados para um outro disco.

Em **grandes ambientes** de banco de dados, o particionamento de tabelas é muito útil para a melhoria do desempenho nas consultas aos dados. Imaginemos um grande banco de dados como o de uma instituição bancária com tabelas muito grandes e com um grande número de acessos de usuários que fazem consultas a estas tabelas.

Existem dois tipos de particionamento, o **vertical** e o **horizontal**. Quando a tabela tem muitas linhas, utilizaremos o particionamento vertical. Se acontecer um crescimento muito grande no número de colunas de uma tabela, será necessário o particionamento horizontal, que é caracterizado pela retirada de um determinado número de colunas para uma nova tabela, no qual a chave primária será replicada e ambas as tabelas se relacionarão através da chave primária, portanto, teremos um relacionamento com cardinalidade "1 para 1". Como estamos utilizando o SGBD **MySQL**, ele só possui mecanismos para o particionamento vertical, permitindo que uma tabela tenha 1024 partições.

Os tipos de particionamento vertical mais utilizados são os seguintes:

1.1. Particionamento tipo RANGE

Neste tipo de particionamento, seleciona-se um **valor** ou uma **faixa de valores** de uma coluna, podendo-se utilizar expressões baseadas em uma coluna. Um ponto a observar neste tipo de particionamento, como se utiliza valores ou faixa de valores para as funções de partição, se for necessário particionar colunas do tipo data (*date*), hora (*time*) ou date e hora (*datetime*), será necessário utilizar funções do MySQL que retornem valores inteiros. Como exemplo, se for necessário fazer o particionamento tendo como base o mês de uma coluna cujo tipo de dado é *date* ou *datetime*, deveremos utilizar a função ***MONTH***(), que retornará o valor inteiro referente ao mês da data constante na coluna.

Os comandos SQL a seguir mostram a criação de uma tabela particionada, utilizando o particionamento do tipo RANGE.

Observação:

Iremos utilizar um novo banco de dados para a criação das tabelas de exemplo dos tipos de particionamento. Abaixo os comandos de criação do banco de dados e o comando de utilização.

```
CREATE DATABASE Uniateneu;    #Cria o banco de dados com nome Uniateneu

USE Uniateneu;                #Abre o banco de dados criados para ser utilizado
```

Exemplo do particionamento RANGE:

```
CREATE TABLE tbVendedor (

matricula INTEGER NOT NULL,

nome VARCHAR(100),

loja INT NOT NULL

)

PARTITION BY RANGE (loja) (

PARTITION maraponga VALUES LESS THAN (6),

PARTITION messejana  VALUES LESS THAN (11),

PARTITION bomjardim  VALUES LESS THAN (16),

PARTITION parangaba  VALUES LESS THAN (21),

PARTITION aldeota    VALUES LESS THAN (26)

);
```

Neste exemplo, utilizamos a coluna LOJA como a coluna base para o particionamento. A lógica do particionamento está explicada no quadro a seguir:

Quadro 01: Particionamento do tipo RANGE.

COLUNA BASE DO PARTICIONAMENTO	NOME DO PARTICIONAMENTO	FUNÇÃO DE PARTICIONAMENTO
LOJA	Maraponga	Os registros correspondentes aos vendedores das lojas cujos identificadores sejam menores que 6. Lojas 1 a 5 serão gravados na partição da tabela tbVendedor nomeada como maraponga.
	Messejana	Os registros correspondentes aos vendedores das lojas cujos identificadores sejam iguais a 6 e menores que 11. Lojas 6 a 10 serão gravados na partição da tabela tbVendedor nomeada como messejana.
	Bomjardim	Os registros correspondentes aos vendedores das lojas cujos identificadores sejam iguais ou maiores que 11 e menores que 16. Lojas 11 a 15 serão gravados na partição da tabela tbVendedor nomeada como bomjardim.
	Parangaba	Os registros correspondentes aos vendedores das lojas cujos identificadores sejam iguais ou maiores que 16 e menores que 21. Lojas 16 a 20 serão gravados na partição da tabela tbVendedor nomeada como parangaba.
	Aldeota	Os registros correspondentes aos vendedores das lojas cujos identificadores sejam iguais ou maiores que 21 e menores que 26. Lojas 21 a 25 serão gravados na partição da tabela tbVendedor nomeada como aldeota.

Fonte: Elaborado pelo autor.

1.2. Particionamento tipo LIST

O particionamento tipo LIST utiliza na função de particionamento uma lista de **valores inteiros** separados por vírgula que será a base para a comparação com a coluna de referência do particionamento. Vejamos o exemplo para um melhor esclarecimento.

Exemplo do particionamento LIST:

```
CREATE TABLE tbLoja (
codigo INTEGER NOT NULL,
descricao VARCHAR(100),
bairro INT NOT NULL
)
```

```
PARTITION BY LIST (bairro) (
PARTITION regional_I    VALUES IN (2,4,6,8,10),
PARTITION regional_II   VALUES IN (1,3,5,7,9),
PARTITION regional_III  VALUES IN (11,12,13,14,15,16),
PARTITION regional_IV   VALUES IN (21,41,61,81,91),
PARTITION regional_V    VALUES IN (32,42,52,62,72)

);
```

A lógica do particionamento está explicada no quadro a seguir:

Quadro 02: Particionamento do tipo LIST.

COLUNA BASE DO PARTICIONAMENTO	NOME DO PARTICIONAMENTO	FUNÇÃO DE PARTICIONAMENTO
BAIRRO	Regional_I	Os registros correspondentes às lojas cujos identificadores dos bairros estejam na lista :(2,4,6,8,10) serão gravados na partição da tabela tbLoja nomeada como regional_I.
	Regional_II	Os registros correspondentes às lojas cujos identificadores dos bairros estejam na lista :(1,3,5,7,9) serão gravados na partição da tabela tbLoja nomeada como regional_II.
	Regional_III	Os registros correspondentes às lojas cujos identificadores dos bairros estejam na lista :(11,12,13,14,15,16) serão gravados na partição da tabela tbLoja nomeada como regional_III.
	Regional_IV	Os registros correspondentes às lojas cujos identificadores dos bairros estejam na lista :(21,41,61,81,91) serão gravados na partição da tabela tbLoja nomeada como regional_IV.
	Regional_V	Os registros correspondentes às lojas cujos identificadores dos bairros estejam na lista :(32,42,62,72) serão gravados na partição da tabela tbLoja nomeada como regional_V.

Fonte: Elaborado pelo autor.

1.3. Particionamento tipo COLUMN

A característica deste tipo de particionamento é permitir a utilização de **tipos de dados** tais como *date*, *datetime*, *char*, *varchar*, *binary* e *varbinary* na função de particionamento. Adicionalmente, ele permite que se utilize uma ou mais **colunas** da tabela para a definição do particionamento.

A título de uma melhor organização, poderemos dizer que o particionamento tipo COLUMN pode ser subdividido em RANGE COLUMNS e LIST COLUMNS. Se observarmos com cuidado, iremos ver que o tipo RANGE COLUMNS é uma extensão do tipo RANGE, assim como o tipo LIST COLUMNS é uma extensão do tipo LIST. A seguir, eles são descritos com detalhes.

1.3.1. Particionamento RANGE COLUMNS

A escolha entre o tipo de particionamento RANGE ou RANGE COLUMNS depende muito do **ambiente de banco de dados** que se está trabalhando. Podemos ter uma determinada situação em que o tipo RANGE resolve perfeitamente o problema que estamos solucionando. Iremos utilizar somente uma coluna da tabela e a restrição do uso do tipo de dado *INTEGER* para definir a função de particionamento não é um problema.

Então, devemos ter em mente que o tipo de particionamento RANGE COLUMNS tem as seguintes características:

- Não tem a restrição de utilização de uma única coluna, podendo utilizar uma ou várias colunas como referência do particionamento;
- A função de particionamento será definida utilizando-se uma lista de valores da coluna;
- Tem a restrição de só poder ser utilizado os nomes das colunas e não expressões que permitem pegar parte dos valores das colunas, por exemplo, numa coluna do tipo date, utilizar a função MONTH() para trabalhar apenas com o mês da data;
- Pode utilizar colunas com vários tipos de dados como *char*, *varchar*, *date*, *datetime* etc.

Exemplo do particionamento RANGE COLUMNS:

```
CREATE TABLE tbEndereco (
codigo INTEGER NOT NULL,
descricao VARCHAR(100),
estado INT NOT NULL,
cidade INT NOT NULL,
bairro INT NOT NULL,
)
```

```
PARTITION BY RANGE COLUMNS (estado,cidade,bairro) (
PARTITION area_operacional_I   VALUES LESS THAN (5,40,6),
PARTITION area_operacional_II  VALUES LESS THAN (10,114,21),
PARTITION area_operacional_III VALUES LESS THAN (17,500,100),
PARTITION area_operacional_IV  VALUES LESS THAN (23,600,610),
PARTITION area_operacional_V   VALUES LESS THAN
(MAXVALUE,MAXVALUE,MAXVALUE)
);
```

A lógica do particionamento está explicada no quadro a seguir:

Quadro 03: Particionamento do tipo RANGE COLUMNS.

COLUNA BASE DO PARTICIO-NAMENTO	NOME DO PARTICIO-NAMENTO	FUNÇÃO DE PARTICIONAMENTO
ESTADO CIDADE BAIRRO	Área_ope-racional_I	Serão gravados nesta partição os registros que atenderem às seguintes regras da função de particionamento: ESTADO: identificador menor do que 5; CIDADE: identificador menor do que 40; BAIRRO: identificador menor do que 6.
	Área_ope-racional_II	Serão gravados nesta partição os registros que atenderem às seguintes regras da função de particionamento: ESTADO: identificador maior ou igual a 5 e menor que 10; CIDADE: identificador maior ou igual a 40 e menor que 114; BAIRRO: identificador maior ou igual a 6 e menor que 21.
	Área_ope-racional_III	Serão gravados nesta partição os registros que atenderem às seguintes regras da função de particionamento: ESTADO: identificador maior ou igual a 10 e menor que 17; CIDADE: identificador maior ou igual a 114 e menor que 500; BAIRRO: identificador maior ou igual a 21 e menor que 100.
	Área_ope-racional_IV	Serão gravados nesta partição os registros que atenderem às seguintes regras da função de particionamento: ESTADO: identificador maior ou igual a 17 e menor que 23; CIDADE: identificador maior ou igual a 500 e menor que 600; BAIRRO: identificador maior ou igual a 100 e menor que 610
	Área_operacional_V	Serão gravados nesta partição os registros que atenderem às seguintes regras da função de particionamento: ESTADO: (MAXVALUE) os identificadores de estado que não se enquadrarem nos critérios anteriores serão gravados nesta partição; CIDADE: (MAXVALUE) os identificadores de estado que não se enquadrarem nos critérios anteriores serão gravados nesta partição; BAIRRO: (MAXVALUE) os identificadores de estado que não se enquadrarem nos critérios anteriores serão gravados nesta partição.

Fonte: Elaborado pelo autor.

1.3.2. Particionamento LIST COLUMNS

Neste tipo de particionamento, temos uma **lista de valores** para a coluna selecionada como a referência do particionamento. Diferentes tipos de dados podem ser utilizados, como o tipo *integer*, *char*, *varchar*, *date*, *datetime*. Não são permitidas as **expressões ou funções** na cláusula, por exemplo a função MONTH() para selecionar o número inteiro correspondente ao mês.

No exemplo a seguir, a coluna selecionada como referência do particionamento é a coluna ESTADO da tabela tbCliente. Os registros serão direcionados para os particionamentos Centro_Oeste, Nordeste, Norte, Sudeste, Sul conforme estejam nas listas de estados correspondentes a cada uma das partições.

Exemplo do particionamento **LIST COLUMNS**:

```
CREATE TABLE tbCliente
 (codigo INTEGER NOT NULL,
  nome  VARCHAR(100),
  estado VARCHAR(100), NOT NULL)
PARTITION BY LIST COLUMNS (estado) (
PARTITION Centro_Oeste  VALUES IN
('Goiás', 'Mato Grosso','Mato Grosso do Sul','Distrito Federal'),
PARTITION Nordeste  VALUES IN ('Alagoas','Bahia','Ceará','Maranhão','Paraíba','Pernambuco','Piauí', 'Rio Grande do Norte','Sergipe'),
PARTITION Norte  VALUES IN ('Acre','Amapá','Amazonas','Pará','Rondônia','Roraima','Tocantins'),
PARTITION Sudeste  VALUES IN ('Espírito Santo','Minas Gerais','Rio de Janeiro','São Paulo'),
PARTITION Sul  VALUES LESS THAN ('Paraná','Rio Grande do Sul','Santa Catarina'));
```

A lógica do particionamento está explicada no quadro a seguir:

Quadro 04: Particionamento do tipo **LIST COLUMNS**.

COLUNA BASE DO PARTICIO-NAMENTO	NOME DO PARTICIO-NAMENTO	FUNÇÃO DE PARTICIONAMENTO
ESTADO	Centro_ Oeste	Caso o valor da coluna estado da tabela tbCliente estiver na lista: ('Goiás', 'Mato Grosso', 'Mato Grosso do Sul', 'Distrito Federal'), o registro será gravado na partição da tabela cujo nome é Centro_Oeste.
	Nordeste	Caso o valor da coluna estado da tabela tbCliente estiver na lista: ('Alagoas','Bahia','Ceará','Maranhão','Paraíba','Pernambuco','Piauí', 'Rio Grande do Norte', 'Sergipe'), o registro será gravado na partição da tabela cujo nome é Nordeste.
	Norte	Caso o valor da coluna estado da tabela tbCliente estiver na lista: ('Acre', 'Amapá', 'Amazonas', 'Pará', 'Rondônia', 'Roraima', 'Tocantins'), o registro será gravado na partição da tabela cujo nome é Norte.
	Sudeste	Caso o valor da coluna estado da tabela tbCliente estiver na lista: ('Espírito Santo', 'Minas Gerais', 'Rio de Janeiro', 'São Paulo'), o registro será gravado na partição da tabela cujo nome é Sudeste.
	Sul	Caso o valor da coluna estado da tabela tbCliente estiver na lista: ('Paraná', 'Rio Grande do Sul', 'Santa Catarina'), o registro será gravado na partição da tabela cujo nome é Sul.

Fonte: Elaborado pelo autor.

1.4. Particionamento do tipo HASH

Este tipo de particionamento é bastante simples, ele exige que a **coluna referência** para o particionamento seja do **tipo inteiro**, o número de partições também deve ser informado, e caso não seja, ele coloca como padrão uma única partição. O **MySQL** vai nomear essas partições automaticamente, já que os nomes das partições não são informados no comando de criação e particionamento da tabela. Também será selecionado de forma automática a melhor partição para o registro que está sendo inserido de acordo com a coluna referência do particionamento e do número de partições definidas. Esse é o tipo de particionamento recomendado quando desejarmos ter uma **distribuição uniforme dos registros** dentro das partições. O exemplo a seguir ilustra este tipo de particionamento.

Exemplo do particionamento **HASH**:

```
CREATE TABLE tbVendedor (
matricula INTEGER NOT NULL,
nome VARCHAR(100),
loja INT NOT NULL
)
PARTITION BY HASH (loja)
PARTITIONS 5;
```

No exemplo anterior, a coluna da tabela tbVendedor selecionada para ser a referência do particionamento é a coluna loja, que é do tipo inteiro. O número de partições definidas foram CINCO. Então, a cada registro inserido, o próprio MySQL, automaticamente, define em que partição ele deverá ficar, fazendo um balanceamento. As partições também são nomeadas automaticamente.

1.5. Particionamento do tipo KEY

O particionamento do tipo KEY é mais simples do que o particionamento do tipo HASH. No tipo KEY nem mesmo a coluna referência do particionamento precisa ser informada pelo usuário, basta que a tabela tenha uma **chave primária** ou uma coluna da tabela com restrição do **tipo única** (*unique*). O MySQL identifica e a utiliza como referência do particionamento. O número de partições deve ser informado e se não for o padrão de uma partição será utilizado.

Exemplo do particionamento **KEY**:

```
CREATE TABLE tbVendedor (
matricula INTEGER NOT NULL PRIMARY KEY,
nome VARCHAR(100),
loja INT NOT NULL
)
PARTITION BY KEY
PARTITIONS 5);
```

No exemplo acima, notamos que não foi informada uma coluna da tabela tbVendedor para ser a referência do particionamento, como nos demais tipos de particionamentos apresentados, mas notamos que existe a coluna “matricula” que é uma chave primária, e no tipo de particionamento KEY automaticamente o MySQL irá selecioná-la como referência. O número de partições definidas foram CINCO. Então, a cada registro inserido, o próprio MySQL, automaticamente, tomando como base a chave primária, define em que partição ele deverá ser gravado. O MySQL também nomeia as partições automaticamente.

2. Indexação

Antes de criarmos **índices** no **MySQL**, faremos uma introdução básica sobre como os registros estão gravados dentro das tabelas do nosso banco de dados. Sob o ponto de vista de sua organização, os registros podem estar **desordenados** e **ordenados**.

Estando os registros na tabela de **forma desordenada**, significa dizer que eles são inseridos independentes da preocupação de os colocar em uma determinada ordem, por exemplo, no cadastro de clientes, não tem como se cadastrar, inicialmente, todos os clientes cujo nome comece com a letra “A”, depois com a letra “B” etc. Os clientes são cadastrados conforme as suas compras em uma determinada loja, sem a preocupação de gravá-los na tabela organizados alfabeticamente por seus nomes. Assim também, o cadastramento dos alunos da UniATENEU é feito conforme os alunos vão sendo matriculados nos cursos, sem atender a uma ordem, seja por nome, data de matrícula etc.

No caso da inclusão dos registros de **forma ordenada**, uma das grandes vantagens é a **pesquisa** muito mais eficiente, uma vez que, por exemplo, estando os registros organizados alfabeticamente por nome do cliente, posso fazer uma pesquisa muito mais rápida. Se os registros estiverem desorganizados, terei que ler os registros um a um para identificar o nome do cliente que procuro, ao posso que se estiverem organizados, já posso ir diretamente nos registros que correspondem à letra inicial do nome do cliente, otimizando em muito o tempo da pesquisa.

Buscando uma definição de **índices** na literatura de banco de dados, selecionamos a de Abraham Silberschatz (2006, p. 321), reproduzida abaixo:

> Os índices no sistema de banco de dados desempenham o mesmo papel dos índices de livro nas bibliotecas. Por exemplo, para apanhar um registro de conta dado o número da conta, o sistema de banco de dados pesquisaria um índice para descobrir em que bloco do disco o registro correspondente reside, e depois apanharia o bloco do disco, para obter o registro de conta.
>
> Manter uma lista classificada de números de conta não funcionaria bem em muitos sistemas grandes de banco de dados, com milhões de contas, pois o próprio índice seria muito grande, além disso, embora manter um índice classificado reduza o tempo de busca, encontrar uma conta ainda pode ser um tanto demorado. Em vez disso, técnicas de indexação mais sofisticadas podem ser utilizadas.

Esta definição compara os **índices** de bancos de dados a índices de livros. Vamos supor que precisemos fazer uma pesquisa de determinado assunto num livro, imaginemos que o livro tem 500 páginas, e precisamos encontrar um determinado assunto, sem o índice do livro, não saberei diretamente a qual página ir para encontrar o assunto que me interessa, então, todas as vezes que necessitássemos fazer uma pesquisa, teríamos que percorrer todo o livro até encontrar as páginas referentes ao assunto desejado.

Quando trabalhamos com uma tabela sem índices e executamos uma busca em seus registros, no caso utilizando o comando SELECT, estaremos executando uma ação chamada **TABLE SCAN**, que consiste em percorrer a tabela partindo do primeiro registro, e ir testando cada registro, se é aquele que procuramos. Como exemplo, precisamos fazer uma busca de um cliente por seu código de cadastro em uma tabela não indexada, seja o código que estamos buscando igual a “3”, para esta busca, o SGBD inicia fazendo a comparação do código que estamos pesquisando com o primeiro registro da tabela, se forem iguais, ele mostra os registros segundo as colunas definidas no comando SELECT, senão, salta para o próximo registro e faz o mesmo teste. Repete o procedimento até que o registro tenha sido encontrado, ou tenha atingido o fim do arquivo.

No caso da tabela indexada, o SGDB a organiza pela **coluna referência do índice** e as buscas serão feitas seguindo um algoritmo chamado **busca binária**, no qual a tabela sofre sucessivas divisões durante o processo de busca, até que o registro seja encontrado. No início da busca, a tabela é dividida na metade e o SGBD verifica se o registro procurado está na primeira ou na segunda metade, se estiver na primeira metade, por exemplo, a segunda metade é descartada, se estiver na segunda, a primeira é descartada. Aqui, já notamos o ganho em termos de **tempo de busca**, pois a metade da tabela é descartada. Este processo é repetido inúmeras vezes até que o registro seja encontrado.

2.1. Classificação dos índices conforme sua estrutura

- **Índice Agrupado (*Clustered Index*)**

Este tipo de índice define uma **ordem física** de **gravação dos registros**, geralmente, baseada na chave primária da tabela, o que resulta na restrição de termos somente um tipo de índice agrupado por tabela. Os registros são classificados de acordo com a coluna que possui o índice, como exemplo, a nossa tabela **tbcliente**, que tem a chave primária na coluna código, terá seus registros ordenados fisicamente tomando como base esta coluna.

- **Índice Não Agrupado (*Non Clustered Index*)**

Neste tipo de índice não há a alteração da forma como os dados são armazenados, o objeto índice é criado com as **colunas referenciadas na indexação**, com ponteiros para os registros da tabela original. Uma tabela pode ter mais de um índice deste tipo, como exemplo, a nossa tabela **tbestado**, poderemos criar um índice não agrupado na coluna referente ao nome do estado e outro índice referente à coluna sigla do estado.

2.2. Criação de índices no MySQL

Para a criação dos índices, utilizar o comando CREATE INDEX, conforme mostrado a seguir. Como exemplo, foram criados 3 índices na tabela tbestado, o primeiro deles tendo como referência a coluna “codigo”, que é a chave primária da tabela, será um ÍNDICE AGRUPADO (CLUSTERED INDEX), e os demais serão do tipo ÍNDICE NÃO AGRUPADO (NON CLUSTERED INDEX):

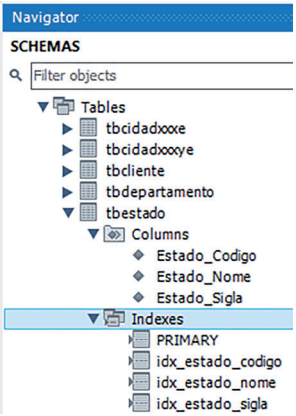
```
CREATE INDEX idx_estado_codigo ON tbestado(Estado_Codigo);

CREATE INDEX idx_estado_nome   ON tbestado(Estado_Nome);

CREATE INDEX idx_estado_sigla  ON tbestado(Estado_Sigla);
```

Havendo a necessidade de **exclusão** dos índices, utilizar o comando a seguir:

```
DROP INDEX idx_estado_sigla  ON tbestado;
```

Para a verificação dos índices no MySQL Workbench, basta abrir a sessão *Table > tbestado > Indexes*, conforme mostrado na figura a seguir.

Figura 01: MySQL Workbench – ÍNDICES.



Fonte: MySQL Workbench.

3. Objetos avançados de banco de dados

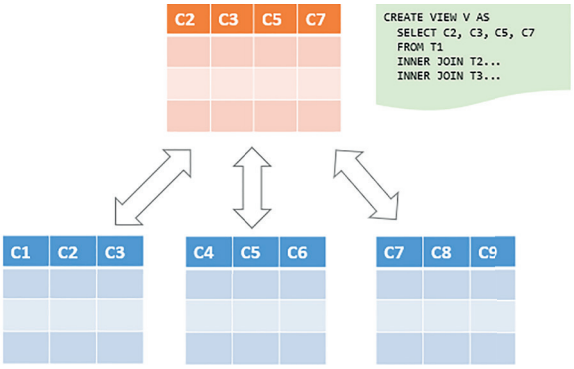
3.1. Visões (*Views*)

Uma visão é um **objeto** de banco de dados que é criado baseado em um **comando SELECT**, portanto, não armazena nenhum dado, pode ser formado por uma ou mais tabelas, que são chamadas de **tabelas bases da visão**. As visões podem ser atualizáveis, sendo possível inserir, alterar ou excluir dados. Neste caso, os comandos são dados tendo como base a visão, mas vai modificar as tabelas bases que são as fontes dos dados das visões. As visões também são conhecidas como “tabelas virtuais” e podem compor o comando SELECT com outras visões ou tabelas.

Como mostrado na figura a seguir, a visão '**V**' é formada pelas seguintes colunas:

- Coluna C2 e C3, que vem da tabela base 'T1';
- Coluna C5, que vem da tabela base 'T2';
- Coluna C7, que vem da tabela base 'T3'.

Figura 02: Formação de uma VISÃO.



Fonte: https://bit.ly/33J8XWm.

- Aplicação das visões

➢ Como podem conter colunas oriundas de várias tabelas base, que se relacionam de forma complexa, elas fornecem uma ferramenta de simplificação, pois ao invés de utilizar um comando SELECT complexo, podemos criar uma visão baseada neste comando SELECT e termos uma única "tabela virtual", facilitando assim o acesso aos dados;

➢ Fornecem uma camada extra de segurança, pois somente determinadas colunas são filtradas na composição da visão, assim como também poderemos utilizar a cláusula WHERE para filtrar registros;

➢ Permitem a utilização de comando de inserção, atualização e exclusão de dados, que afetará as tabelas base. A única restrição feita neste caso é que as visões, obrigatoriamente, devem conter as colunas das tabelas base que não aceitam valores NULOS, ou seja, as colunas com característica NOT NULL.

- **Criando visões**

Para a criação das visões, utilizamos o comando de definição de dados (*DATA DEFINITION LANGUAGE* – DDL) **CREATE VIEW**. A seguir, colocaremos exemplos de criação de visões:

I. Criação de uma visão simples contendo todas as colunas de uma única tabela, devemos observar a utilização do **"*"** (asterisco), significando que todas as colunas farão parte da visão:

```
Create view vis_cliente as

select * from tbcliente;
```

II. Criação de uma visão contendo somente algumas colunas de uma única tabela, no caso, selecionamos somente a coluna Estado_Sigla da tabela tbestado:

```
Create view vis_estado_sigla as

select Estado_Sigla as Sigla from tbestado;
```

Observação: observamos neste comando select a utilização da cláusula "**as**" que permite que mudemos o nome da coluna a ser mostrada. No exemplo, ao invés de mostrar o nome "Estado_Sigla" como título da coluna será mostrado "Sigla". Portanto, criamos um "apelido" para a coluna (alias). Na literatura de banco de dados, encontramos esta funcionalidade definida como **alias de coluna**. Isto resulta que ao executarmos um SELECT na visão, teremos como retorno o apelido da coluna e não mais o seu nome, como título da mesma.

III. Criação de uma visão contendo colunas vindas de várias tabelas base, utilizando as junções de tabelas:

```
Create view vis_cliente_completo as

select a.Cliente_Nome as Cliente,

b.Estado_Nome as Estado,

c.Cidade_Nome as Cidade,

d.Bairro_Nome as Bairro,

e.Logradouro_Nome as Logradouro
```

```
from tbestado as a
left join tbestado b on a.Cliente_Estado=b.Estado_Codigo
left join tbcidade c on a.Cliente_Cidade=c.Cidade_Codigo
left join tbbairro  d on a.Cliente_Bairro=d.Bairro_Codigo
left join tblogradouro e on a.Cliente_Logradouro=e.Logradouro_codigo;
```

Observação: neste exemplo, além da utilização do **alias de colunas**, estamos utilizando também o **alias de tabela**, a tabela tbcliente passou a ser referenciada com a letra "**a**", a tbestado com a letra "**b**" etc. Observamos também que a cláusula "**as**" não é obrigatória, podendo ser utilizada ou não.

IV. Criação de uma visão contendo todas as colunas necessárias para a geração de um relatório de vendas:

```
Create view vis_relatorio_vendas as
select
Venda_Codigo,Venda_Data,Venda_Quantidade,tbcliente.Cliente_
Nome,
tbvendedor.Vendedor_Nome,tbproduto.Produto_Nome,tbloja.Loja_
Nome,tbdepartamento.Departamento_Nome
from tbvenda
LEFT JOIN tbcliente
on tbvenda.Venda_cliente=tbcliente.Cliente_Codigo
LEFT JOIN tbvendedor
on tbvenda.Venda_Vendedor=tbvendedor.Vendedor_Matricula
LEFT JOIN tbproduto
n tbvenda.Venda_produto=tbproduto.Produto_Codigo
LEFT JOIN tbloja
on tbvenda.Venda_Loja=tbloja.Loja_Codigo
LEFT JOIN tbdepartamento
on      tbvenda.Venda_Departamento=tbdepartamento.Departamento_
Codigo;
```

Observação: neste exemplo, nenhum **alias de colunas ou de tabelas**, foi utilizado. A referência às tabelas é feita diretamente, utilizando o nome da tabela, um ponto e o nome da coluna.

- **Utilizando as visões**

Para se utilizar as visões, o procedimento é igual à utilização do comando SELECT em tabelas, como mostrado a seguir:

```
select * from vis_cliente;

select Sigla from vis_estado_sigla

select * from vis_cliente_completo

select Cliente,Estado,Cidade,Bairro,Logradouro from vis_relatorio_
vendas
```

- **Alterando as visões**

Para a alteração das visões, utilizamos o comando de definição de dados (*DATA DEFINITION LANGUAGE* – DDL) **ALTER VIEW**. A seguir, um exemplo modificando a **vis_estado_sigla** para a colocação das colunas Estado_Codigo e Estado_Nome, mantendo a coluna Estado_Sigla:

```
alter view vis_estado_sigla as

select Estado_Codigo,Estado_Nome,Estado_Sigla as Sigla from
tbestado;
```

- **Excluindo as visões**

A exclusão das visões é feita através do comando de definição de dados (*DATA DEFINITION LANGUAGE* – DDL) DROP VIEW. No exemplo a seguir, a visão **vis_estado_sigla** é excluída:

```
drop view vis_estado_sigla
```

3.2. Funções (*Functions*)

O objeto de banco de dados "função" (*function*) é um tipo de **objeto programável** que se utiliza para retornar valores, é geralmente utilizada para fazer cálculos, ou aplicar regras de negócios sobre um determinado dado ou conjunto de dados. O SGBD MySQL já traz internamente muitas funções, mas também permite que o usuário crie as suas próprias, para isto, precisaremos conhecer a linguagem SQL e utilizá-la para construir as funções.

Como exemplo de funções internas do MySQL (*buit-in functions*), poderemos verificar algumas no quadro:

Quadro 05: Algumas funções internas do MySQL.

FUNÇÃO	EXEMPLO	OBSERVAÇÃO
NOW()	SELECT **NOW();**	Função que retorna a data e hora atuais.
CHAR_LENGTH();	SELECT **CHAR_LENGTH**('UNIATENEU');	Esta função retornará o número de caracteres da palavra UNIATENEU, ou seja, retornará o número 10.
CONCAT()	SELECT CONCAT('BANCO','DE','DADOS');	Será retornada a cadeia de caracteres: "BANCODEDADOS", pois a função fará a concatenação da palavra banco+de+dados.
LCASE()	SELECT LCASE('BANCO DE DADOS');	A função retornará a palavra 'BANCO DE DADOS' em letras minúsculas.
LCASE()	SELECT UCASE('banco de dados');	O resultado desta função será a palavra 'banco de dados' em letras maiúsculas.
LEFT()	SELECT LEFT('banco de dados',5);	Será mostrada a palavra 'banco', pois a função retira da cadeia de caracteres 'banco de dados' os 5 caracteres mais à esquerda.
RIGHT()	SELECT RIGHT('banco de dados',5);	Neste caso, a função retornará os 5 caracteres mais à direita da cadeia de caracteres "banco de dados", ou seja, a palavra "dados".

Fonte: https://bit.ly/39UMaL9.

Observemos no quadro das funções que existem algumas delas que não exigem que digitemos nada entre os parênteses, como a função NOW(), outras, como a função CHAR_LENGHT(), já exigem que digitemos uma cadeia de caracteres. As funções LEFT() e RIGHT() necessitam que digitemos a cadeia de caracteres e um número que significa quantos caracteres desejamos que sejam retirados da cadeia de caracteres informados. No caso da função CONCAT(), podem ser digitados vários caracteres ou cadeia de caracteres.

Estes valores que são digitados dentro dos parênteses na chamada de uma função são chamados de **parâmetros**, e sobre eles, no caso das funções que exigem parâmetros, a função irá trabalhar e retornar um valor resultado do processamento dos parâmetros.

- **Criando funções (*Functions*)**

Existem inúmeras **função internas** do MySQL para os mais diversos usos, mas há também a possibilidade de construirmos **funções personalizadas**, por várias razões, dentre elas poderemos citar:

- Em desenvolvimento de sistemas podem surgir situações nas quais iremos precisar utilizar uma determinada rotina SQL em diversos módulos de um sistema ou mesmo em sistemas diferentes, então, construindo uma função, poderemos ter o reaproveitamento de código. Por exemplo, em vários sistemas precisamos validar os CPFs dos clientes, então, construiremos uma função que recebe como parâmetro o CPF e retorna se ele é válido ou não;

- Se tivermos uma função única que atenda a diversos módulos de um sistema, ou mesmo a diversos sistemas, na necessidade de uma alteração, ela será feita somente em um local, evitando que no caso do mesmo trecho de código está espalhado em diversos locais, haja a necessidade de alterações deste código em várias partes do sistema, o que torna o procedimento de alteração mais propenso a erros;

- Podem surgir situações em sistemas muito grandes que ficará mais fácil para a manutenção do código, separá-los em partes permitindo que a lógica fique mais fácil de entender;

- Para separar os códigos SQL em blocos que possam ser logicamente compreendidos de forma isolada.

Para a criação das funções definidas pelo usuário, utilizamos o comando de definição de dados (*DATA DEFINITION LANGUAGE* – DDL) **CREATE FUNCTION.**

No quadro de exemplos a seguir, iremos criar algumas funções:

Quadro 06: Funções definidas pelo usuário.

FUNÇÃO	UTILIZAÇÃO	OBSERVAÇÃO
CREATE FUNCTION saudacao () RETURNS CHAR(50) DETERMINISTIC RETURN 'Bom dia';	SELECT saudacao()	A função criada não tem parâmetros de entrada, ela retorna a cadeia de caracteres 'Bom dia'.
CREATE FUNCTION bom_dia (nome CHAR(20)) RETURNS CHAR(50) DETERMINISTIC RETURN CONCAT('Bom dia, ',nome,'!');	SELECT bom_dia('José');	Neste exemplo, a função pede um parâmetro que será um nome qualquer. Como resultado da função é retornado a cadeia de caracteres "Bom dia", seguida do nome digitado e do sinal de exclamação.

DELIMITER $$
CREATE FUNCTION Classifica_cliente(quantidade INT)
RETURNS VARCHAR(20) DETERMINISTIC
BEGIN
DECLARE nivel_cliente VARCHAR(20);
 	
IF quantidade = 1 THEN
SET nivel_cliente = 'JUNIOR';
ELSEIF (quantidade >= 2 AND quantidade <= 4) THEN
SET nivel_cliente = 'PLENO';
ELSEIF quantidade >= 10 THEN
SET nivel_cliente = 'SÊNIOR';
END IF;
RETURN (nivel_cliente);
END$$
DELIMITER ;
	SELECT Classifica_cliente(1)
	
	SELECT Classifica_cliente (3)
	
	SELECT Classifica_cliente (11)
		Esta função é um pouco mais complexa que as demais, ela solicita um parâmetro de entrada, que deve ser um inteiro, e dentro do corpo da função ele é testado pelo comando IF... ELSEIF. Se o valor digitado for igual a "1" a função retornará a palavra "JUNIOR". Caso o valor esteja entre "2" e "4" a função retornará "PLENO" e por último se o valor for maior ou igual a "10" ele retornará o valor "SÊNIOR".

Fonte: Elaborado pelo autor.

Poderemos utilizar qualquer função dentro de um comando SELECT, por exemplo, vamos utilizar a função Classifica_Cliente() dentro de um comando SELECT na tabela tbvenda. Esta tabela tem a coluna Venda_Quantidade que mostra a quantidade do produto vendido a um cliente, portanto, poderemos fazer uma consulta nas informações de vendas, classificando os clientes através da utilização da função Classifica_cliente() usando as duas opções abaixo:

- **Utilizando um comando SELECT diretamente**

Uma forma de obter os dados para o relatório de vendas seria construir o comando SELECT inteiro, incluindo a chamada à função **Classifica_cliente()**, como mostrado a seguir:

```
select tbvenda.Venda_Codigo,tbvenda.Venda_Data,tbvenda.Venda_Quantidade,

Classifica_cliente(tbvenda.Venda_Quantidade) as Nivel_cliente,

tbcliente.Cliente_Nome, tbvendedor.Vendedor_Nome,

tbproduto.Produto_Nome,tbloja.Loja_Nome,tbdepartamento.Departamento_Nome

from tbvenda

LEFT JOIN tbcliente

on tbvenda.Venda_cliente=tbcliente.Cliente_Codigo

LEFT JOIN tbvendedor

on tbvenda.Venda_Vendedor=tbvendedor.Vendedor_Matricula

LEFT JOIN tbproduto

n tbvenda.Venda_produto=tbproduto.Produto_Codigo

LEFT JOIN tbloja

on tbvenda.Venda_Loja=tbloja.Loja_Codigo

LEFT JOIN tbdepartamento

on tbvenda.Venda_Departamento=tbdepartamento.Departamento_Codigo;
```

Neste comando SELECT, utilizamos a chamada da função **Classifica_Cliente()** na segunda linha, passando como parâmetro a coluna Venda_Quantidade, que será tratada pela função e os clientes serão classificados em JUNIOR, PLENO ou SÊNIOR. Outro detalhe a observar é que colocamos um “apelido” na chamada da função, o cabeçalho da coluna que conterá as classificações será nomeado NIVEL_CLIENTE.

- **Utilizando a visão VIS_RELATORIO_VENDAS**

Outra opção é a utilização da visão vis_relatorio_venda, que foi construída com o comando SELECT anterior. Incluiremos a chamada à função Classifica_cliente(), o que torna o comando bem mais simples e com o mesmo resultado, conforme mostrado a seguir:

```
select Venda_Codigo,Venda_Data,Venda_Quantidade,
Classifica_cliente(Venda_Quantidade) as Nivel_cliente,
tbcliente.Cliente_Nome,
tbvendedor.Vendedor_Nome,tbproduto.Produto_Nome,tbloja.Loja_
Nome,tbdepartamento.Departamento_Nome
from vis_relatorio_venda
```

3.3. Gatilhos (*Triggers*)

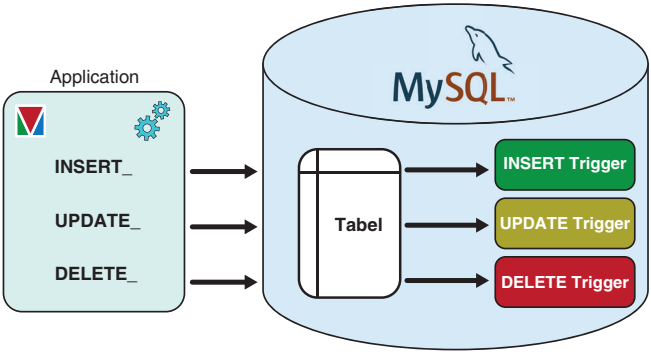
Um gatilho, assim como as funções, são objetos de banco de dados programáveis, mas tem uma característica peculiar que é de estarem **associados** a uma **tabela**. Quando ocorrem eventos de inclusão (insert), atualização (*update*) ou exclusão (*delete*), o gatilho é ativado. Ele pode ser definido para agir em um destes eventos separadamente, ou em todos. Como exemplo, podemos definir que um gatilho dispare somente quando ocorrer um comando de inserção, ou podemos definir também que ele seja disparado quando ocorrer inserção, alteração ou exclusão.

Além desta peculiaridade de estarem associados a tabelas, ainda podemos citar as seguintes características:

- Diferentemente das funções, os gatilhos não retornam resultados;
- Os gatilhos não aceitam a passagem de parâmetros de entrada;
- Os gatilhos não podem ser disparados diretamente, eles são invocados automaticamente quando ocorrem os eventos de inserção, alteração e exclusão.

A figura a seguir ilustra o procedimento de ação dos gatilhos. Em uma tabela do banco de dados existem gatilhos de inclusão, atualização e exclusão. Uma determinada aplicação faz inclusões, alterações e exclusões, tendo como consequência o disparo dos gatilhos a cada ocorrência destas ações. A ação a ser executada pelos gatilhos será escrita em linguagem SQL, como veremos a seguir.

Figura 03: Funcionamento dos gatilhos (*triggers*).



Fonte: https://bit.ly/3oncvVP.

- **Criando gatilhos**

Para ilustrar a utilização dos gatilhos, vamos construir uma estrutura para **gravar o histórico** das inclusões, alterações e exclusões que serão feitas na tabela **tbestado**. Todas as vezes que houver uma inclusão, alteração ou exclusão de um registro na tabela **tbestado**, o registro que está sofrendo a ação será transferido para uma outra tabela chamada **tbestado_historico**, que iremos criar e que terá as mesmas colunas da tabela tbestado, acrescida de duas colunas: uma que irá gravar a **hora da ocorrência** e outra que indicará o **tipo de operação** ocorrida. Portanto, a estrutura desta tabela será a seguinte:

```
CREATE TABLE `tbestado_historico` (
  `Estado_Codigo` INT NOT NULL,
  `Estado_Nome` VARCHAR(100) NOT NULL,
  `Estado_Sigla` VARCHAR(2) NOT NULL,
  `Dthr_Modificacao` DATETIME NOT NULL,
  `Acao`  CHAR(1) NOT NULL)
```

O funcionamento do gatilho será o seguinte, havendo uma inserção na tabela tbestado, o registro será também enviado para a tabela tbestado_historico, acrescentando-se a **data** e **hora** do evento e qual foi a **ação**. Como a ação é de inserção, será gravado na coluna ação a letra "**I**". No caso do evento de alteração, um registro na tabela tbestado está sendo modificado, então, este registro, antes da alteração feita, deve ser enviado para a tabela tbestado_historico, tendo a coluna acao gravada com a letra "**A**" e também a data e hora da modificação. Na exclusão, os registros que estão sendo excluídos devem ser gravados na tabela tbestado_historico a data e hora do evento deve ser gravado na coluna Dthr_Modificacao e a coluna acao será gravada com a letra "E".

Quadro 07: Implementação dos gatilhos.

AÇÃO	GATILHO
INSERÇÃO	CREATE TRIGGER trg_auditoria_INS AFTER INSERT ON tbestado FOR EACH ROW INSERT INTO tbestado_historico(Estado_Codigo,Estado_Nome, Estado_Sigla,Dthr_Modificacao,Acao) values (NEW.Estado_Codigo,NEW.Estado_Nome,NEW.Estado_Sigla,NOW(),'I');
ALTERAÇÃO	CREATE TRIGGER trg_auditoria_ALT AFTER UPDATE ON tbestado FOR EACH ROW INSERT INTO tbestado_historico(Estado_Codigo,Estado_Nome, Estado_Sigla,Dthr_Modificacao,Acao) values (OLD.Estado_Codigo,OLD.Estado_Nome,OLD.Estado_Sigla, NOW(),'A');
EXCLUSÃO	CREATE TRIGGER trg_auditoria_EXC AFTER UPDATE ON tbestado FOR EACH ROW INSERT INTO tbestado_historico(Estado_Codigo,Estado_Nome, Estado_Sigla,Dthr_Modificacao,Acao) values (OLD.Estado_Codigo,OLD.Estado_Nome,OLD.Estado_Sigla, NOW(),'E');

Fonte: Elaborado pelo autor.

Observando o comando de **criação dos gatilhos**, verificamos que existe uma cláusula **AFTER**, isto significa que ele será disparado APÓS a execução da ação na tabela principal, ou seja, no caso do nosso exemplo, antes de incluir, alterar ou excluir, o registro será inserido na tabela tbestado_historico. Caso desejemos que o gatilho seja disparado antes das ações de inclusão, alteração ou exclusão e posteriormente o registro seja incluído, alterado ou excluído, utilizaremos a cláusula **BEFORE** (antes do evento) ao invés da cláusula AFTER.

Outra observação que devemos levar em conta são as palavras **NEW** e **OLD** que aparecem no comando de criação do gatilho, estas palavras significam o seguinte:

- A palavra NEW significa as informações do novo registro, e a palavra OLD significa as informações do registro antigo;
- No gatilho disparado com o comando INSERT somente podemos utilizar a palavra NEW, pois é uma inserção e o registro será um novo ainda não existente na tabela;
- No gatilho disparado com o comando UPDATE, as duas palavras podem ser utilizadas dependendo do problema que se está querendo resolver, pois como se trata de uma atualização, teremos duas situações: o registro no seu estado anterior, ou seja, o registro já existente e, portanto, a palavra OLD fará referência a esta situação, e o registro no seu estado atual que virá com as atualizações e a palavra NEW fará referência;
- No gatilho disparado com o comando DELETE só poderemos utilizar a palavra OLD, uma vez que não existe registro novo, somente o já existente;

Em todos os gatilhos, utilizamos a função **NOW**(), que é uma função interna do MySQL e cujo objetivo do seu uso no comando INSERT é de gerar a **data** e a **hora atual**, que no nosso exemplo, deve ser gravada na coluna Dthr_Modificacao.

Para **testar** a estrutura de gatilhos, deveremos fazer as seguintes ações:

I. INSERÇÃO

- Inserir um registro na tabela tbestado, e fazer a verificação nesta tabela e na tabela tbestado_historico, como mostrado a seguir:

```
INSERT INTO tbestado(Estado_Nome,Estado_Sigla) values ('AMAZONAS','AM');

SELECT * FROM tbestado;

SELECT * FROM tbestado_historico;
```

II. ALTERAÇÃO

- Alterar um registro na tabela tbestado, e fazer a verificação nesta tabela e na tabela tbestado_historico, como mostrado a seguir:

```
ALTER TABLE tbestado SET Estado_Sigla='AX' WHERE Estado_Codigo=7;

SELECT * FROM tbestado;

SELECT * FROM tbestado_historico;
```

III. EXCLUSÃO

- Excluir um registro da tabela tbestado, e fazer a verificação nesta tabela e na tabela tbestado_historico, como mostrado a seguir:

```
DELETE FROM  tbestado WHERE Estado_Codigo=7;

SELECT * FROM tbestado;

SELECT * FROM tbestado_historico;
```

- **Excluindo gatilhos**

Para a exclusão dos gatilhos, basta utilizar o comando a seguir:

```
DROP TRIGGER  <nome do gatilho>;
```

No caso dos nossos exemplos, para excluir os gatilhos, utilize os seguintes comandos:

```
DROP TRIGGER trg_auditoria_INS;

DROP TRIGGER trg_auditoria_ALT;

DROP TRIGGER trg_auditoria_EXC;
```

4. Procedimentos armazenados (*stored procedures*)

Vimos anteriormente que as visões eram uma forma de guardar **comandos SELECT complexos**, aqui, veremos que os **procedimentos armazenados** também poderão ser utilizados para a mesma função.

Antes de iniciarmos a implementação dos procedimentos armazenados utilizando a linguagem DDL, iremos fazer algumas observações sobre eles. Podemos dizer que é um conjunto de comando que são executados de uma única vez, aceitando parâmetros de entrada, assim como uma função. Dentro de um procedimento armazenado, poderemos ter comandos de manipulação de dados (*Data Manipulation Language* - DML), que são os que encontramos mais frequentemente, no entanto, também poderemos ter procedimentos armazenados constituídos de comandos de definição de dados (*Data Definition Language* - DDL). Umas das grandes vantagens da utilização deste objeto de banco de dados é que ele encapsula a lógica de negócio escrita em **linguagem SQL** para a sua **reutilização**, desta forma haverá uma economia de recursos de memória e tempo de CPU, uma vez que o SGBD otimiza as consultas que compõem um procedimento armazenado, contribuindo para que o banco de dados tenha um alto desempenho.

Os procedimentos armazenados também encapsulam a complexidade da lógica escrita em linguagem SQL. Permitem a **centralização** de comandos SELECT, que ao invés de serem escritos em várias partes da aplicação, serão escritos uma única vez em um bloco de comandos no banco de dados. Por sua vez, o aplicativo fará as chamadas deste procedimento armazenado nos locais da aplicação onde eles são necessários, passando o nome do procedimento e os parâmetros, facilitando inclusive a manutenção do código, uma vez que ele se encontra em apenas um local no banco de dados.

- **Vantagens dos procedimentos armazenados**

I. Contribuir para o aumento da segurança do banco de dados:

Os administradores do SGBD implantarão políticas de segurança no diz respeito a acessos de usuários ou aplicação aos objetos de banco de dados. Neste caso, podem fornecer privilégios de execução dos procedimentos armazenados às aplicações, sem necessariamente fornecer acessos aos objetos que o compõem.

II. Centralizar a lógica de negócios no banco de dados:

A lógica do negócio pode ser implementada no banco de dados através da utilização de procedimentos armazenados. As aplicações para fazer acessos a esta lógica, executam o procedimento armazenado passando parâmetros, com isto evita a proliferação de comandos SELECT contendo a lógica de negócios em várias partes de um aplicativo ou em aplicativos diferentes, facilitando inclusive a manutenção.

III. Redução do tráfego de rede:

Como já foi mencionado, ao invés de um aplicativo executar os comandos SELECT que traduzem a lógica do negócio em várias partes diferentes, o procedimento é executado passando o nome dele e seus parâmetros. Com isto, o tráfego de rede entre os aplicativos e o SGBD é grandemente reduzido.

Como toda utilização de recursos que visam aumentar o desempenho em sistema de banco de dados, no caso de procedimentos armazenados, também teremos que fazer um bom planejamento, pois teremos que levar em consideração que ao utilizá-lo, estaremos comprometendo um recurso caro que é a **memória**. A criação de muitos procedimentos armazenados significa o consumo de uma maior parcela de memória. Outro fator é que dependendo da complexidade dos procedimentos armazenados, se tiverem uma lógica de negócios que exija um processamento pesado, teremos também um maior **consumo de CPU**.

Para exemplificarmos a utilização dos procedimentos armazenados, poderemos iniciar utilizando a visão **vis_estado_sigla**, que foi construída quando estudamos as visões. O código de construção desta visão está reproduzido a seguir:

```
create view vis_estado_sigla as

select Estado_Codigo,Estado_Nome,Estado_Sigla as Sigla from tbestado;
```

Iremos aproveitar o comando SELECT que gera esta visão e criar um procedimento armazenado sem parâmetros de entrada e executá-lo. Diferentemente da utilização da visão que utiliza um comando SELECT para lá retornar os dados, no procedimento armazenado, fazemos a sua execução através do comando CALL. Abaixo está o comando para construir o procedimento armazenado que daremos o nome de pa_estado_sigla:

```
DELIMITER $$

CREATE PROCEDURE pa_estado_sigla ()

BEGIN

select Estado_Codigo,Estado_Nome,Estado_Sigla as Sigla from tbestado;

END $$

DELIMITER ;
```

Tendo sido criado o procedimento armazenado com sucesso, poderemos executá-lo conforme o comando a seguir:

```
CALL pa_estado_sigla();
```

A execução deste procedimento armazenado terá os mesmos resultados do comando SELECT:

```
select Estado_Codigo,Estado_Nome,Estado_Sigla as Sigla from tbestado;
```

O mesmo resultado será obtido se executarmos um comando SELECT na visão **vis_estado_sigla**:

```
select Estado_Codigo,Estado_Nome,Estado_Sigla as Sigla from vis_estado_sigla;
```

Vamos criar mais um procedimento armazenado conforme mostrado abaixo, observe que ainda não tem **parâmetros de entrada**:

```
DELIMITER $$

CREATE PROCEDURE pa_relatorio_venda()

BEGIN

select Venda_Codigo,Venda_Data,Venda_Quantidade,tbcliente.
Cliente_Nome,

tbvendedor.Vendedor_Nome,tbproduto.Produto_Nome,tbloja.Loja_
Nome,tbdepartamento.Departamento_Nome

from tbvenda

LEFT JOIN tbcliente

on tbvenda.Venda_cliente=tbcliente.Cliente_Codigo

LEFT JOIN tbvendedor

on tbvenda.Venda_Vendedor=tbvendedor.Vendedor_Matricula

LEFT JOIN tbproduto

n tbvenda.Venda_produto=tbproduto.Produto_Codigo

LEFT JOIN tbloja

on tbvenda.Venda_Loja=tbloja.Loja_Codigo

LEFT JOIN tbdepartamento

on tbvenda.Venda_Departamento=tbdepartamento.Departamento_
Codigo;

END $$

DELIMITER ;
```

Para executar o procedimento armazenado, basta utilizar o comando:

```
CALL pa_relatorio_venda();
```

Antes de fazermos modificações no procedimento pa_relatorio_venda(), voltaremos ao procedimento pa_estado_sigla() no qual é necessária a inclusão de um parâmetro de entrada, com o objetivo de, na execução do procedimento armazenado, podermos passar a informação da SIGLA de um estado. Iremos chamar o procedimento armazenado passando como parâmetro uma sigla e ele filtrará os valores retornados segundo este parâmetro.

Para fazer esta modificação, iremos excluir o procedimento armazenado e depois criá-lo novamente, incluindo a alteração necessária. Para inicialmente fazermos a exclusão, utilizaremos o comando abaixo:

```
DROP PROCEDURE  pa_estado_sigla;
```

Agora, iremos recriá-lo, colocando um parâmetro de entrada que deve ser passado quando o procedimento armazenado for chamado.

```
DELIMITER $$

CREATE PROCEDURE pa_estado_sigla (var_sigla char(2))

BEGIN

select Estado_Codigo,Estado_Nome,Estado_Sigla as Sigla from tbestado where Estado_Sigla=var_sigla;

END $$

DELIMITER ;
```

Para chamar o procedimento armazenado, agora, colocaremos um parâmetro que é uma sigla qualquer de um estado:

```
CALL  pa_estado_sigla('CE');
```

Observe na definição do parâmetro a informação do tipo de dado, que deve ser o mesmo ou equivalente ao tipo de dado do campo da tabela que será filtrado. No caso do exemplo, o parâmetro **var_sigla** deve ser do mesmo tipo do campo **Estado_Sigla**.

Faremos, agora, uma alteração no procedimento armazenado **pa_relatorio_venda** para colocar alguns parâmetros de entrada.

Vamos excluí-lo e recriá-lo com os devidos parâmetros, conforme os comandos abaixo:

```
DROP PROCEDURE pa_relatorio_venda;

DELIMITER $$

CREATE PROCEDURE pa_relatorio_venda(var_cliente INT,var_
Vendedor INT,var_produto INT,var_loja INT)

BEGIN

select Venda_Codigo,Venda_Data,Venda_Quantidade,tbcliente.
Cliente_Nome,

tbvendedor.Vendedor_Nome,tbproduto.Produto_Nome,tbloja.Loja_
Nome,tbdepartamento.Departamento_Nome

from tbvenda

LEFT JOIN tbcliente

on tbvenda.Venda_cliente=tbcliente.Cliente_Codigo

LEFT JOIN tbvendedor

on tbvenda.Venda_Vendedor=tbvendedor.Vendedor_Matricula

LEFT JOIN tbproduto

n tbvenda.Venda_produto=tbproduto.Produto_Codigo

LEFT JOIN tbloja

on tbvenda.Venda_Loja=tbloja.Loja_Codigo

LEFT JOIN tbdepartamento

on tbvenda.Venda_Departamento=tbdepartamento.Departamento_
Codigo

WHERE tbcliente.Cliente_Codigo=var_cliente and tbvendedor.
Vendedor_Matricula = var_vendedor and

tbproduto.Produto_Codigo=var_produto and

tbloja.Loja_Codigo=var_loja;

END $$

DELIMITER ;
```

Para chamar o procedimento armazenado, teremos que informar quatro parâmetros do tipo inteiro, conforme mostrado no comando abaixo:

```
CALL pa_relatorio_venda(4,1,3,2);
```

PRATIQUE

1) Quais os comandos para criar um banco de dados chamado bdexercicio? Qual o comando para usá-lo?

2) Necessito criar uma tabela no banco de dados bdexercicio para guardar informações de alunos contendo informações de matrícula, nome e curso que o aluno está cursando. Qual o comando SQL para isto?

3) Escreva o comando para tornar a coluna matrícula dos alunos com a propriedade de AUTO INCREMENTO. Para que serve esta propriedade?

4) Escreva o comando para tornar a coluna de matrícula do aluno na chave primária da tabela.

5) Faça a inserção de cinco registros na tabela de alunos que você criou e escreva os comandos no espaço abaixo.

6) Escreva o comando select para que todos os registros da tabela sejam mostrados.

7) Escreva no espaço abaixo o comando SQL para criar uma visão usando como tabela base a tabela que você criou, mostrando apenas as colunas referentes à matrícula do aluno e seu nome.

8) O que é uma função interna do MySQL?

9) Quais as diferenças principais entre uma função e um gatilho (trigger)?

10) O que é um procedimento armazenado? Escreva os comandos para criar um procedimento armazenado que mostre ao ser executado os registros da tabela que você criou.

Relembre

Nesta unidade, avançamos na aprendizagem de técnicas para incrementar o desempenho do nosso banco de dados. Vimos a técnica de particionamento de tabelas, que permite uma organização lógica para otimizar as consultas. Aprendemos a trabalhar com os índices, que são estruturas fundamentais para aumentar a velocidade de respostas das nossas requisições SELECT ao SGBD.

Aprendemos ainda a trabalhar com objetos avançados de banco de dados, como as visões, que permitem que simplifiquemos o trabalho com consultas complexas, e aumentar a segurança dos dados, pois podemos selecionar as colunas das tabelas base que serão mostradas na visão.

Compreendemos também o funcionamento dos objetos programáveis, como as funções, os gatilhos e os procedimento armazenados, entendendo que o uso destes objetos contribui para que o banco de dados tenha um alto desempenho.

Referências

MEHTA, Chintan *et al*. **MySQL 8 Administrator's Guide**. 1. ed., 2018.

MYSQL. Disponível em: https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html. Acesso em: 04 dez. 2020.

SCHWARTZ, Baron; ZAITSEV, Peter; TKACHENKO, Vadim. **O'Reilly High Performance MySQL**. 3. ed., mar., 2012.

Anotações

Rua Coletor Antônio Gadelha, Nº 621
Messejana, Fortaleza – CE
CEP: 60871-170, Brasil
Telefone (85) 3033.5199