Material de Banco de Dados

Atualizado 2022

nov, 2022

Sumário

CAPÍTULO 1

Linguagem SQL

Ótimo local para buscar referências e exemplos de comandos em diversos SG´s.

http://www.w3schools.com/sql/

https://www.devmedia.com.br/sql-server/

1.1 CREATE

- Comando utilizado para criar os principais objetos em um banco de dados.

Neste tópico vamos trabalhar com as diversas variações do comando `CREATE` relacionados ao início dos trabalhos com criação das entidades no banco de dados.

O Primeiro comando é o `CREATE DATABASE`, que cria o Banco de dados e suas dependências, como arquivos e metadados dentro do sistema. Vale lembrar que alguns sistemas gerenciadores de bancos de dados podem implementar maneiras diferentes de tratar os bancos de dados ou espaços de trabalho de cada usuário ou sistema.

No nosso banco de dados de Exemplo temos a criação básica de um banco de dados e a criação de uma tabela chamada `Clientes`. Depois usamos o comando use para posicionar a execução dos comandos no banco de dados `MinhaCaixa`.

```
1  CREATE DATABASE MinhaCaixa;
2
3  use MinhaCaixa;
4
5  CREATE TABLE Clientes (
6    ClienteCodigo int,
7    ClienteNome varchar(20)
8  );
```

Podemos ter variações do comando `CREATE TABLE` de acordo com a necessidade. Abaixo temos diversas implementações do comando `CREATE` e suas CONSTRAINT´s.

1.1.1 CONSTRAINT PRIMARY KEY & IDENTITY

Nesse exemplo adicionamos uma chave primária ao campo `ClienteCodigo` e configuramos a propriedade `IDENTITY` que vai gerar um número com incremento de (um) a cada inserção na tabela `Clientes`. Você pode personalizar o incremento de acordo com sua necessidade, neste exemplo temos (1,1) iniciando em um e incrementando um.

```
CREATE TABLE Clientes (
  ClienteCodigo int IDENTITY (1,1) CONSTRAINT PK_Cliente PRIMARY KEY,
  ...
);
```

Nesse exemplo adicionamos uma chave primária composta.

```
CREATE TABLE Clientes (
  ClienteCodigo int IDENTITY (1,1) ,
  ClienteCPF(11)
      CONSTRAINT PK_Cliente PRIMARY KEY (ClienteCodigo,ClienteCPF)
);
```

1.1.2 CONSTRAINT FOREIGN KEY

Neste exemplo temos a criação da `FOREIGN KEY` dentro do bloco de comando `CREATE`. Se tratando de uma chave estrangeira temos que tomar o cuidado de referenciar tabelas que já existem para evitar erros. Repare que no comando abaixo estamos criando uma tabela nova chamada `Contas` e especificando que o código de cliente deverá estar cadastrado na tabela de Cliente, portanto deve existir antes uma tabela `Cliente` que será referenciada nessa chave estrangeira `FOREIGN KEY`. Repare que sempre damos um nome para a `CONSTRAINT`, isso é uma boa prática, para evitar que o sistema dê nomes automáticos.

```
CREATE TABLE Contas
  (
  AgenciaCodigo int,
  ContaNumero VARCHAR (10) CONSTRAINT PK_CONTA PRIMARY KEY,
  ClienteCodigo int,
  ContaSaldo MONEY,
  ContaAbertura datetime
  CONSTRAINT FK_CLIENTES_CONTAS FOREIGN KEY (ClienteCodigo) REFERENCES _
 →Clientes(ClienteCodigo)
  );
```

1.1.3 ALTERTABLEADD COLUMN

```
ALTER TABLE Pessoas ADD PessoaSexo CHAR(2);
```

1.1.4 ALTER TABLE ADD CONSTRAINT

Também podemos adiconar CONSTRAINT´s através do comando `ALTER TABLE ... ADD CONSTRAINT`. Geralmente após criar todas as entidades podemos então criar as restrições entre elas.

```
ALTER TABLE Contas ADD CONSTRAINT FK_CLIENTES_CONTAS FOREIGN KEY (ClienteCodigo)
REFERENCES Clientes(ClienteCodigo);
```

1.1.5 CONSTRAINT´ s de domínio

```
ALTER TABLE Clientes ADD CONSTRAINT chk_cliente_saldo CHECK ([ClienteNascimento] <
→GETDATE() AND ClienteNome <> 'Sara');
```

Abaixo a mensagem de tentativa de violação da `CONSTRAINT` acima.

```
The INSERT statement conflicted with the CHECK constraint "chk_cliente_saldo". The
→conflict occurred in database "MinhaCaixa", table "dbo.Clientes".
```

Apenas checando uma condição, data de nascimento menor que data atual. No SQL Server para pegarmos a data atual usamos `GETDATE()`:

```
ALTER TABLE Clientes ADD CONSTRAINT TESTE CHECK ([ClienteNascimento] < GETDATE());
```

1.2 INSERT

- Comando utilizando para popular as tabelas no banco.

O comando `INSERT` também possui algumas variações que devem ser respeitadas para evitar problemas. O primeiro exemplo abaixo mostra a inserção na tabela `Clientes`. Repare que logo abaixo tem um fragmento da criação da tabela `Clientes` mostando que o campo `ClienteCodigo` é `IDENTITY`, portanto não deve ser informado no momento do `INSERT`.

```
INSERT Clientes (ClienteNome) VALUES ('Nome do Cliente');

CREATE TABLE Clientes
  (
  ClienteCodigo int IDENTITY CONSTRAINT PK_CLIENTES PRIMARY KEY...
```

Quando vamos fazer o `INSERT` em uma tabela que não possui o campo `IDENTITY` passamos o valor desejado, mesmo que o campo seja `PRIMARY KEY`.

```
INSERT Clientes (ClienteCodigo, ClienteNome) VALUES (1, 'Nome do Cliente');

CREATE TABLE Clientes
  (
  ClienteCodigo int CONSTRAINT PK_CLIENTES PRIMARY KEY...

INSERT Clientes (colunas) VALUES (valores);

INSERT INTO Clientes SELECT * FROM ...
```

1.3 UPDATE

- Comando utilizado para alterar registros em um banco de dados. Antes de executar qualquer comando `UPDATE`, procure se informar sobre transações (será abordado mais pra frente).
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
UPDATE CartaoCredito SET CartaoLimite = 1000 WHERE ClienteCodigo = 1;
```

1.4 DELETE

- Comando utilizado para deletes registros em um banco de dados.
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
1 DELETE FROM CartaoCredito WHERE ClienteCodigo = 1;
```

1.5 SELECT

- Comando utilizado para recuperar as informações armazenadas em um banco de dados.

O comando `SELECT` é composto dos atributos que desejamos, a ou as tabela(s) que possuem esses atributos e as condições que podem ajudar a filtrar os resultados desejados. Não é uma boa prática usar o `*` ou *star* para trazer os registros de uma tabela. Procure especificar somente os campos necessários. Isso ajuda o motor de execação de consultas a construir bons planos de execução. Se você conhecer a estrutura da tabela e seus índices, procure tirar proveito disso usando campos chaves, ou buscando e filtrando por atributos que fazem parte de chaves e índices no banco de dados.

```
1 SELECT * FROM Clientes;
```

- O Comando `FROM` indica a origem dos dados que queremos.

Na consulta acima indicamos que queremos todas as informações de clientes. É possível especificar mais de uma tabela no comando `FROM`, porém, se você indicar mais de uma tabela no comando `FROM`, lembre-se de indicar os campos que fazem o relacionamento entre as tabelas mencionadas na cláusula `FROM`.

- O comando `WHERE` indica quais as consições necessárias e que devem ser obedecidadas para aquela consulta.

Procure usar campos restritivos ou indexados para otimizar sua consulta. Na tabela `Clientes` temos o código do cliente como chave, isso mostra que ele é um bom campo para ser usado como filto.

```
1 SELECT ClienteNome FROM Clientes WHERE ClienteCodigo=1;
```

- Um comando que pode auxiliar na obtenção de metadados da tabela que você deseja consultar é o comando `sp_help`. Esse comando mostrar a estrutura da tabela, seus atributos, relacionamentos e o mais importante, se ela possui índice ou não.

```
1 sp_help clientes
```

- Repare que a tabela Clientes possui uma chave no `ClienteCodigo`, portanto se você fizer alguma busca ou solicitar o campo `ClienteCodigo` a busca será muito mais rápida. Caso você faça alguma busca por algum campo que não seja chave ou não esteja "indexado" (Veremos índice mais pra frente) a busca vai resultar em uma varredura da tabela, o que não é um bom negócio para o banco de dados.
- Para escrever um comando `SELECT` procuramos mostrar ou buscar apenas os atributos que vamos trabalhar, evitando assim carregar dados desnecessários e que serão descartados na hora da montagem do formulário da aplicação. Também recomendamos o uso do nome da Tabela antes dos campos para evitar erros de ambíguidade que geralmente aparecem quando usamos mais de uma tabela.

```
1 SELECT Clientes.ClienteNome FROM Clientes;
```

- Você pode usar o comando `AS` para dar apelidos aos campos e tabelas para melhorar a visualiação e compreensão.

```
1 SELECT Clientes.ClienteNome AS Nome FROM Clientes;
2
3 SELECT C.ClienteNome FROM Clientes AS C;
```

- Você pode usar o operador `ORDER BY` para ordenar os registros da tabela.

Procure identificar os campos da ordenação e verificar se eles possuem alguma ordenação na tabela através de algum índice. As operações de ordenação são muito custosas para o banco de dados. A primeira opção traz os campos ordenados em ordem ascendente `ASC`, não precisando informar o operador. Caso você deseje uma ordenação descendente você deverá informar o `DESC`.

```
SELECT Clientes.ClienteNome FROM Clientes
  ORDER BY Clientes.ClienteNome;

SELECT Clientes.ClienteNome FROM Clientes
  ORDER BY Clientes.ClienteNome DESC;
```

- Outro operador que é muito utilizado em parceria com o `ORDER BY` é o `TOP`, que permite limitar o conjunto de linhas retornado. Caso ele não esteja associado com o `ORDER BY` ele trará um determinado conjunto de dados baseado na ordem em que estão armazenados. Caso você use um operador `ORDER BY` ele mostrará os `TOP` maiores ou menores. O Primeiro exemplo mostra as duas maiores contas em relação ao seu saldo. A segunda, as duas menores.

```
SELECT TOP 2 ContaNumero, ContaSaldo FROM Contas
  ORDER BY ContaSaldo DESC;

SELECT TOP 2 ContaNumero, ContaSaldo FROM Contas
  ORDER BY ContaSaldo;
```

- Podemos usar mais de uma tabela no comando `FROM` como falamos anteriormente, porém devemos respeitar seus relacionamentos para evitar situações como o exemplo abaixo. Execute o comando e veja o que acontece.

```
SELECT * FROM Clientes, Contas;
```

- A maneira correta deve levar em consideração que as tabelas que serão usadas tem relação entre si “chaves”, caso não tenham, poderá ser necessário passar por um outra tabela antes. Lembre-se das tabelas associativas.

```
SELECT CLientes.ClienteNome, Contas.ContaSaldo
  FROM Clientes, Contas
  WHERE Clientes.ClienteCodigo=Contas.ClienteCodigo;
```

- O comando `LIKE` é usado para encontrar registros usando parte do que sabemos sobre ele. Por exemplo podemos buscar todas as pessoas que tenham nome começado com `R`, usando um coringa `%` (Percentual). Podemos fazer diversas combinação com o `%`.

Documentação do comando LIKE

```
SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua LIKE 'a%' AND ClienteRua NOT
 →LIKE 'E%';

SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua LIKE '%a%';

SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua LIKE '%a';

SELECT ClienteRua FROM dbo.Clientes WHERE ClienteRua NOT LIKE 'a%';
```

- O Comando `CASE` é utilizado quando queremos fazer validações e até gerar novar colunas durante a execução da consulta. No exemplo abaixo fazemos uma classificação de um cliente com base no seu saldo, gerando assim uma nova coluna `Curva Cliente`.

```
SELECT ContaNumero,
  CASE WHEN ContaSaldo < 200 THEN 'Cliente C' WHEN ContaSaldo < 500 THEN 'Cliente B
 →'
  ELSE 'Cliente A' END AS 'Curva Cliente'
  FROM dbo.Contas;
```

- Podemos incluir em nossas consultas diversos operadores condicionais: = (igual), <> (diferente), > (maior), < (menor), <= (menor ou igual), >= (maior ou igual), `OR` (ou), `AND` (e) e `BETWEEN` (entre).

```
SELECT Nome_agencia, Numero_conta, saldo
  FROM Conta
  WHERE saldo > 500 AND Nome_agencia = 'Joinville';

SELECT AgenciaCodigo FROM dbo.Agencias
  WHERE AgenciaCodigo BETWEEN 1 AND 3;
```

- O `ALIAS` ou apelido ajuda na exibição de consultas e tabelas. Dessa forma podemos dar nomes amigáveis para campos e tabelas durante a execução de consultas. Use sempre o `AS` antes de cada `ALIAS`, mesmo sabendo que não é obrigatório.

```
SELECT Nome_agencia,C.Numero_conta,saldo AS [Total em Conta],
    Nome_cliente,D.Numero_conta AS 'Conta do Cliente'
  FROM Conta AS C, Depositante AS D
  WHERE C.Numero_conta=D.Numero_conta AND Nome_cliente IN ('Rodrigo','Laura')
  ORDER BY saldo DESC
```

- O comando `DISTINCT` serve para retirar do retorno da consulta registros repetidos.

```
SELECT DISTINCT Cidade_agencia FROM Agencia;
```

- A SUB CONSULTA, `IN` e `NOT IN` são poderosos recursos para auxiliar em buscas e filtragem de registros. Podemos criar subconjuntos de registros e usar operadores como `IN` para validar se os registros estão dentro daquele subconjunto.

```
SELECT AgenciaCodigo FROM dbo.Agencias
  WHERE AgenciaCodigo NOT IN ('1','4');

SELECT Contas.ContaNumero, Contas.ContaSaldo, Contas.AgenciaCodigo
  FROM Contas INNER JOIN
    (
    SELECT AgenciaCodigo, MAX(ContaSaldo) AS VALOR
    FROM Contas
    GROUP BY AgenciaCodigo
    ) AS TB2
  ON
  TB2.AgenciaCodigo=Contas.AgenciaCodigo AND TB2.VALOR=Contas.ContaSaldo;
```

- Os operadores `UNION` e `UNION ALL` ajudam a consolidar conjuntos de registros que são retornados por consultas distintas. O operador `ALL` faz a junção das consultas sem eliminar itens duplicados. Precisamosobedecer o mesmo número de colunas e tipos de dados entre as consultas.

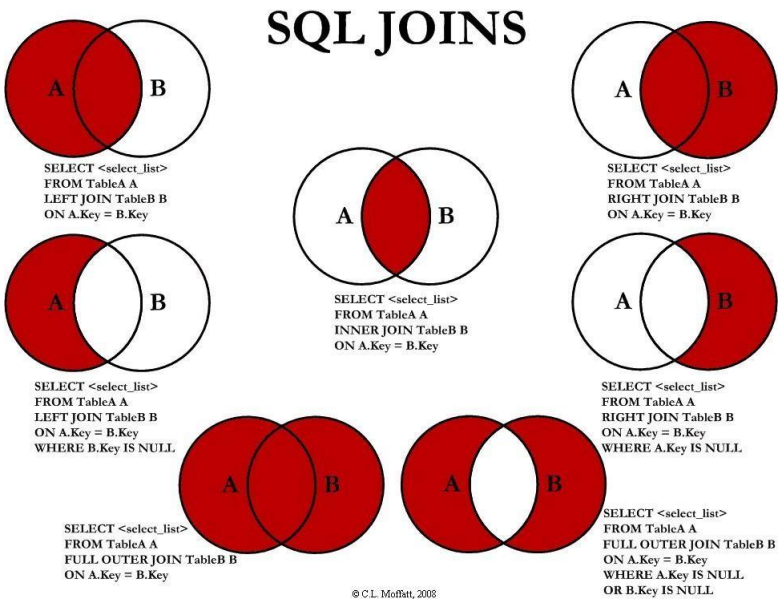
```
SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 1
UNION
SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 2;

SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 1
UNION ALL
SELECT ClienteNome FROM dbo.Clientes WHERE ClienteCodigo = 1;
```

- Existem diversos tipos de `JOINS`. O mais tradicional e restritivo é o `JOIN` ou `INNER JOIN` que requer que o registros usado na comparação exista em ambas as tabelas.

No exemplo abaixo, o `ClienteCodigo` não poderá ser vazio em nenhuma das tabelas envolvidas, caso isso ocorra, aquela linha não será retornada no resultado.

Fonte da imagem: Representação Visual das Joins




```sql
SELECT * FROM Clientes
  JOIN Contas
    ON Clientes.ClienteCodigo=Contas.ClienteCodigo;

SELECT * FROM CLIENTES
  INNER JOIN Contas
    ON Clientes.ClienteCodigo=Contas.ClienteCodigo;
```

- LEFT JOIN

O comando LEFT indica que todos os registros existentes na tabela da sua esquerda serão retornados e os registros da outra tabela da direita irão ser retornados ou então virão em branco.

```sql
SELECT ClienteNome, ContaSaldo,
  CASE WHEN CartaoCodigo IS NULL THEN 'LIGAR' ELSE 'NÃO INCOMODAR' END AS
→'NN'
  FROM Clientes
  INNER JOIN Contas
  ON (Contas.ClienteCodigo = Clientes.ClienteCodigo)
  LEFT JOIN CartaoCredito
  ON (CartaoCredito.ClienteCodigo = Clientes.ClienteCodigo);
```

- RIGHT

Já o comando RIGHT traz todos os registros da tabela da direita e os registos da tabela da esquerda, mostrando em branco aqueles que não tem relação.

```sql
SELECT * FROM CartaoCredito RIGHT JOIN Clientes ON CartaoCredito.
→ClienteCodigo=Clientes.ClienteCodigo;
```

- FULL

O comando FULL retorna todos os registros das tabelas relacionadas, mesmo que não exista um correspondente entre elas.

```sql
SELECT * FROM CartaoCredito FULL OUTER JOIN Clientes ON CartaoCredito.
→ClienteCodigo=Clientes.ClienteCodigo;
```

- CROSS

Efetua um operação de produto cartesiano, para cada registro de uma tabela ele efetua um relacionamento com os registros das outras tabelas.

```
SELECT * FROM CLIENTES CROSS JOIN Contas;
```

- As FUNÇÕES DE AGREGAÇÃO, `SUM` (soma), `MIN` (mínimo), `MAX` (máximo), `COUNT` (contagem), `AVG` (média), permitem um nível mais robusto de informação, criando conjuntos de dados agrupados, médias entre outros, permitindo o resumo e a totalização de conjuntos de resultados. Sempre que usarmos a função de agregação em conjunto com um campo agregador, devemos usar a função `GROUP BY` para indicar qual o campo será o responsável pelo agrupamento das informações.

Caso você deseje comparar conjuntos de informações contidos na função de agragação você deve compará-los usando o `HAVING`.

```
SELECT TOP 2 AgenciaNome, SUM(ContaSaldo) AS TOTAL
  FROM Contas,  Agencias
  WHERE Agencias.AgenciaCodigo=Contas.AgenciaCodigo
  GROUP BY AgenciaNome
  HAVING SUM(ContaSaldo) > (SELECT MAX(ContaSaldo) AS VALORMETA FROM Contas AS
→META)
  ORDER BY 2 DESC;

SELECT SUM( Contas.ContaSaldo),
  AgenciaCodigo, ContaNumero
  FROM Contas
  GROUP BY AgenciaCodigo,ContaNumero
  --WHERE COM AVG ???
  --WHERE COM SUBCONSULTA ???
  HAVING SUM( Contas.ContaSaldo) > (SELECT AVG( Contas.ContaSaldo) FROM  Contas); -
→-667,0833

SELECT MAX(ContaSaldo)  FROM  Contas;
SELECT MIN(ContaSaldo)  FROM  Contas;
SELECT AVG(ContaSaldo) FROM  Contas;
SELECT COUNT(*), COUNT(CONTAS.ClienteCodigo), COUNT(DISTINCT CONTAS.ClienteCodigo)
→FROM  Contas;
```

- EXISTS

O comando EXISTS é parecido com o comando `IN`, quando queremos comparar mais de um campo contra uma subconsulta.

```
SELECT * FROM  Contas C
    WHERE EXISTS
                    (SELECT * FROM  CartaoCredito CC
                            WHERE C.ClienteCodigo=CC.ClienteCodigo
                            AND C.AgenciaCodigo=CC.AgenciaCodigo
                    )
```

- FUNÇÕES DE Data e Hora

```
SET DATEFORMAT YDM

SET LANGUAGE PORTUGUESE

SELECT YEAR(getdate()) -YEAR( Clientes.ClienteNascimento),
  DATEDIFF(YEAR,ClienteNascimento,GETDATE()),
  DATEPART(yy,ClienteNascimento),
  dateadd(yy,1,ClienteNascimento),
  EOMONTH(GETDATE()),
  DATENAME(MONTH,(GETDATE()))
FROM  Clientes;
```

```sql
SELECT * FROM  Contas
  WHERE YEAR(ContaAbertura) = '2011'
  ORDER BY ContaAbertura;
```

- Variáveis

Muitas vezes necessitamos armazenar determinados valores para uso posterior. Um exemplo é guardar um valor total em uma variável para que ele seja usado em cálculo de percentual por exemplo

```sql
declare @numero int
set @numero = 1

declare @dia int
set @dia = (select day(getdate()))
```

- SELECT INTO

```sql
      SELECT Clientes.ClienteNome,
      DATEDIFF(YEAR,Clientes.ClienteNascimento,GETDATE()) AS IDADE
      INTO ClientesIdade -- O comando INTO vem depois do campos listados no SELECT e antes do FROM.
      FROM Clientes

      SELECT * FROM ClientesIdade
```

- CAST, CONVERT e concatenação

Comandos utilizados para converter tipos de dados e concatenar Strings.

```sql
SELECT Clientes.ClienteNome + Clientes.ClienteCidade FROM Clientes;

SELECT Clientes.ClienteNome + ' ' + Clientes.ClienteCidade FROM Clientes;

SELECT Clientes.ClienteNome + ' de ' + Clientes.ClienteCidade FROM Clientes;

SELECT Clientes.ClienteNome + ' - R$ ' + CAST (Contas.ContaSaldo AS VARCHAR(10) )FROM Clientes INNER JOIN Contas ON Contas.ClienteCodigo = Clientes.ClienteCodigo;

SELECT Clientes.ClienteNome + ' - R$ ' + CONVERT  (VARCHAR(10), Contas.ContaSaldo )FROM Clientes INNER JOIN Contas ON Contas.ClienteCodigo = Clientes.ClienteCodigo;
```

1.6 VIEW

- Comando utilizado para alterar registros em um banco de dados. Antes de executar qualquer comando `UPDATE`, procure se informar sobre transações (será abordado mais pra frente).
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```sql
CREATE VIEW ClientesIdade
AS
SELECT ClienteNome,DATEDIFF(YEAR,ClienteNascimento,GETDATE()) AS Idade FRO
dbo.Clientes;
```

1.7 FUNÇÕES

- Uma função é uma sequência de comandos que executa alguma tarefa e que tem um nome. A sua principal finalidade é nos ajudar a organizar programas em pedaços que correspondam a como imaginamos uma solução do problema.

  Exemplo de um Função:

```
CREATE FUNCTION fnRetornaAno (@data DATETIME)
RETURNS int
AS
  BEGIN
  DECLARE @ano int
  SET @ano = YEAR(@data)

      RETURN @ano

END
```

- Chamada ou execução da função

```
SELECT dbo.fnRetornaAno(GETDATE())

SELECT dbo.fnRetornaAno(Clientes.ClienteNascimento) FROM dbo.Clientes
```

1.8 PROCEDURES

- Uma procedure é um bloco de comandos ou instruções SQL organizados para executar uma ou mais tarefas. Ela pode ser utilizada para ser acionada através de uma chamada simples que executa uma série de outros comandos.

```
CREATE PROCEDURE uspRetornaIdade
@CodigoCliente int
AS
SELECT Clientes.ClienteNome, YEAR(GETDATE())-YEAR(ClienteNascimento) AS IDADE
FROM Clientes
INNER JOIN Contas ON Clientes.ClienteCodigo=Contas.ClienteCodigo
WHERE Clientes.ClienteCodigo = @CodigoCliente;
```

- Execução da procedure, opção 1

```
exec uspRetornaIdade 1;
```

- Execução da procedure, opção 2

```
declare @parametro int
set @parametro = 1 --Código do Cliente desejado
exec uspRetornaIdade @parametro;
```

1.8.1 IF

- Comando utilizado para checar condições.

```
CREATE PROCEDURE uspRetornaSeTemCartao
@CodigoCliente int
AS
BEGIN
```

(continues on next page)

(continuação da página anterior)

```
5
6  DECLARE @CodigoClienteCartao INT
7
8  SET @CodigoClienteCartao = (SELECT CartaoCredito.ClienteCodigo FROM Clientes LEFT
   →JOIN CartaoCredito
9  ON CartaoCredito.ClienteCodigo = Clientes.ClienteCodigo WHERE CartaoCredito.
   →ClienteCodigo = @CodigoCliente)
10
11         IF @CodigoClienteCartao IS NULL
12                 BEGIN
13                 SELECT * FROM CartaoCredito WHERE ClienteCodigo = @CodigoCliente;
14                 END
15         ELSE
16                 BEGIN
17                 SELECT 'LIGAR', * FROM Clientes WHERE ClienteCodigo =
   →@CodigoCliente
18                 END
19
20 END;
21
22 EXEC uspRetornaSeTemCartao @CodigoCliente = 25; -- TEM CARTÃO
23
24 EXEC uspRetornaSeTemCartao @CodigoCliente = 1; --NÃO TEM CARTÃO
```

1.8.2 WHILE

- Comando utilizado para realizar laços de repetição.

```
1 DECLARE @contador INT
2 SET @contador = 1
3 WHILE @contador <= 5
4 BEGIN
5   SELECT @contador
6   SET @contador = @contador + 1
7 END
```

1.9 CURSORES

- Cursor.

  Exemplo de um Cursor:

```
1   DECLARE @ClienteNome VARCHAR(50), @ClienteSexo CHAR(1), @contador INT=0;
2
3     DECLARE [cursorListaCliente] CURSOR FOR
4     SELECT Clientes.ClienteNome , ClienteSexo
5     FROM Clientes
6
7     OPEN [cursorListaCliente]
8     FETCH NEXT FROM [cursorListaCliente] INTO @ClienteNome, @ClienteSexo;
9
10    WHILE @@FETCH_STATUS = 0
11    BEGIN
12         SET @contador=@contador+1;
13
14    SELECT @ClienteNome as Nome, @ClienteSexo AS Sexo, @contador;
15    FETCH NEXT FROM [cursorListaCliente] INTO @ClienteNome, @ClienteSexo
```

(continues on next page)

(continuação da página anterior)

```
    END
    CLOSE [cursorListaCliente];
      DEALLOCATE [cursorListaCliente];
```

1.10 TRANSAÇÕES

1.10.1 Transações

- Comando utilizado para alterar registros em um banco de dados. Antes de executar qualquer comando `UPDATE`, procure se informar sobre transações (será abordado mais pra frente).
- Sempre que for trabalhar com o comando `UPDATE` ou `DELETE`, procure executar um `SELECT` antes para validar se os registros que serão afetados, são exatamente aqueles que você deseja.

```
BEGIN TRAN --> Inicia a transação

UPDATE dbo.CartaoCredito SET CartaoLimite = CartaoLimite * 1.1

COMMIT --> Finaliza a transação

--OR

ROLLBACK --> Desfaz a transação
```

Execute primeiro sem o WHERE e verifique que nenhuma linha será alterada. Depois remova o comentário e verá que apenas uma linha foi alterada.

```
BEGIN TRAN

UPDATE dbo.CartaoCredito SET CartaoLimite = CartaoLimite * 1.1
--WHERE ClienteCodigo = '12'

IF (@@ROWCOUNT > 1 OR @@ERROR > 0)

  ROLLBACK

ELSE

  COMMIT
```

1.10.2 Try Catch

```
BEGIN TRY

  SELECT 1/0

END TRY

BEGIN CATCH
  SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

1.11 TRIGGERS

- Comando vinculado a uma tabela que executa um ação assim que algum comando de UPDATE, INSERT ou DELETE é executado na tabela onde a trigger está vinculada.

1.11.1 Trigger para INSERT

```
CREATE TRIGGER trgINSERT_CLIENTE
ON Clientes
FOR INSERT
AS
BEGIN
INSERT clientes_audit
SELECT *,[TRG_OPERACAO] = 'INSERT', [TRG_DATA]=GETDATE(), [TRG_FLAG]='NEW' FROM _
→Inserted
END;
```

1.11.2 Trigger para DELETE

```
CREATE TRIGGER trgDELETE_CLIENTE
ON dbo.Clientes
FOR DELETE
AS
BEGIN
INSERT dbo.clientes_audit SELECT *,[TRG_OPERACAO] = 'DELETE',[TRG_DATA]=GETDATE(),
→[TRG_FLAG]='OLD' FROM Deleted
END;
```

1.11.3 Trigger para UPDATE

```
CREATE TRIGGER trgUPDATE_CLIENTE
ON dbo.Clientes
FOR UPDATE
AS
BEGIN
INSERT dbo.clientes_audit SELECT *,[TRG_OPERACAO] = 'UPDATE',[TRG_DATA]=GETDATE(),
→[TRG_FLAG]='OLD' FROM Deleted
INSERT dbo.clientes_audit SELECT *,[TRG_OPERACAO] = 'UPDATE',[TRG_DATA]=GETDATE(),
→[TRG_FLAG]='NEW' FROM Inserted
END;
```

1.12 INDICES

- Criação de índices e estatísitcas

Os índices garantem um bom desempenho para as consultas que serão realizadas no banco de dados. Comece verificando com a procedure `sp_help` os metadados das tabelas para verificar se não existe um índice que possa ajudar na sua consulta.

Caso precise criar um índice comece analisando os campos que estão na sua cláusula WHERE. Esses campos são conhecidos como predicados. Ainda dentro da cláusula WHERE procure filtrar primeiramente os campos com maior seletividade, que possam filtar os dados de forma que não sejam trazidos ou pesquisados dados descessários.

Em seguida olhe os campos da cláusula SELECT e adicione eles no índice.

- Atenção Leia o material complementar na biblioteca Virtual
- Exemplo

A consulta abaixo busca nome e data de nascimentos do cliente com base em uma data passada pelo usuário ou sistema. Como primeiro passo vamos olhar a cláusula WHERE e em seguida a cláusula SELECT. Dessa forma temos um índice que deverá conter ClienteNascimento e ClienteNome onde ClienteNascimento é o predicado.

Comando

```
      SELECT Clientes.ClienteNome, Clientes.ClienteNascimento
      FROM Clientes
      WHERE ClienteNascimento >= '1980-01-01'
```

Índice

```
      CREATE INDEX IX_NOME ON Clientes
      (
              ClienteNascimento,
              ClienteNome
      )
```

1.13 BACKUP

- Comando .

1.13.1 Comando para BACKUP

```
BACKUP DATABASE [MinhaCaixa]
TO  DISK = 'C:\bkp\MinhaCaixa2018.bak';


BACKUP DATABASE [MinhaCaixa]
TO  DISK = N'C:\bkp\MinhaCaixa2018_diff.bak'
WITH  DIFFERENTIAL , STATS = 10;

BACKUP LOG [MinhaCaixa] TO
DISK = N'C:\bkp\MinhaCaixa2018_log.trn' WITH NOFORMAT,  STATS = 10;


USE [master]
RESTORE DATABASE [MinhaCaixa]
FROM  DISK = N'C:\bkp\MinhaCaixa2018.bak'
WITH REPLACE,  STATS = 10;

USE [master]

RESTORE DATABASE [MinhaCaixa] FROM  DISK = N'C:\bkp\MinhaCaixa2018.bak'
WITH  FILE = 1,  NORECOVERY,  NOUNLOAD,  STATS = 5
RESTORE DATABASE [MinhaCaixa] FROM  DISK = N'C:\bkp\MinhaCaixa2018_diff.bak'
WITH  FILE = 1,  NORECOVERY,  NOUNLOAD,  STATS = 5
RESTORE LOG [MinhaCaixa] FROM  DISK = N'C:\bkp\MinhaCaixa2018_log.trn'
WITH  FILE = 1,  NOUNLOAD,  STATS = 5;
```