

SQL Server

Readme.rmd

Sergio Pedro R Oliveira

2022-07-31

Contents

1	Objetivo	4
2	Referência	4
3	Modulo 24 - Instalação e delimitador GO	5
3.1	Instalação	5
3.2	Acessando SQL Server pelo terminal	6
3.3	Bancos do sistema	6
3.4	Detalhes básicos do SQL SERVER	7
3.5	Uso do delimitador GO	8
4	Modulo 25 - Arquitetura do SQL Server	9
4.1	Acessando arquivos de banco de dados	9
4.2	Arquitetura do SQL Server na maquina	9
4.3	Tipos de arquivos	9
4.4	TRANSACTION - Transação	11
4.5	Função ERRO	11
4.6	Criando Grupos de arquivos	12
4.7	Direcionando dados de tabela para grupos de arquivos (NDF)	13
5	Tipos de dados	14
5.1	Numéricos exatos	14
5.2	Numéricos aproximados	15
5.3	Data e hora	15
5.4	Cadeias de caracteres	16
5.5	Cadeias de caracteres Unicode	16
5.6	Cadeia de caracteres binária	17
5.7	Outros tipos de dados	17

6	Modulo 26 PARTE 1 - Comandos básicos, CONSTRAINTS e descrição de tabelas	18
6.1	Comandos básicos	18
6.2	Regras/Restrições - CONSTRAINTS	21
6.3	Comandos de descrição tabelas - SP__	24
7	Modulo 26 PARTE 2 - Funções, Projeções, Seleções e Junções	26
7.1	Funções	26
7.2	Projeção, seleção e Junção - SELECT, WHERE e JOIN	30
8	Modulo 26 PARTE 3 - Conversão de tipo de dados	35
8.1	Conversão de dados automatica pelo sistema	35
8.2	Tabela de conversões de dados automatico pelo sistema	36
8.3	Funções de conversão	37
9	Modulo 26 PARTE 4 - Importação de arquivo de dados	39
9.1	Aspacetos importantes da importação de Arquivos	39
9.2	Função de importação de arquivos BULK INSERT	40
10	Modulo 26 PARTE 5 - Técnica de “flag-ar” coluna (SELECT)	41
11	Modulo 27 PARTE 1 - TRIGGER (Gatilho) DML (Data Manipulation Language)	43
11.1	Principais fatores a serem auditados por um TRIGGER	44
11.2	Conceitos Préliminares - Argumentos temporais (INSERTED/DELETED) e Declaração de variáveis (DECLARE)	45
11.3	CREATE TRIGGER	47
11.4	ALTER TRIGGER	49
11.5	DROP TRIGGER	49
11.6	Boas Práticas	50
12	Modulo 27 PARTE 2 - Simplificando TRIGGER (Gatilho) e Bloco Anônimo	52
12.1	Bloco Anônimo	52
12.2	Simplificando TRIGGERS	53
13	Modulo 27 PARTE 3 - TRIGGER (Gatilho), TRANSAÇÃO (TRANSACTION) e ERROR	55
13.1	ERROR e RAISERROR	55
13.2	TRANSACTION (Transação)	56
13.3	TRIGGER com restrição de regra de negócio e ERROR	58

14 SCHEMAS	60
14.1 SCHEMA padrão do sistema - <i>dbo</i>	61
14.2 Criação de SCHEMA	61
14.3 Criação de um SCHEMA no SSMS	61
14.4 Adicionando uma tabela a um SCHEMA	62
14.5 Permissões do SCHEMA	62
14.6 ALTER SCHEMA	63
14.7 Listar todos os SCHEMAS do banco de dados	63
15 Modulo 28 - Variáveis e PROCEDURES	64
15.1 Variáveis locais e globais	64
15.2 STORAGE PROCEDURES (ou STORED PROCEDURES)	65
15.3 PROCEDURES	66
16 Categorias de comandos	72
16.1 DML - <i>Data Manipulation Language</i> (Linguagem de Manipulação de Dados)	72
16.2 DDL - <i>Data Definition Language</i> (Linguagem de definição de dados)	74
16.3 DCL - <i>Data Control Language</i> (Linguagem de Controle de Dados)	77
16.4 TCL - <i>Tool Command Language</i> (Linguagem de Comandos de Ferramentas)	84
17 Observações	85
17.1 Problemas para fazer <i>login</i> o SSMS	85
17.2 Abreviações do nome de restrições (CONSTRAINTS) no dicionário de dados - sistema (boas práticas)	85
17.3 Formato da data no sistema	85
18 Andamento dos Estudos	86
18.1 Assunto em andamento	86

1 Objetivo

Estudo dirigido de SQL Server.

2 Referência

Vídeo aulas “O curso completo de Banco de Dados e SQL, sem mistérios” - Udemy.

3 Modulo 24 - Instalação e delimitador GO

3.1 Instalação

3.1.1 Instalar SQL-server

- Versão:
Versão usada é a express 2019, por ser a versão mais completa gratuita.
- Ubuntu
<https://docs.microsoft.com/pt-br/sql/linux/quickstart-install-connect-ubuntu?view=sql-server-ver15>
Basta seguir o passo a passo do site, ou pesquisar por “SQL Server Ubuntu” no youtube e seguir alguns tutoriais.
- Windows

3.1.2 Instalar Azure Data Studio

- Gerenciador de banco de dados usado para SQL-server, que estou usando no Ubuntu.
- Onde baixar:
<https://docs.microsoft.com/pt-br/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15>

3.2 Acessando SQL Server pelo terminal

- Execute o `sqlcmd` com parâmetros para o nome do SQL Server (-S), o nome de usuário (-U) e a senha (-P). Neste tutorial, você está se conectando localmente, portanto, o nome do servidor é `localhost`. O nome de usuário é SA (system administrator, equivalente ao root do MySQL) e a senha é a mesma fornecida para a conta SA durante a instalação.

```
sqlcmd -S localhost -U SA -P 'YourPassword'
```

- É possível omitir a senha na linha de comando para receber uma solicitação para inseri-la.

```
sqlcmd -S localhost -U SA
```

3.3 Bancos do sistema

- São os bancos de dados do sistema que armazenam os dicionários de dados.
- Bancos de dados do sistema:
 - **master**
 - * É o banco de dados principal do sistema.
 - * Todas as informações dos outros bancos de dados criados ficam armazenados nele.
 - **model**
 - * São modelos de tabelas e bancos de dados, que ficam armazenados nesse banco de dados.
 - * Pode servir de modelo automatico na criação de uma nova tabela ou banco de dados.
 - **msdb**
 - * Armazenamento de rotinas.
 - * Integrations Services, área de BI (ferramenta de **ETL**).
 - **tempdb**
 - * Bancos de dados temporarios, ele é apagado todo vez que fecha e abre o sistema do banco de dados.
 - * Muito utilizado para agilizar o teste de aplicações.

3.4 Detalhes básicos do SQL SERVER

3.4.1 Inserindo comentarios

- Um comentário é uma seqüência arbitrária de caracteres começando por dois hífen (“--”) e prosseguindo até o fim da linha.
- Como alternativa, podem ser utilizados blocos de comentários no estilo C (*/*bloco de comentarios*/*). Utilizado para comentar mais de uma linha.

3.4.2 Extensão de arquivo script SQL

- O arquivo com o script SQL é salvo em “.sql”.
- As três formas recomendadas de escrever os script’s são:
 - **SQL server Management Studio (SSMS)**
É um gerenciador de bancos de dados oferecidos pela microsoft, ótimo para gerenciar e trabalhar com banco de dados e arquivos “.sql”.
 - **Azure**
É um gerenciador de banco de dados e oferece ferramentas para o melhor entendimento e programação de um script “.sql”.
 - Num arquivo de texto
Preferencialmente o programa “**Sublime Text**”, pois oferece a opção de escrever e salvar arquivos “.sql” com todas as ferramentas que envolve o processo.

3.4.3 Abrindo uma “nova consulta”

- Para começar a escrever um script no **SSMS** é necessario iniciar uma “nova consulta”, abrir uma pagina que serve para escrever os comandos SQL.
- As duas formas de iniciar essa pagina são:
 - Clickar em nova consulta, parte superior da pagina.
 - Atalho **CRTL + N**

3.5 Uso do delimitador GO

- O **SQL Server** funciona da seguinte forma com seus script's:
 - Não precisa do delimitador para compilar o código, *processamento assíncrono*.
 - Quando é pedido para compilar todo o script (sem seleções do código e sem uso de delimitador), o **SQL Server** executa o que for mais rápido primeiro, fora de ordem, por conta do *processamento assíncrono*.
 - Ao selecionar uma parte do código ele compila apenas aquela parte do código.
 - Usando o delimitador **GO** executar o código por partes.
- Delimitador **GO**
 - O uso do **GO** ao final de cada instrução serve como delimitador.
 - O **GO** quebra o código em pequenos pacotes que são enviados para o servidor executar.
 - Colocando o **GO** no código ao final de cada instrução, o servidor não faz o *processamento assíncrono*, assim quebrando o grande pacote que é o script inteiro, em pequenos pacotes para serem executados na ordem de envio.
- Modo de usar:
 - Colocar o **GO** ao final de cada instrução.
 - Inserir o **GO** na linha de baixo a instrução.
 - Sintaxe:

```
CREATE DATABASE nome_database
GO
USE nome_database
GO
CREATE TABLE nome_tabela(
campo tipo
)
GO
```


4 Modulo 25 - Arquitetura do SQL Server

4.1 Acessando arquivos de banco de dados

- Primeiro clicando com botão direito no banco de dados desejado.
 - Propriedades > Arquivos.

4.2 Arquitetura do SQL Server na maquina

- No Ubuntu os dados de arquitetura ficam gravados no caminho:
'/var/opt/mssql/data'
- No Windows os dados de arquitetura ficam gravados no caminho:
'C:\Program Files\Microsoft SQL Server\MSSQL15.SQLEXPRESS\MSSQL\DATA'

4.3 Tipos de arquivos

- **MDF** (*master data file*)
 - Armazena dados do sistema (dicionario de dados).
 - Criação automatica pelo sistema.
 - Recomenda-se que use o MDF apenas para dados do sistema (mudança manual).
 - Arquivos **MDF** sempre vão dentro do grupo **PRIMARY**.
- **LDF** (*log data file*)
 - Armazena log's, transações, conjuntos de instruções.
 - Criação automatica pelo sistema.
 - É apagado quando explicitado (**BEGIN**) a transação, ao finalizada com **COMMIT** (confirmando a transação) ou **ROLLBACK** (desfazendo a transação).
- **NDF** (*not master data file*)
 - Não é criado automaticamente pelo sistema (criação manual), diferente dos outros.
 - Utilizado para armazenar dados.
 - Podendo armazenar dados atraves de grupos dados (*GP*), para melhor organizar os dados, assim fazendo a separação dos dados por assunto.

- Possibilita a separação física dos dados em HD.

4.4 TRANSACTION - Transação

- É uma instrução que só executa as instruções dentro dela, no caso (**INSERT**, **UPDATE**, **DELETE**, ...), apenas se todas as instruções sejam concluídas com sucesso.
- Caso alguma instrução dentro dela dê ERRO, tudo é desfeito.
- Muito útil para fazer operações de transação financeira entre contas.
 - Exemplo de transação financeira, transferência de dinheiro entre contas:
 - * Subtrair dinheiro de uma conta.
 - * Somar dinheiro em outra conta.
- **COMMIT** ou **ROLLBACK**: Comandos que finalizam a transação onde o '**COMMIT**' confirma o conjunto de comandos e o '**ROLLBACK**' desfaz todo o processo executado pelo corpo de comandos caso tenha ocorrido algum evento contrário ao desejado.
- Sintaxe:
BEGIN TRANSACTION (ou **BEGIN**)
UPDATE tabela SET coluna1_a_modificar = expressão1
WHERE tabela IN (lista_dos_registros_a_modificar)
UPDATE tabela SET coluna2_a_modificar = expressão2
WHERE tabela IN (lista_dos_registros_a_modificar)
COMMIT (ou **ROLLBACK**)
- Observação: Pode usar **BEGIN TRANSACTION** ou apenas **BEGIN**.

4.5 Função ERRO

- No **SQL Server** temos uma função de sistema que faz a identificação de um erro dentro de uma transação chamada de '@@ERROR' função essa que por padrão recebe o valor 0 (zero) caso não ocorra nem um erro, no caso de algum erro ela assume o valor 1 (um).
- Uso da função '@@ERROR' dentro de um **IF**, para determinar uma *transação* (**TRANSACTION**) pode se mostrar uma boa solução.
- Sintaxe:
BEGIN TRANSACTION
UPDATE FROM *tabela*
SET *campo_1* = 10.000
WHERE *campo_1* < 50
IF @@ERROR = 0
COMMIT
ELSE
ROLLBACK
END

4.6 Criando Grupos de arquivos

4.6.1 Criando grupos de dados apartir de um novo banco de dados

- Clickar com o botão direito no “Banco de dados”, na aba “Pesquisador de objetos”.
- Opção “novo banco de dados”.
- Na aba “Geral” pode dar nome para o banco de dados criado.
- Na aba “Grupos de arquivos” é onde é criado os grupos de arquivos.
 - O botão “adicionar grupo de arquivos” cria um novo grupo de arquivo.
 - Podemos nomear esse novo grupo, a boa pratica indica sempre começar com o prefixo “GA_” (abreviação de “grupo de arquivo”), depois o nome grupo.
 - A opção “PADRÃO” indica que todos os arquivos não especificados o grupo, vai cair nesse grupo. Não deixar ele marcado em **PRIMARY**, pois esse grupo é para pertencer apenas os **MDF - dicionarios do sistema**.
- Criar arquivos de banco de dados **NDF**:
 - Na aba “Geral”, clickar no botão “adicionar” para criar um novo “arquivo de banco de dados”.
 - Em “Nome” nomear esse novo arquivo pela categoria (setor, ...).
 - Em “Nome do Arquivo” repetir o nome + o sufixo “.ndf”.
 - Em “Grupo de Arquivos” direcionar para o grupo desejado (criado anteriormente).
 - Em “Aumento Automático/Tamanho Máximo” podemos determinar o crescimento do banco de dados cada vez que ele atinge o limite, e determinar um tamanho máximo para o banco de dados (quando ele atinge o tamanho máximo, o banco de dados para).
- “OK” para confirmar as criações no final.

4.6.2 Criando grupos de dados em bancos de dados existentes

- Clickar com o botão direito no Banco de dados desejado, na aba “Pesquisador de objetos”.
- Na opção “Propriedades”.
- Na aba “Grupos de arquivos” é onde é criado os grupos de arquivos.
 - O botão “adicionar grupo de arquivos” cria um novo grupo de arquivo.
 - Podemos nomear esse novo grupo, a boa pratica indica sempre começar com o prefixo “GA_” (abreviação de “grupo de arquivo”), depois o nome grupo.
 - A opção “PADRÃO” indica que todos os arquivos não especificados o grupo, vai cair nesse grupo. Não deixar ele marcado em **PRIMARY**, pois esse grupo é para pertencer apenas os **MDF -**

dicionarios do sistema.

- Criar arquivos de banco de dados **NDF**:
 - Na aba “Arquivos”, clicar no botão “adicionar” para criar um novo “arquivo de banco de dados”.
 - Em “Nome” nomear esse novo arquivo pela categoria (setor, ...).
 - Em “Nome do Arquivo” repetir o nome + o sufixo “.ndf”.
 - Em “Grupo de Arquivos” direcionar para o grupo desejado (criado anteriormente).
 - Em “Aumento Automático/Tamanho Máximo” podemos determinar o crescimento do banco de dados cada vez que ele atinge o limite, e determinar um tamanho máximo para o banco de dados (quando ele atinge o tamanho máximo, o banco de dados para).
- “OK” para confirmar as criações no final.

4.7 Direcionando dados de tabela para grupos de arquivos (NDF)

- Clicar com o botão direito na tabela desejada.
- Na opção “design”.
- Na aba de “Propriedades”.
 - Dentro de “Identidade”, em “Nome” podemos alterar o nome da tabela.
 - Dentro de “Designer de tabela”, dentro de “Especificação de Espaço de Dados Regular”, em “Nome do Esquema de Partição ou Grupo de Arquivos” podemos selecionar um grupo de arquivos criado anteriormente para enviar a tabela.
 - Dentro de “Designer de tabela”, em “Grupo de Arquivos de Texto/Imagem” caso o banco de dados salve arquivos do tipo texto (.doc ou .odt) e imagem (pdf, jpeg ou .png) pode direcionar para ser salvo dentro de um grupo de arquivos criado anteriormente, podendo ser um grupo diferente do tópico anterior.

5 Tipos de dados

5.1 Numéricos exatos

- *bigint*
 - Inteiro, porem com espaço de armazenamento dobrado do *int*, 8bytes.
- *numeric*
 - *float* com números exatos, sem arredondamento.
 - Recebe como argumentos, tamanho total do numero e o numero de casas decimais.
 - Sintaxe:
numeric(n_total,n_deci)
- *bit*
 - Valores de cadeias de caracteres '**TRUE**' e '**FALSE**', podem ser convertidas em bit.
 - 1 (ou qualquer valor diferente de 0) é **TRUE**.
 - 0 é **FALSE**.
- *smallint*
 - Inteiro, porem com espaço de armazenamento pela metade do *int*, 2bytes.
- *decimal*
 - Igual a *numeric*.
- *int*
 - Número inteiro.
 - Espaço de armazenamento de 4bytes.
- *tinyint*
 - Inteiro num intervalo de 0 a 255.
 - Usado para armazenar idade de pessoas.
- *money*

- Apresenta valores monetários, duas casas decimais.
- Não é muito útil, melhor usar outro tipo em situação normal (*numeric* ou *float*).

5.2 Numéricos aproximados

- *float*
 - Dado do tipo “ponto flutuante”, são valores aproximados.
 - Varia entre precisão de 7 a 15 dígitos, dependendo do valor de “n” especificado (variando entre 1 a 24 para 7 dígitos e 25 a 53 para 15 dígitos.).
- *real*
 - Igual a *float*(24).

5.3 Data e hora

- *date*
 - Usado para armazenar datas.
 - No formato: “AAAA-MM-DD”.
- *datetimeoffset*
 - Com as vantagens do *datetime2*, somado ao reconhecimento de fuso horário com base no UTC.
 - No formato: “AAAA-MM-DD hh:mm:ss.nnnnn +/ -hh:mm”.
- *datetime2*
 - Apresenta um intervalo de datas maior que *datetime*.
 - Uma precisão fracionada padrão dos segundos maior que *datetime*.
 - Precisão opcional especificada pelo usuário. Podendo usar menos armazenamento.
- *smalldatetime*
 - Arredonda *datetime*.
 - Usa o mesmo formato que *datetime*.

- *datetime*
 - Usado para armazenar data e horário.
 - No formato: “AAAA-MM-DD hh:mm:ss.nnnnn”.
- *time*
 - Usado para armazenar horário.
 - No formato: “hh:mm:ss.nnnnn”.

5.4 Cadeias de caracteres

- *char*
 - Aceita somente determinado número de caracteres estabelecidos, tamanho fixo.
 - Sintaxe:
char(3)
- *varchar*
 - Aceita número de caracteres variável, determinar tamanho máximo.
 - Sintaxe:
varchar(50)
- *text*
 - *text* é usado para grandes pedaços de dados de *string*. Se o comprimento do texto exceder um determinado limite, o texto será armazenado fora da linha.
 - Usado se precisar armazenar textos grandes no seu banco de dados.
 - Sintaxe:
text (*tamanho_max_string*)

5.5 Cadeias de caracteres Unicode

- *nchar*(n)
 - O valor “n” é referente a quantidade de bytes necessarios para o armazenamento, de valor fixo.
- *nvarchar*(n)

- O valor “n” é referente a quantidade de bytes necessarios para o armazenamento, de valor variável.
- *ntext*
 - Usado se precisar armazenar textos grandes no seu banco de dados.
 - “n” é referente a quantidade de bytes necessarios para o armazenamento.

5.6 Cadeia de caracteres binária

- *binary*
 - Usado para armazenar valores binarios de valor fixo.
 - “n” é o numero de bytes.
- *varbinary*
 - Usado para armazenar valores binarios de valor variável.
 - “n” é o numero de bytes.
- *imagem*
 - Dados binarios de comprimento variavel.
 - Salva imagem.

5.7 Outros tipos de dados

- *cursor*
- *rowversion*
- *hierarchyid*
- *uniqueidentifier*
- *sql_variant*
- *xml*
- *table*

6 Modulo 26 PARTE 1 - Comandos básicos, CONSTRAINTS e descrição de tabelas

6.1 Comandos básicos

- **CREATE DATABASE**

- Criação de banco de dados.
- Sintaxe:
CREATE DATABASE *nome_database*
GO

- **DROP DATABASE**

- Apaga um banco de dados e tudo esta contido dentro.
- Sintaxe:
DROP DATABASE *nome_database*
GO

- **USE**

- Conectando a um banco de dados.
- Sintaxe:
USE *nome_database*
GO

- **CREATE TABLE**

- Criação de banco de dados.
- Sintaxe:
CREATE TABLE *nome_tabela*(
 coluna1 tipo regras,
 ...
)
GO

- **DROP TABLE**

- Apaga uma tabela.
- Sintaxe:
DROP TABLE *nome_da_tabela*
GO

- **ALTER TABLE**

- Adiciona regras (**CONSTRAINT**) a tabelas.
- É uma boa prática o uso de **ALTER TABLE** para normalizar o nome salvo das regras no sistema. Facilita a pesquisa posteriormente.
- Sintaxe:
ALTER TABLE *nome_tabela*
ADD CONSTRAINT *nome_regra*
[*regra a ser implementada*]
GO
- Observação: o *nome_regra* (nome da regra) é o nome que fica salvo no **dicionário de dados** (sistema).

- **INSERT**

- Inserindo novos registros numa tabela.
- No **SQL Server**, diferente do **MySQL**, nas colunas **PK (PRIMARY KEY)** com **IDENTITY** não precisa passar nenhum valor (nem **NULL**), o sistema já entende que vai haver preenchimento e incrementação automática dos valores.
- Os valores que vão ser inseridos do *tipo String (char, varchar, ...)* ou *DATE* devem ser passados entre aspas simples(*'valor'*).
- Sintaxe:
INSERT INTO *nome_tabela* **VALUES**
(*valor_coluna1, valor_coluna2, valor_coluna3, valor_coluna4, ...*), ...
(*valor_coluna1, valor_coluna2, valor_coluna3, valor_coluna4, ...*) **GO**

- **DELETE**

- Apaga os registros de uma tabela.
- Quando usado em conjunto com **WHERE**, apaga apenas os registros determinados por uma condição.
- Sintaxe:
DELETE FROM *nome_tabela*
WHERE *condição*
GO

- **UPDATE**

- Altera os registros de uma tabela.
- Quando usado em conjunto com **WHERE**, altera apenas determinados registros definidos por uma condição

– Sintaxe:
UPDATE *tabela_nome*
SET
coluna_1 = *valor_1*,
coluna_2 = *valor_2*,
...
WHERE *condição*
GO

6.2 Regras/Restrições - CONSTRAINTS

- Uma boa prática é criar as **CONSTRAINTS** por fora da tabela, para ter o controle dos nomes das *restrições* que ficaram salvas no dicionário de dados (sistema).

– Normalmente com uso de **ALTER TABLE**.

6.2.1 IDENTITY

- Exerce a mesma função que **AUTO_INCREMENT** no **MySQL**, incrementar automaticamente a coluna determinada.
- Trás de novo a opção de argumentos “**IDENTITY**(*1^o_n^o*,*2^o_n^o*)”:

– O primeiro número é onde começa.

– O segundo numero é quanto incrementa a cada vez.

- É possível suprimir os argumentos, onde “**IDENTITY** = **IDENTITY**(1,1)”.
- No **SQL Server**, diferente do **MySQL**, quando feito o **INSERT** de dados no campo onde tem **IDENTITY** não precisa entrar com valor nenhum (nem **NULL**), basta ignorar este campo, o **SQL Server** entende automaticamente que é para preencher ele.
- Sintaxe:
CREATE TABLE *nome_tabela*(
 coluna1 int PRIMARY KEY IDENTITY(100,10),
 ...
)
GO

- Observação: A *coluna1* começa em 100 e recebe o incremento de 10 em 10 a cada novo dado.

6.2.2 CONSTRAINTS

- **PRIMARY KEY (PK)**

- Toda tabela necessita de pelo menos um campo que identifique todo registro como sendo único (é o que chamamos de “*Chave Primaria*” ou “**Primary Key**”).
- Sintaxe:
ALTER TABLE *nome_tabela*
ADD CONSTRAINT PK_*nome_tabela*
PRIMARY KEY (*coluna*)
GO

- **FOREIGN KEY (FK)**

- Cria uma relação entre duas tabelas, através de uma *chave estrangeira* na tabela.
- Sintaxe:
ALTER TABLE *nome_tabela*
ADD CONSTRAINT FK_*tabela-recebe_tabelare-referenciada*
FOREIGN KEY (*coluna_FK*)
REFERENCES *tabela_referenciada*(*coluna_referenciada*)
GO
- Observações:
 - * A *tabela-recebe* é a tabela que vai receber a regra **FK**.
 - * A *coluna_FK* é a coluna específica na tabela, que recebe a regra, que vai servir para fazer a ligação (relação).
 - * As *tabela_referenciada* e *coluna_referenciada* é respectivamente referente a tabela e coluna que serão referenciadas pela ligação (relação) **FK**, ou seja, são as que não recebem a regra.

- **CHECK (CK)**

- Verifica (checa) se determinada coluna os valores dos dados são iguais aos especificados em uma lista.
- Um substituto no **SQL Server** ao **ENUM** no **MySQL**.
- Pode ser usado tanto na criação de tabela (**CREATE TABLE**) quanto na alteração de tabelas (**ALTER TABLE**).
- Sintaxe:
ALTER TABLE *nome_tabela*
ADD CONSTRAINT CK_*nome_tabela*
CHECK (*nome_coluna* **IN** (*‘valor1’,‘valor2’*))
GO

- **UNIQUE (UQ)**

- A restrição **UNIQUE** garante que todos os valores em uma coluna sejam diferentes.
- Sintaxe:
ALTER TABLE *nome_tabela*
ADD CONSTRAINT UQ_*nome_tabela*
UNIQUE (*coluna_recebe_UQ*)
GO

6.3 Comandos de descrição tabelas - SP__

- No **SQL Server** a descrição de uma tabela é através de **PROCEDURES** (funções).
- **PROCEDURES** já criadas e armazenadas no sistema, “**STORAGE PROCEDURES**” (**SP**).

6.3.1 SP_COLUMNS

- **SP_COLUMNS** é igual a **DESC**, no **MySQL**.
- Faz uma descrição da tabela:
 - Nome das colunas
 - *Tipo* de cada coluna
 - *Regras* em cada coluna
 - ...
- Sintaxe:
SP_COLUMNS *nome_tabela*
GO

6.3.2 SP_HELP

- **SP_HELP** é igual ao **SHOW CREATE TABLE**, no **MySQL**.
- Faz uma descrição mais detalha da tabela que **SP_COLUMNS**:
 - Quem criou a tabela.
 - Permissões.
 - Datas importantes (criação e modificação).
 - ...
- Sintaxe:
SP_HELP *nome_tabela*
GO

6.3.3 SP_HELPTEXT

- Exibe a definição de uma regra definida pelo usuário.

- **TRIGGERS**
 - **PROCEDURES**
 - **VIEWS**
 - **FUNCTIONS**
- É utilizada para podermos visualizar códigos criados dentro da base de dados.
 - O código é exibido no formato, cada linha é um registro.
 - Sintaxe:


```
SP_HELPTEXT '[schema.]nome_objeto'
```

```
GO
```

 - Objeto é o nome da regra definida pelo usuário (**TRIGGER, PROCEDURES,VIEWS, FUNCTIONS, ...**).
 - Colocar entre aspas simples o nome do objeto.

7 Modulo 26 PARTE 2 - Funções, Projeções, Seleções e Junções

7.1 Funções

7.1.1 Funções usuais

- **ISNULL()**

- Trata os valores **nulos**, na coluna especificada, na consulta.
- Equivalente do **IFNULL()** do **MySQL**.
- Dentro do **ISNULL()**, os argumentos são:
 - * *Nome da coluna* a ser avaliada.
 - * *Texto* se o valor for **nulo**.
- Dentro do **ISNULL()** usar aspas simples ('').
- Sintaxe:
SELECT
A.coluna1, ISNULL(T.coluna2, 'SEM') AS "alias1",
ISNULL(T.coluna3, 'SEM_NUMERO') AS "alias2", FROM tabela1 A
LEFT JOIN tabela3 T
ON A.colunaPK = T.colunaFK
GO

- **PRINT 'mensagem'**

- Imprime na tela uma mensagem, colocado entre aspas simples ('').
- Pode ser usado dentro de **TRIGGERS** e **PROCEDURES** para passar alguma informação importante ao usuário.
- Sintaxe com exemplo:
CREATE TRIGGER nome_da_trigger
ON DBO.tabela
FOR UPDATE
AS
...
PRINT 'TRIGGER EXECUTADO COM SUCESSO'
GO

7.1.2 Funções de auditoria

- **SUSER_NAME()**

- Função que retorna o usuario logado no banco de dados no momento.
- Útil para usar dentro de **TRIGGERS** para salvar o usuario reponsavel por alguma alteração numa tabela (auditoria).
- Sintaxe:
SELECT SUSER_NAME()
GO

- **GETDATE()**

- Pega a data no sistema (data e horário).
- Util para usar dentro de **TRIGGERS** para salvar a data e horario de alguma alteração numa tabela (auditoria).
- Formato:
“aaaa-mm-dd hh:mm:ss.mmm”
- Sintaxe:
SELECT GETDATE()
GO

7.1.3 Funções de datas

- **GETDATE()**

- Pega a data no sistema (data e horário).
- Formato:
“aaaa-mm-dd hh:mm:ss.mmm”

- **DATEDIFF()**

- Calcula a diferença entre duas datas.
- * Retorna um valor inteiro (*INT*), dia (*DAY*), ou mês (*MONTH*), ou ano (*YEAR*), ou dia da semana (*WEEKDAY*).
- * Sintaxe:
DATEDIFF(*intervalo*, *data_inicio*, *data_termino*)
 - *intervalo*, indica a função com que parametro estou trabalhando (*day*, *month*, *year*, *weekday*)
- Outras funções podem se usadas em conjunto, como parametros, para ajudar a fazer os cálculos.
- * Comummente usada em conjunto com **GETDATE()** para calcular idade.
- * Sintaxe:
DATEDIFF(*intervalo*, *data_inicio*, **GETDATE**())
 - a função **GETDATE()**, data atual, entra no lugar do parametro *data de termino*, calculando assim a idade atual.

- **DATENAME()**

- Retorna o nome da parte da data em questão. (ex.: nome do mês)
- Retorna uma *string*.
- Sintaxe:
DATENAME(*intervalo*, *data*)
 - * *intervalo*, indica a função com que parametro estou trabalhando (*day*, *month*, *year*, *weekday*)

- **DATEPART()**

- Função parecida com **DATENAME()**, porem retorna um inteiro (*INT*).
- Retorna uma parte da data.

- Sintaxe:

DATEPART(*intervalo*, *data*)

* *intervalo*, indica a função com que parametro estou trabalhando (*day*, *month*, *year*, *weekday*)

- **DATEADD()**

- Retorna uma data somada a outra data.

- Sintaxe:

DATEADD (*intervalo*, *incremento_INT*, *data*)

* *intervalo*, indica a função com que parametro estou trabalhando (*day*, *month*, *year*, *weekday*)

* *incremento_INT*, com base no parametro informado pelo *intervalo*, quanto deve ser somado (valor *INT*).

* *data*, data a ser incrementada.

- **DAY()**

- Recebe como argumento uma data, formato do sistema.

- Retorna o dia (**DAY**) contido na data.

- Sintaxe:

DAY(*data*)

- **MONTH()**

- Recebe como argumento uma data, formato do sistema.

- Retorna o mês (**MONTH**) contido na data.

- Sintaxe:

MONTH(*data*)

- **YEAR()**

- Recebe como argumento uma data, formato do sistema.

- Retorna o ano (**YEAR**) contido na data.

- Sintaxe:

YEAR(*data*)

7.2 Projeção, seleção e Junção - SELECT, WHERE e JOIN

Principais passos de uma consulta.

7.2.1 PROJEÇÃO

- O primeiro passo de uma consulta é montar o que quer ver na tela - **SELECT**.
- É tudo que você quer ver na tela.
- Sintaxe comentada:
SELECT *coluna_1* (PROJEÇÃO)
FROM *tabela* (ORIGEM)
GO
ou
SELECT 2+2 **AS** *alias* (PROJEÇÃO)
GO
- É possível mostrar mais de uma consulta ao mesmo tempo.
 - Sintaxe comentada:
SELECT * **FROM** *tabela_1* (PROJEÇÃO 1) **SELECT** * **FROM** *tabela_2* (PROJEÇÃO 2)
GO

Obs.: o que está entre parênteses é comentário.

7.2.2 SELEÇÃO

- O segundo passo de uma consulta é a seleção dos dados de uma consulta - **WHERE**.
- É filtrar.
- Trazer um subconjunto do conjunto total de registros de uma tabela.
- Sintaxe comentada:
SELECT *coluna_1, coluna_2, coluna_3* (PROJEÇÃO)
FROM *tabela* (ORIGEM)
WHERE *critero = valor_do_critério* (SELEÇÃO)
GO

Obs.: o que esta entre parênteses é comentario.

7.2.3 JUNÇÃO

7.2.3.1 Junção forma errada - gambiarra

- Usa seleção como uma forma de juntar tabelas.
- Como consequencia:
 - Uso de operadores lógicos para mais criterios de seleção - **WHERE**.
 - Ineficiencia na pesquisa, maior custo computacional.

- Sintaxe comentada:

```
SELECT coluna1_tab1, coluna2_tab1, coluna1_tab2 (PROJEÇÃO)  
FROM tabela1, tabela2 (ORIGENS)  
WHERE chave_primaria_tab1 = chave_estrangeira_tab2 (JUNÇÃO)  
GO
```

ou

```
SELECT coluna1_tab1, coluna2_tab1, coluna1_tab2 (PROJEÇÃO)  
FROM tabela1, tabela2 (ORIGENS)  
WHERE chave_primaria_tab1 = chave_estrangeira_tab2 (JUNÇÃO)  
AND criterio = valor (SELEÇÃO com operador lógico)  
GO
```

Obs.: o que esta entre parênteses é comentario.

7.2.3.2 Junção forma certa - JOIN

- Junção **JOIN**, junta duas ou mais tabelas apartir das colunas de *chaves primarias* e *chaves estrangeiras*.
- Admite seleção - **WHERE** - sem maiores custos computacionais.

1. INNER

- Exclui os registros sem par (orfans) na outra tabela - **INNER**.
- Consulta com duas tabelas.

- Sintaxe comentada:

```
SELECT coluna1_tab1, coluna2_tab1, coluna1_tab2 (PROJEÇÃO)  
FROM tabela1 (ORIGEM)  
INNER JOIN tabela2 (JUNÇÃO)  
ON chave_primaria_tab1 = chave_estrangeira_tab2  
WHERE criterio = valor (SELEÇÃO)  
GO
```


2. LEFT

- Mostra ate os registros sem par (nulos) - **LEFT**.
 - Comum usar a função *ISNULL()* para tratar os valores nulos.
- Consulta com duas tabelas.
 - Sintaxe comentada:
SELECT *coluna1_tab1, coluna2_tab1, coluna1_tab2* (PROJEÇÃO)
FROM *tabela1* (ORIGEM)
LEFT JOIN *tabela2* (JUNÇÃO)
ON *chave_primaria_tab1 = chave_estrangeira_tab2*
WHERE *critério = valor* (SELEÇÃO)
GO

7.2.3.3 Cláusulas ambíguas e Ponteiramento

- Consulta com mais de duas tabelas.
 - Pode apresentar colunas/campos com o mesmo nome, de tabelas diferentes. Caso comum das *chaves estrangeiras* (**FK**).
 - Indicar de onde vem cada coluna através de “*nome_da_tabela.nome_da_coluna*”.
 - Sintaxe comentada:
SELECT
tabela1.coluna1_tab1,
tabela1.coluna2_tab1,
tabela2.coluna1_tab2,
tabela3.coluna1_tab3 (PROJEÇÃO)
FROM *tabela1* (ORIGEM)
LEFT JOIN *tabela2* (JUNÇÃO)
ON *tabela1.chave_primaria_tab1 = tabela2.chave_estrangeira_tab2*
INNER JOIN *tabela3* (JUNÇÃO)
ON *tabela1.chave_primaria_tab1 = tabela3.chave_estrangeira_tab3*
WHERE *critério = valor* (SELEÇÃO)
GO
Obs.: o que esta entre parênteses é comentario.

- Ponteiramento (alias para tabelas)
 - Melhora a performance da consulta.
 - Sintaxe comentada:
SELECT
A.coluna1_tab1,
A.coluna2_tab1,
B.coluna1_tab2,
C.coluna1_tab3
FROM *tabela1 A* (PONTEIRAMENTO DA TABELA 1)
LEFT JOIN *tabela2 B* (PONTEIRAMENTO DA TABELA 2)
ON *A.chave_primaria_tab1 = B.chave_estrangeira_tab2*
INNER JOIN *tabela3 C* (PONTEIRAMENTO DA TABELA 3)
ON *A.chave_primaria_tab1 = C.chave_estrangeira_tab3*
WHERE *critério = valor*
GO

8 Modulo 26 PARTE 3 - Conversão de tipo de dados

8.1 Conversão de dados automatica pelo sistema

- Conversões de *TIPO* que o **SQL Serve** faz automaticamente pelo sistema.
- Existe um direcionamento em que o sistema costuma forçar de **STRING** para **INT**, nunca o contrario de maneira automatica.
- O simbolo '+', alem de operador matemático, funciona como concatenador.

– Sintaxe:

```
SELECT '1' + '1'
```

```
GO
```

(Retorno '11')

8.2 Tabela de conversões de dados automatico pelo sistema

From \ To	binary	varbinary	char	nchar	varchar	nvarchar	datetime	smalldatetime	date	time	datetimeoffset	datetime2	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml	CLR UDT	hierarchyid
binary		●	●	●	●	●	●	●	●	●	●	●	●	●	✗	✗	●	●	●	●	●	●	●	●	●	●	✗	✗	●	●	●	●
varbinary	●		●	●	●	●	●	●	●	●	●	●	●	●	✗	✗	●	●	●	●	●	●	●	●	●	●	✗	✗	●	●	●	●
char	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
varchar	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
nchar	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
nvarchar	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
datetime	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
smalldatetime	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
date	●	●	●	●	●	●	●	●		✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
time	●	●	●	●	●	●	●	●	✗		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
datetimeoffset	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
datetime2	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
decimal	●	●	●	●	●	●	●	●	●	●	●	●	◆	◆	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
numeric	●	●	●	●	●	●	●	●	●	●	●	●	◆	◆	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
float	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
real	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
bigint	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
int(INT4)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
smallint(INT2)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
tinyint(INT1)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
money	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
smallmoney	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
bit	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
timestamp	●	●	●	●	✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
uniqueidentifier	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
image	●	●	✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
ntext	✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
text	✗	✗	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
sql_variant	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
xml	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○
CLR UDT	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
hierarchyid	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Explicit conversion

Implicit conversion

Conversion not allowed

Requires explicit CAST to prevent the loss of precision or scale that might occur in an implicit conversion.

Implicit conversions between xml data types are supported only if the source or target is untyped xml. Otherwise, the conversion must be explicit.

- *Conversão implícita.*
 - São conversões automáticas pelo sistema.
- *Conversão explícita.*
 - São conversões por meio de funções. (Ex.: **CAST()**)

8.3 Funções de conversão

- **CAST()**

- A função **CAST()** converte um valor (de qualquer tipo) em um tipo de dados especificado.
- O tipo de dados para converter a expressão. Pode ser um dos seguintes:

- * **BIGINT**
- * **INT**
- * **SMALLINT**
- * **TINYINT**
- * **BIT**
- * **DECIMAL**
- * **NUMERIC**
- * **MONEY**
- * **SMALLMONEY**
- * **FLOAT**
- * **REAL**
- * **DATETIME**
- * **SMALLDATETIME**
- * **CHAR**
- * **VARCHAR**
- * **TEXT**
- * **NCHAR**
- * **NVARCHAR**
- * **NTEXT**
- * **BINARY**
- * **VARBINARY**
- * **IMAGE**

- *Expressões* do tipo **STRING** devem entrar entre aspas simples (').
- Sintaxe:
SELECT
CAST(*expressão* **AS** *TIPO_especificado*)
GO

- **CHARINDEX()**

- Retorna um numero inteiro de acordo com a posição de determinada caracter num **VARCHAR**.
 - * As posições no VARCHAR começam a ser contadas a partir da posição 1.
 - * O retorno 0, é caso não tenha achado nenhum caracter procurado.
- Os argumento do CHARINDEX(*o que procurar?*, *onde procurar?*, *a partir de tal posição?*)
 - * *O que procurar?*
O caracter que deve ser encontrado.
 - * *onde procurar?*
O **VARCHAR** que deve ser percorrido procurando o caracter.
 - * *a partir de tal posição?*
A partir de qual posição a busca deve começar. As posições do VARCHAR começam a ser contadas a partir da posição 1.
Pode omitir esse ultimo argumento, a função entenderá como começando da posição 1 (a inicial).
- Sintaxe:


```
SELECT
CHARINDEX('character',string, numero_da_posição_inicial_procura) AS 'alias'
FROM tabela
GO
```

9 Modulo 26 PARTE 4 - Importação de arquivo de dados

9.1 Aspectos importantes da importação de Arquivos

- Além da função de importação de arquivo (**BULK INSERT**), é necessário antes, montar uma estrutura preparada para receber os dados do arquivo (criação de **BANCO DE DADOS** e **TABELAS** para receber os dados).

– Sintaxe:

```
CREATE DATABASE nome_database
GO
CREATE TABLE tabela(
    campo1 tipo regra,
    campo2 tipo regra,
    ...
)
GO
```

- Outro aspecto importante é como está organizado os dados no arquivo importados.
 - A organização dos dados, no arquivo, interfere diretamente no processo de importação do arquivo.
 - Partes em branco, dentro do arquivo, provavelmente resultarão em registros *nulos* (**NULL**), quando não em erro.
 - É importante para importação conhecer os caracteres de comando da tabela **ASCII**, são necessários como argumentos da função **BULK INSERT**.

##	Nome_na_ASC	Descricao	Representacao_em_C
## 1	nul	null byte/byte nulo	\\0*
## 2	bel	bell character/apito	\\a
## 3	bs	backspace	\\b
## 4	ht	horizontal tab/tabulação	\\t
## 5	np	formfeed/fim da pagina	\\f
## 6	nl	newline/nova linha	\\n
## 7	cr	carriage return	\\r
## 8	vt	vertical tab	\\v

* É uma barra invertida só.

9.2 Função de importação de arquivos BULK INSERT

- A função **BULK INSERT** serve para importação dos dados, de um arquivo qualquer, para dentro do **SQL Server**.
- Antes de qualquer coisa, deve ser criado anteriormente um estrutura para receber esses dados no **SQL Server**, ou seja, a criação do banco de dados e da tabela que vai receber esses dados.

- Sintaxe:

```
BULK INSERT tabela_importação  
FROM 'caminho'  
WITH(  
  FIRSTROW = 2,  
  DATAFILETYPE = 'char',  
  FIELDTERMINATOR = '\t',  
  ROWTERMINATOR = '\n'  
)  
GO
```

- Argumentos do **BULK INSERT**:

- *tabela_importação*

- A tabela a qual os dados importados serão direcionados.

- *caminho*

- O caminho no sistema do computador onde o arquivo está localizado. O caminho é colocado entre aspas simples, pois é uma **string**.

- Ex.: 'C:/SPB_Data/github_bkp/SQL-Server/Arquivos_importacao/CONTAS.txt'

- Argumentos do **WITH**:

- **FIRSTROW**

- É um número inteiro que indica a partir de qual linha começa os dados, começando na linha 1. Normalmente exclui-se o cabeçalho, começando assim a partir da linha 2, ou seja o valor 2.

- **DATAFILETYPE**

- Tipo do arquivo do arquivo (dados).

- **FIELDTERMINATOR**

- Determina onde termina cada dado.

- Usar o caractere de comando da tabela ASCII. Ou o que seja que faça a separação dos dados no arquivo, muito comum o uso do “;”.

- O caractere de comando é entre aspas simples ''.

- **ROWTERMINATOR**

- Determina onde termina cada registro/linha.

- Usar o caractere de comando da tabela ASCII.

- O caractere de comando é entre aspas simples ''.

10 Modulo 26 PARTE 5 - Técnica de “flag-ar” coluna (SELECT)

- Técnica usada para criar, numa consulta (**SELECT**), uma espécie de tabela verdade com os resultados possíveis de uma coluna.
- Essa técnica se baseia no uso da função **CHARINDEX()** para achar determinados resultados e a partir dele criar novas colunas, na consulta (**SELECT**).
- Sendo cada nova coluna, um dos resultados possíveis.
- E os resultados são valores de “0” ou “1”, em cada coluna nova.
 - Resultado “0”, na nova coluna, significa que a consulta daquele dado, no registro correspondente, não corresponde aquele resultado.
 - Resultado “1”, na nova coluna, significa que a consulta daquele dado, no registro correspondente, corresponde aquele resultado.
- Outra possibilidade de continuação da técnica é a partir dessas novas colunas, criar um multiplicador para interagir com os dados e transformar ele.
- Sintaxe exemplo, técnica em duas partes:

– Parte 1:

```
SELECT  
  CONTA,  
  VALOR,  
  DEB_CRED,  
  CHARINDEX('D',DEB_CRED) AS DEBITO,  
  CHARINDEX('C',DEB_CRED) AS CREDITO,  
  ((CHARINDEX('C',DEB_CRED)*2)-1) AS MULTIPLICADOR  
FROM LANCAMENTO_CONTABIL  
GO
```

* Cria duas colunas *DEBITO* e *CREDITO*.

* Na nova coluna *CREDITO*, se na coluna *DEB_CRED* o valor é credito ('C') a coluna leva “1”, senão leva “0”.

* Na nova coluna *DEBITO*, se na coluna *DEB_CRED* o valor é debito ('D') a coluna leva “1”, senão leva “0”.

* Por último, cria uma coluna *MULTIPLICADOR*, onde se é credito leva “1”, se é debito leva “-1”.

– Parte 2:

```
SELECT  
  CONTA,  
  SUM((VALOR*(CHARINDEX('C',DEB_CRED)*2)-1)) AS SALDO  
FROM LANCAMENTO_CONTABIL  
GROUP BY CONTA  
ORDER BY CONTA
```

GO

- * Cria uma coluna *SALDO* que é a soma dos créditos e debitos.
- * Sendo debito negativo e crédito positivo.
- * agrupando os dados pela coluna *CONTA* e ordenando pela coluna *CONTA*.

11 Modulo 27 PARTE 1 - TRIGGER (Gatilho) DML (Data Manipulation Language)

- A **TRIGGER** é um gatilho de programação, que dispara toda vez que algo predeterminado acontecer.
- Exemplos de gatilhos disparadores de uma **TRIGGER** são:
 - **INSERT**
 - **UPDATE**
 - **DELETE**
- Após os gatilhos (**TRIGGERS**) disparados, são executados blocos de programação.
- Os comandos a seguir são de *criação* (**CREATE TRIGGER**), *modificação* (**ALTER TRIGGER**) e *pagar* (**DROP TRIGGER**) **TRIGGERS**.
- É uso comum do **TRIGGER** salvar modificação de dados (**INSERT**, **UPDATE** e **DELETE**) em uma tabela que sirva de *backup* dos dados, e/ou uma tabela que sirva para auditoria das modificações desses dados.
 - Logo, se faz necessario preparar antes da criação do **TRIGGER** (**CREATE TRIGGER**), a tabela para receber os dados enviados pelo **TRIGGER** (**CREATE TABLE**).

11.1 Principais fatores a serem auditados por um TRIGGER

- Usuário responsável pela modificação.
SUSER_NAME()
- Data da modificação.
GETDATE()
- *STRING* (ou *CHAR*) que define o tipo de ação de modificação executada.
 - **SET** @ACAO = 'VALOR MODIFICADO *DML* PELA TRIGGER *nome_TRIGGER* [*NA nome_coluna*]'
 - **SET** @ACAO = 'INS/UPD/DEL'
- Salvar dados do registro/linha que não foram alterados.
- Caso **INSERT**:
 - Dados inseridos na tabela, **INSERTED**.
SELECT @variável_recebe_select = coluna_tabela **FROM** INSERTED
- Caso **UPDATE**:
 - Dados antigos (antes da modificação), **DELETED**.
SELECT @variável_recebe_select = coluna_tabela **FROM** DELETED
 - Dados novos (depois da modificação), **INSERTED**.
SELECT @variável_recebe_select = coluna_tabela **FROM** INSERTED
- Caso **DELETE**:
 - Dados apagados da tabela (antes do apagamento), **DELETED**.
SELECT @variável_recebe_select = coluna_tabela **FROM** DELETED

11.2 Conceitos Préliminares - Argumentos temporais (INSERTED/DELETED) e Declaração de variáveis (DECLARE)

11.2.1 Argumentos temporais - INSERTED e DELETED

- São áreas do sistema que guardam dados.
- Comparando com MySQL:
 - **INSERTED** = **AFTER** (depois)
 - **DELETED** = **BEFORE** (antes)

11.2.1.1 INSERTED

- A área **INSERTED** guarda os dados novos inseridos, ou seja, ao usar o **INSERTED** pega os novos dados (“depois” de) inseridos na tabela.
- Os **DML** que se valem desse artifício, normalmente, são:
 - **INSERT**
Usado normalmente para guardar os novos dados inseridos na tabela, guarda os dados “depois” de inseridos.
 - **UPDATE**
Usado para guardar os novos dados modificados na tabela, guarda os dados “depois” de modificados.

11.2.1.2 DELETED

- A área **DELETED** guarda os dados antigos, ou seja, ao usar o **DELETED** pega os antigos dados (“antes” de) modificados na tabela.
- Os **DML** que se valem desse artifício, normalmente, são:
 - **DELETE**
Usado normalmente para guardar os antigos dados, “antes” de deletados da tabela.
 - **UPDATE**
Usado para guardar os antigos dados modificados na tabela, guarda os dados “antes” de modifica-los.

11.2.2 Declaração de variáveis - DECLARE

- As variáveis podem ser declaradas no corpo de uma **PROCEDURE**, ou **TRIGGER**, com a instrução **DECLARE**.
- Os nomes de variáveis devem começar com uma arroba '@', para caracterizar uma variável (regra de identificação), um arroba '@' para variável local e dois arroba '@@' para variáveis globais.
- Para declara um variável, basta usar o **DECLARE**, variáveis não são restritas ao uso apenas dentro do corpo de algum **PROCEDURE**, ou **TRIGGER**. Mas neste capítulo abortaremos variáveis locais dentro de **TRIGGERS**.
- Valores são atribuídos as variáveis com as instrução **SELECT** ou **SET**.
 - Sendo **SELECT**, valores vindos de tabelas.
 - Sendo **SET**, valores vindos de funções, ou valores literais.
- Depois da declaração, todas as variáveis são inicializada com **NULL**, a menos que um valor seja fornecido como parte da declaração.
- Sintaxe:
DECLARE @variavel_local tipo (= valor_inicialização)
- É possível declarar varias variáveis ao mesmo tempo, com apenas um **DECLARE**.
 - Sintaxe:
DECLARE @nome_variavel_local1 tipo, @nome_variavel_local2 tipo, ...

11.3 CREATE TRIGGER

- Comando usado para criação de **TRIGGER**.
- Argumentos do **TRIGGER**:
 - *nome_TRIGGER* é o nome do **TRIGGER** que deseja criar, ver “boas praticas - Nome do **TRIGGER**”.
 - **ON** especifica a tabela ou **VIEW** na qual o gatilho (**TRIGGER**) é criado.
 - **FOR** indica quando o gatilho deve ser acionado quando um evento acontece (**INSERT**, **UPDATE** ou **DELETE**).
- *DML (Data Manipulation Language - Linguagem de Manipulação de Dados)*, no contexto que é usado no **TRIGGER**, pode ser substituído por algum comando como **INSERT**, **UPDATE** ou **DELETE**. Definindo assim qual o comando que dispara o **TRIGGER** (gatilho).

11.3.1 TRIGGER de modificação especificada da tabela

- Sintaxe:
CREATE TRIGGER *nome_TRIGGER*
ON *DBO.tabela*
FOR DML
AS
 - Primeiro Bloco - declaração de variáveis (**DECLARE**)
DECLARE *@variavel_local tipo (= valor_inicialização)*
 - Segundo Bloco - Atribuindo valor em variáveis via **SELECT**
SELECT *@variável_recebe_select = coluna_tabela* **FROM** **INSERTED** (ou **DELETED**)
 - Terceiro Bloco - Atribuindo valor em variáveis via funções
SET *@variável_função = GETDATE()*
SET *@variável_mensagem = 'VALOR INSERIDO PELA TRIGGER TRG_ATUALIZA_PRECO'*
 - Quarto Bloco - **INSERT** dados na tabela do **TRIGGER**
INSERT INTO *tabela_auditoria*
(*PRODUTO, CATEGORIA, PRECOANTIGO, PRECONOVO, DATA, USUARIO, MENSAGEM*)
VALUES
(*@PRODUTO, @CATEGORIA, @PRECO, @PRECONOVO, @DATA, @USUARIO, @ACAO*)
 - Bloco de mensagem ao usuário - **PRINT**
PRINT *'TRIGGER EXECUTADO COM SUCESSO'*

GO

11.3.2 TRIGGER olhando uma modificação específica de uma coluna

- O comando “**IF DML(coluna)**”, seguido de **BEGIN bloco de comandos END**, define qual comando *DML* em respeito a uma coluna, em especificada, dispara o **TRIGGER**.
- Ou seja, não dispara o **TRIGGER** em qualquer coluna, mesmo que seja o *DML* especificado.
- Lembrar de ao usar **IF**, indentar depois do **BEGIN**. E terminar o **IF** com **END**.

- Sintaxe:

```
CREATE TRIGGER nome_TRIGGER  
ON DBO.tabela  
FOR DML AS  
IF DML(coluna)  
BEGIN
```

- Primeiro Bloco - declaração de variáveis (**DECLARE**)
DECLARE @variavel_local tipo (= valor_inicialização)
- Segundo Bloco - Atribuindo valor em variáveis via **SELECT**
SELECT @variável_recebe_select = coluna_tabela **FROM** INSERTED (ou **DELETED**)
- Terceiro Bloco - Atribuindo valor em variáveis via funções
SET @variável_função = **GETDATE**()
SET @variável_mensagem = ‘VALOR INSERIDO PELA TRIGGER TRG_ATUALIZA_PRECO’
- Quarto Bloco - INSERT dados na tabela do **TRIGGER**
INSERT INTO tabela_auditoria
(PRODUTO, CATEGORIA, PRECOANTIGO, PRECONOVO, DATA, USUARIO, MENSAGEM)
VALUES
(@PRODUTO, @CATEGORIA, @PRECO, @PRECONOVO, @DATA, @USUARIO, @ACAO)
- Bloco de mensagem ao usuário - **PRINT**
PRINT ‘TRIGGER EXECUTADO COM SUCESSO’

```
END  
GO
```


11.4 ALTER TRIGGER

- Modifica um **TRIGGER** existente.
- Argumentos do **TRIGGER**:
 - *nome_TRIGGER* é o nome do TRIGGER que deseja alterar.
 - **ON** especifica a tabela ou **VIEW** na qual o gatilho (**TRIGGER**) é criado.
 - **FOR** indica quando o gatilho deve ser acionado quando um evento acontece (**INSERT**, **UPDATE** ou **DELETE**).
- Sintaxe:
ALTER TRIGGER *nome_TRIGGER*
ON *DBO.tabela*
FOR DML AS
[Comandos SQL]
GO

11.5 DROP TRIGGER

- Apaga/deleta o **TRIGGER**.
- Sintaxe:
DROP TRIGGER *nome_TRIGGER*
GO
ou
DROP TRIGGER *dbo.nome_TRIGGER*
GO

11.6 Boas Práticas

11.6.1 Nome do TRIGGER

- Para normalizar os nomes, diferenciando dos demais, é indicado usar:
“TR_”(ou “TRG_”)+*DML*+“_”+*tabela/campo_ou_coluna*
- *DML*:
 - **INSERT**
 - **UPDATE**
 - **DELETE**

11.6.2 Blocos de Programação

- São os blocos de programação (instruções **SQL**) dentro do **TRIGGER**.

11.6.2.1 Primeiro Bloco - declaração de variáveis (DECLARE)

- Espaço usado para declarar todas as variáveis que serão usadas dentro do **TRIGGER**.
- Sintaxe:
DECLARE @nome_variavel1 *tipo*
DECLARE @nome_variavel2 *tipo*
...

11.6.2.2 Segundo Bloco - Atribuindo valor em variáveis via **SELECT**

- Insere nas variáveis valores vindos de tabelas, que são inseridos pelo comando “**SELECT**”.
- Sintaxe:
SELECT @nome_variavel1 = nome_coluna **FROM** INSERTED (ou **DELETED**)
...

11.6.2.3 Terceiro Bloco - Atribuindo valor em variáveis via funções

- Insere nas variáveis valores vindos de funções ou literais, que são inseridos pelo comando “**SET**”.
- Sintaxe:
SET @nome_variavel1 = função_qualquer()
SET @nome_variavel2 = 'texto'
...
Obs.: *texto* inserido dentro de variável através de aspas simples.

11.6.2.4 Quarto Bloco - INSERT dados na tabela do TRIGGER

- Inserindo dados das variáveis na tabela de armazenamento do **TRIGGER**, tabela normalmente de auditoria dos dados.
- Sintaxe:
INSERT INTO *tabela_auditoria*
(*coluna1, coluna2, coluna3, coluna4, coluna5, coluna6, coluna7*)
VALUES
(*@nome_variavel1, @nome_variavel2, @nome_variavel3, @nome_variavel4, @nome_variavel5,*
@nome_variavel6, @nome_variavel7)

11.6.2.5 Mensagem ao usuário - PRINT

- A função **PRINT** imprime uma mensagem na tela.
- Este é o espaço no bloco de programação para deixar alguma mensagem para o usuário ao final da aplicação da **TRIGGER**.
- A mensagem a ser passada pela função **PRINT**, fica entre aspas simples(''), aspas duplas retorna **ERRO**.
- Sintaxe:
PRINT '*TRIGGER EXECUTADO COM SUCESSO*'

12 Modulo 27 PARTE 2 - Simplificando TRIGGER (Gatilho) e Bloco Anônimo

12.1 Bloco Anônimo

- Quando não tem cabeçalho de nomeação de bloco (anônimo), logo não persiste no sistema (não é salvo no sistema).
- Será executado apenas uma vez.
- Muito utilizado para testes, justamente por ser executado apenas uma vez.
- No caso de **TRIGGER**, se assemelha ao bloco de programação do **TRIGGER**, sem o cabeçalho (sem a nomeação do **TRIGGER**).
 - Por conta disto, é usado como teste para **TRIGGER**.
 - Ou um modelo de **TRIGGER** simplificado.
- Sintaxe (exemplo):

```
DECLARE @variavel INT
SET @variavel = ((SELECT 50 + 50))
INSERT INTO tabela VALUES (@variavel)
PRINT 'VALOR INSERIDO ' + CAST(@variavel AS VARCHAR)
GO
```

 - O uso da função **PRINT** imprime a mensagem e o valor da variavel na tela.
 - O uso da função **CAST** para modar o tipo da variavel *@variavel* PARA *VARCHAR* e imprimir na tela junto da mensagem.
- No caso de armazenar valores atraves da instrução **SELECT**, lembrar de usar parenteses duas vezes.

12.2 Simplificando TRIGGERS

12.2.1 TRIGGERS simplificados

- A simplificação da **TRIGGER** tem haver com o uso menos formal da estruturação, e deixar mais parecido com *bloco anônimo*, porem com cabeçalho (para que possa ser guardado pelo sistema).
- Outra diferença é o uso do **SELECT** e/ou **SET**, dentro de um **INSERT**, mas principalmente o **SELECT**, para fazer uma consulta que já inserindo os dados na tabela do **TRIGGER**.
- O uso de ponteiramento é uma ferramenta poderosa para esse tipo de **TRIGGER** (discutido na próxima seção).

- Sintaxe:

```
CREATE TRIGGER TRG_nome  
ON DBO.tabela  
FOR DML AS  
IF DML(campo)  
BEGIN  
INSERT INTO tabela_auditoria  
(IDEMPREGADO, NOME, ANTIGOSAL, NOVOSAL, DATA)  
SELECT D.IDEMP, I.NOME, D.SALARIO, I.SALARIO, GETDATE()  
FROM DELETED D, INSERTED I  
WHERE D.IDEMP = I.IDEMP  
END  
GO
```

- O **BEGIN** e o **END** definem o começo e o fim de um bloco de programação (instruções), **TCL**.
- O **INSERT** define qual a tabela do **TRIGGER** (tabela de auditora normalmente) vai receber os dados e a ordem dos campos em que vai receber.
- Neste caso, o **SELECT** faz uma projeção/consulta dos dados que vão para a tabela do **TRIGGER** (tabela de auditora normalmente), e por consequencia do **INSERT**, os dados consultados são inseridos na tabela do **TRIGGER**.
- O **FROM** ao inves de apontar para a tabela observada pela **TRIGGER**, situação que já esta explicitado no cabeçalho, pode servir para apontar, no caso de uma modificação **DML**, para as áreas onde vão parar os dados modificados, regiões **DELETED** e **INSERTED** (ponteiramento nesse caso é importante), pegando os dados direto dessas regiões, “antes” e “depois” de modificados.
- A instrução **WHERE** garante que quando muitos dados são modificados ao mesmo tempo, o **TRIGGER** perceba caso a caso as modificações de cada registro.

12.2.2 Ponteiramento para DELETED e INSERTED

- Dentro da **TRIGGER** é possível fazer o ponteiramento para as áreas **DELETED** e **INSERTED** para pegar os valores “antes” e “depois” da **DML** de modificação.

- Sintaxe:

```
CREATE TRIGGER TRG_nome  
ON DBO.tabela  
FOR DML AS  
IF DML(campo)  
BEGIN  
  INSERT INTO tabela_auditoria  
    (IDEMPREGADO, NOME, ANTIGOSAL, NOVOSAL, DATA)  
  SELECT D.IDEMP, I.NOME, D.SALARIO, I.SALARIO, GETDATE()  
  FROM DELETED D, INSERTED I  
  WHERE D.IDEMP = I.IDEMP  
END  
GO
```

- O **SELECT** com ponteiramento, garante que seja inserido (**INSERT**), na tabela de auditoria, os valores da tabela observada pelo **TRIGGER** (cabeçalho) “antes” e “depois” do **DML**.

- Basta colocar o ponteiramento apontando para **DELETED** e **INSERTED**, segundo a tabela observada pelo **TRIGGER** (cabeçalho), os valores estão armazenados lá, dado uma modificação **DML**.

- A instrução **WHERE** garante que seja o mesmo campo **ID** (**PRIMARY KEY**), logo o mesmo registro, que sofreu uma modificação.

- Muito usado em caso de **UPDATE**.

- O uso da instrução **SELECT** se dá por conta de boas práticas em **TRIGGERS**:
“Insere nas variáveis valores vindos de tabelas, que são inseridos pelo comando”**SELECT**”.

13 Modulo 27 PARTE 3 - TRIGGER (Gatilho), TRANSAÇÃO (TRANSACTION) e ERROR

13.1 ERROR e RAISERROR

13.1.1 ERROR

- A instrução **@@ERROR** retorna o número do erro da última instrução Transact-SQL executada.
- A instrução **@@ERROR** pode ser usado em conjunto com **TCL IF** e transação (**TRANSACTION**) para impedir que um **ERROR** passe das instruções (*LOG*) para os dados (sistema).
- Valor do **@@ERROR**:
 - Se **@@ERROR** igual a “0”, não houve **ERROR**.
 - Se **@@ERROR** diferente de “0”, algum **ERROR** aconteceu. Ver o número que retornou para determinar o tipo do **ERROR**.
- Sintaxe:
BEGIN TRANSACTION
[bloco de instruções SQL]
IF (**@@ERROR** > 0)
ROLLBACK TRANSACTION
ELSE
COMMIT TRANSACTION
GO

13.1.2 RAISERROR

- No **SQL Server** o **ERROR** vem acompanhado de um código, onde o “*NIVEL*” e o “*ESTADO*” determinam o tipo do **ERROR**.
 - “ERROR DE USUÁRIO”
Código (NIVEL 16, ESTADO 1)
- O “**ERROR** de usuário” é útil pois determina que o foi um **ERROR** devido alguma ação do usuário.
- Pode ser usado o código do “**ERROR** de usuário” na função **RAISERROR** para mostrar uma mensagem ao usuário onde errou, no uso de alguma função construída (**PROCEDURE**) ou **TRIGGER**.
- Função específica para **PRINT** de **ERROR**.
 - **RAISERROR**(‘mensagem’, numero_código_Nivel, numero_código_Estado)
A função que serve para escrever um **ERROR** e dar um **PRINT** dele na tela do usuário.

13.2 TRANSACTION (Transação)

13.2.1 Teoria (LOG)

- Todo banco de dados do SQL Server tem um *LOG* de transações que registra todas as transações e as modificações de banco de dados feitas por cada transação.
- O *LOG* de transações é um componente crítico do banco de dados. Se houver uma falha no sistema, você precisará que o *LOG* retorne o seu banco de dados a um estado consistente.
- O *LOG* são conjuntos de instruções feitos ao sistema, eles são armazenados no sistema, antes de serem enviados suas modificações para os dados.
- Para efeito de um melhor desenvolvimento do banco de dados, é possível confirmar (**COMMIT**) ou eliminar (**ROLLBACK**) um *LOG* antes de dele implicar em modificações nos dados em definitivo, isto é uma transação (**TRANSACTION**).
 - Testa o *LOG* para ver se dá **ERROR**, antes de enviar para o sistema a modificação nos dados.

13.2.2 Confirmar (COMMIT) e eliminar (ROLLBACK) LOG

- A estrutura de uma transação consiste:
 - Inicialização da transação (**BEGIN TRANSACTION/BEGIN TRAN/BEGIN**).
 - O bloco de programação da transação.
 - Confirmação de uma transação bem sucedida, aplicando assim o *LOG* no sistema e por consequência nos dados (**COMMIT**).
 - Descartar o *LOG* da transação e voltar ao estado inicial, reverte as modificações feitas por uma transação, transação mal sucedida (**ROLLBACK**).
 - No caso de for iniciado apenas por **BEGIN** deve terminar com **END**.
- Lembrando que os resultados da transação podem ser vistos ao longo que as instruções são feitas. Porém podem ser desfeitas (**ROLLBACK**) ou confirmadas (**COMMIT**), não são definitivas até então.
- **TRANSACTION/COMMIT/ROLLBACK** são muito úteis para testar o código antes de aplicá-lo em definitivo no sistema. Procurando alguma incidência de **ERROR** ou resultado não esperado.
- Sintaxe exemplo **ERROR**:
IF (@variavel1 < @Variavel2)
BEGIN
RAISERROR ('SALARIO MENOR QUE O PISO',16,1)
ROLLBACK TRANSACTION
END


```
IF (@variavel1 > @variavel3)
BEGIN
RAISERROR('SALARIO MAIOR QUE O TETO',16,1)
ROLLBACK TRANSACTION
END
ou
BEGIN TRANSACTION
[Bloco de instruções SQL]
IF (@@ERROR > 0)
ROLLBACK TRANSACTION
ELSE
COMMIT TRANSACTION
GO
```

13.3 TRIGGER com restrição de regra de negócio e ERROR

- O **TRIGGER** com restrição de regra de negócio e **ERROR** funciona de forma a estabelecer uma determinar regra de negócio e impedir, com uso de **ERROR**, que a regra seja descumprida.
- Uma das formas de estabelecer um regra de negócio com uso de **TRIGGER** é criar uma tabela com os valores chaves a serem cumpridos (minimos, maximos e valores estabelecidos pelo negócio).
- Depois criar um **TRIGGER** que atraves de **TCL** (**IF**'s, operadores lógicos e **ERROR**) impeça modificação de dados para estados que funjam da regra (os valores estabelecidos na tabela criada anteriormente).
- No caso de **ERROR** da regra de negócio, o **TRIGGER** desfaz a transação (**ROLLBACK**), impedindo assim modificações (**DML**) nos dados que fujam da regra de negócio.
- Ou seja, o **TRIGGER** só deixa a modificação (**DML**) funcionar para dados que estejam em conformidade com as regras de negócio.
- Sintaxe exemplo, comentários entre chaves:

[TABELA DE RESTRIÇÃO DA REGRA DE NEGÓCIO]

```
CREATE TABLE SALARIO_RANGE(  
  MINSAL MONEY,  
  MAXSAL MONEY  
)  
GO
```

[INSERINDO VALORES DA RESTRIÇÃO DA REGRA DE NEGÓCIO]

```
INSERT INTO SALARIO_RANGE VALUES  
(2000.00, 6000.00)  
GO
```

[TRIGGER DE RESTRIÇÃO DA REGRA DE NEGÓCIO]

```
CREATE TRIGGER TRG_RANGE [NOME DA TRIGGER]  
ON DBO.EMPREGADO [TABELA]  
FOR INSERT, UPDATE [DML]  
AS
```

[DECLARAÇÃO DE VARIAS VARIÁVEIS AO MESMO TEMPO]

```
DECLARE  
@MINSAL MONEY,  
@MAXSAL MONEY,  
@ATUALSAL MONEY
```

[BLOCO INSERIR VALORES NAS VARIÁVEIS “SELECT”]

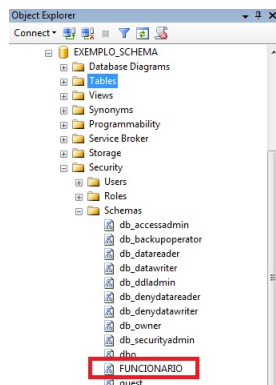
```
SELECT @MINSAL = MINSAL, @MAXSAL = MAXSAL FROM SALARIO_RANGE  
SELECT @ATUALSAL = I.SALARIO
```

FROM INSERTED I

```
[TCL IF]
IF (@ATUALSAL < @MINSAL)
BEGIN
RAISERROR ('SALARIO MENOR QUE O PISO',16,1) [FUNÇÃO PRINT ERROR]
ROLLBACK TRANSACTION [DESFAZE A TRANSAÇÃO (LOG)]
END
IF (@ATUALSAL > @MAXSAL)
BEGIN
RAISERROR('SALARIO MAIOR QUE O TETO',16,1) [FUNÇÃO PRINT ERROR]
ROLLBACK TRANSACTION [DESFAZE A TRANSAÇÃO (LOG)]
END
GO
```

14 SCHEMAS

- **SCHEMA** é um “*Divisor lógico de banco de dados*”.
- Os **SCHEMAS** são uma coleção de objetos dentro de um determinado database (banco de dados), servem para agrupar objetos (tabelas, ...) no nível de aplicação como também para simplesmente fazer divisões departamentais.
- **SCHEMAS** são bastante utilizados em padrões de sistema de banco de dados.
- São muito importantes para a performance e segurança.
 - Podendo receber ou ser retirada permissões dos **SCHEMAS**, por consequência os objetos que ela contém (tabelas) herdam essas características.
 - Facilitando dar permissões em certas tabelas a determinados usuários, permitindo ao usuário acesso a determinado **SCHEMA**.
 - A propriedade dos objetos contidos pelo esquema pode ser transferida para qualquer entidade de segurança no nível de banco de dados, mas o proprietário do esquema sempre retém a permissão *CONTROL* nos objetos do esquema.
- Aparece no nome da tabela no banco de dados. Cada tabela pertence a um **SCHEMA**.
[nome_**SCHEMA**.]nome_tabela
- Todo **SCHEMA** tem um dono.
 - Caso o dono não seja especificado, o dono é o usuário que criou o **SCHEMA**.
- Todos os **SCHEMAS** do banco de dados (**DATABASE**) ficam salvos nas pastas do **SQL Server** do banco de dados: “*Security > Schemas*”.



14.1 SCHEMA padrão do sistema - *dbo*

- O padrão do sistema é o **SCHEMA** *dbo*, ou seja, se você criar uma tabela ou fazer uma inserção em uma tabela e não fazer referência a um **SCHEMA**, ela será criada ou o dado será inserido no **SCHEMA** *dbo* (padrão).

14.2 Criação de SCHEMA

- Para especificar outro usuário como o proprietário do esquema que está sendo criado, o chamador deve ter a permissão *IMPERSONATE* no usuário em questão. Se uma função de banco de dados for especificada como o proprietário, o chamador deverá atender a um dos critérios a seguir: associação na função ou a permissão *ALTER* na função.
- **CREATE SCHEMA**
Cria um schema com determinado nome no sistema.
- **AUTHORIZATION**
Define quem é o proprietário do **SCHEMA** criado.
- Sintaxe:
CREATE SCHEMA *nome_SCHEMA*
AUTHORIZATION *nome_usuário*

14.3 Criação de um SCHEMA no SSMS

- Para criar um **SCHEMA**:
 - No Pesquisador de Objetos, expanda a pasta Bancos de Dados.
 - Expanda o banco de dados no qual o novo esquema (**SCHEMA**) de banco de dados será criado.
 - Clique com o botão direito do mouse na pasta Segurança , aponte para Novo e selecione Esquema (**SCHEMA**).
 - Na caixa de diálogo Esquema - Novo , na página Geral , insira um nome do novo esquema na caixa Nome do esquema (**SCHEMA**).
 - Na caixa Proprietário do esquema , digite o nome de um usuário de banco de dados ou função para ser o proprietário da propriedade do esquema (**SCHEMA**). Como alternativa, clique em Pesquisar para abrir a caixa de diálogo Pesquisar Funções e Usuários.
 - Clique em OK.

14.4 Adicionando uma tabela a um SCHEMA

- Para adicionar uma determinada tabela a um **SCHEMA**, basta na criação da tabela (**CREATE TABLE**) identificar o **SCHEMA** qual ela vai pertencer.
- A identificação do **SCHEMA** vai junto ao nome da tabela.
- Sintaxe:
CREATE TABLE *nome_SCHEMA.nome_tabela* (
...
)
GO
- Se uma tabela for colocada num esquema (**SCHEMA**) que não seja o *dbo*, sempre que ela for nomeada, lembrar de especificar o esquema junto com o nome.
nome_SCHEMA.nome_tabela

14.5 Permissões do SCHEMA

- Comandos auxiliares para dar ou revogar permissões através do **SCHEMA**.
- Para dar ou revogar privilégios a outros usuários por esta forma, o dono do **SCHEMA** precisa ter esse privilégio.
- Sintaxe:
CREATE SCHEMA *nome_SCHEMA* **AUTHORIZATION** *nome_usuario_dono*
GRANT *lista_privilegios* **ON** [*SCHEMA::database.*]*tabela* **TO** [*outro_*]*usuário*
REVOKE *lista_privilegios* **ON** [*SCHEMA::database.*]*tabela* **FROM** [*outro_*]*usuário*
DENY *lista_privilegios* **ON** [*SCHEMA::database.*]*tabela* **TO** [*outro_*]*usuário*
GO
- Sobre permissões:
 - **GRANT**
Concede permissões, se o criador tiver essa autorização do sistema.
 - **REVOKE**
Cancela permissões.
 - **DENY**
Impede explicitamente que um usuário receba uma permissão específica, mesmo herdando ela de alguma forma.
- Note que **REVOKE** usa **FROM** ao invés de **TO**.
- Para mais detalhes ver capítulo “Categorias de comandos - **DCL**”

14.6 ALTER SCHEMA

- Permite apenas transferir propriedade de uma tabela, ou seja, transferir uma tabela de um **SCHEMA** para outro.
- Sintaxe:
 - Transferir tabela entre SCHEMAS:
ALTER SCHEMA *novo_schema* **TRANSFER** *antigo_schema.tabela*
GO
 - Transferir do dbo para um novo_schema:
ALTER SCHEMA *novo_schema* **TRANSFER OBJECT::***dbo.tabela* **GO**

14.7 Listar todos os SCHEMAS do banco de dados

- Comando para listar todos os **SCHEMAS** contidos no banco de dados.
- Sintaxe:
SELECT * FROM *sys.schemas*
GO

15 Modulo 28 - Variáveis e PROCEDURES

15.1 Variáveis locais e globais

- A diferença de variáveis globais e locais são:
 - As variáveis globais são visíveis e podem ser manipuladas por todo sistema e todos os usuários.
 - As variáveis locais têm escopo local e são apenas visíveis dentro do lote ou procedimento, onde elas estão definidas.
- Por regra as variáveis locais começam com um caracter de identificação “@” antes do nome.
- Por regra as variáveis globais começam com dois caracteres de identificação “@@” antes do nome.
- O processo de declaração de variáveis, tanto global quanto local, são semelhantes.
DECLARE *@nome_variavel_local* tipo
DECLARE *@@nome_variavel_global* tipo
- As variáveis em lotes e scripts são normalmente usadas:
 - Como um contador, para contar o número de vezes que um loop é executado ou controlar quantas vezes que o loop é executado.
 - Para reter um valor de dados a ser testado por uma instrução de controle de fluxo.
 - Para salvar um valor de dados a ser retornado por um código de retorno de procedimento armazenado ou valor de retorno de função.

15.2 STORAGE PROCEDURES (ou STORED PROCEDURES)

15.2.1 Teoria STORAGE PROCEDURES

- **STORAGE PROCEDURE** (**SP__**) são **PROCEDURES** armazenadas do sistema.
- São procedimentos de sistema. Armazenados no banco de dados padrão do **SQL Server** (Master), podemos indentifica-los com as siglas **SP**, que se origina de “stored procedure”. Tais **PROCEDURES** executam as tarefas administrativas e podem ser executadas a partir de qualquer banco de dados.

15.2.2 STORAGE PROCEDURES

15.3 PROCEDURES

15.3.1 Teoria PROCEDURE

- **PROCEDURES** são o equivalente a funções em outras linguagens de programação. Podendo ter valores de entrada e saída.
- Serve para executar determinadas operações (bloco de programação SQL) e transformar determinados objetos (dados, tabelas, banco de dados, ...), e repetir o processo toda vez que for chamada.
- Procedimentos armazenados são semelhantes a procedimentos em outras linguagens de programação no sentido de que podem:
 - Aceitar parâmetros de entrada e retornar vários valores no formulário de parâmetros de saída para o procedimento de chamada ou lote.
 - Conter instruções de programação que executam operações no banco de dados, inclusive chamar outros procedimentos.
 - Retornar um valor de status a um procedimento de chamada ou lote para indicar êxito ou falha (e o motivo da falha).

15.3.2 Instruções básicas de PROCEDURES

15.3.2.1 CREATE PROCEDURE

- Para criar uma **PROCEDURE**, basta usar a instrução **CREATE PROCEDURE**, ou **CREATE PROC** + nome da **PROCEDURE**.
- Caso tenha argumentos (parâmetros) basta adicionar no cabeçalho as variáveis (*@variavel*) e seus respectivos tipos.
- Após o cabeçalho (**CREATE PROC** + *nome_PROC* + argumentos), vem a instrução **AS** que inicia o “bloco de instrução **SQL**”, com todas as instruções que a **PROCEDURE** deve executar, encerrando com **GO**.
- Sintaxe:
CREATE PROC *nome_PROC* *@variavel1* tipo, *@variavel2* tipo, ...
AS
[Bloco de programação SQL]
GO

15.3.2.2 Bloco de programação SQL de uma PROCEDURE

- O bloco de programação SQL são as instruções SQL que a **PROCEDURE** vai executar toda vez em que for chamada.

- Sintaxe típica de uma **PROCEDURE** de consulta:
CREATE PROC *nome_PROC* *@TIPO* CHAR(3)
AS
SELECT
P.NOME,
T.NUMERO
FROM *tabela1* **P**
INNER JOIN *tabela2* **T**
ON *P.IDPESSOA* = *T.ID_PESSOA*
WHERE *TIPO* = *@TIPO*
GO

15.3.2.3 Chamando a PROCEDURE

- Há duas maneiras chamar (executar) uma **PROCEDURE**, já criada, apenas chamando pelo nome da **PROCEDURE**, ou usar a instrução **EXEC** + nome da **PROCEDURE**. As boas práticas recomendam o uso de **EXEC**.
- Quando chamada **PROCEDURES** que aceitem parâmetros, eles são inseridos depois do nome da **PROCEDURE**.
- Sintaxe:
EXEC *nome_PROC*
GO
 ou
nome_PROC
GO

15.3.2.4 Apagando PROCEDURES

- **DROP PROC** é uma instrução que apaga uma **PROCEDURE** criada anteriormente com a instrução **CREATE PROCEDURE** no **SQL Server**.
- Sintaxe:
DROP (PROCEDURE | PROC) [*SCHEMA.*]*nome_Procedure*
GO

15.3.2.5 Alterando PROCEDURE

- **ALTER PROC** modifica uma **PROCEDURE** criado anteriormente com a execução da instrução **CREATE PROCEDURE** no **SQL Server**.
- Serve tanto para modificar seus parâmetros, quanto seu bloco de programação **SQL**.

- Sintaxe:
ALTER (**PROCEDURE** | **PROC**) [*SCHEMA.*]*nome_Procedure* *@parâmetro_1 tipo, @parâmetro_2 tipo, ...*
AS
[Bloco de programação SQL]
GO

15.3.3 PROCEDURES estáticas

- **PROCEDURES** estáticas são as **PROCEDURES** que não contêm parâmetros de entrada e/ou saída.
- São uteis para executar determinados cálculos, mostrar determinadas consultas, ou resultados, muito recorrentes.
- Sintaxe com exemplo:
CREATE PROC *nome_PROCEDURE*
AS
SELECT 10 + 10 **AS** *SOMA*
GO
EXEC *nome_PROCEDURE*
GO

15.3.4 PROCEDURES dinâmicas

- **PROCEDURES** dinâmicas são as **PROCEDURES** que contêm parâmetros, de entrada e/ou saída.

15.3.4.1 PROCEDURES com parâmetros de Entrada

- Parâmetros de entrada são dados que são inseridos e aceitos dentro das **PROCEDURES**.
- Os parâmetros de entrada são inseridos na **PROCEDURE** por meio de variáveis definidas quando criadas as **PROCEDURES** (**CREATE PROC**).
CREATE PROCEDURE *nome_PROCEDURE* *@variavel1* tipo_da_variavel, ...
- No bloco de programação SQL da **PROCEDURE** são trabalhadas as variáveis.
...
AS
SELECT *@NUM1* + *@NUM2* **AS** *RESULTADO*
GO
- Para inserir os parâmetros nas variáveis quando chamar as **PROCEDURES** dinâmicas basta inserir os dados ao lado do nome da **PROCEDURE** (se forem mais de um parâmetro basta separar por vírgula).
EXEC PROC *nome_PROCEDURE* *valor1, valor2, ...*
GO
- No caso de variáveis do tipo **CHAR** ou **VARCHAR**, colocar os valores entre parênteses simples.
EXEC *TELEFONES* 'CEL'
GO

- Sintaxe:
CREATE PROCEDURE *nome_PROCEDURE* *@variavel1* *tipo_da_variavel*, *@variavel2*
tipo_da_variavel
AS
SELECT *@variavel1* + *@variavel2* **AS** *RESULTADO*
GO
EXEC PROC *nome_PROCEDURE* *valor1*, *valor2*
GO

15.3.4.2 PROCEDURES com parâmetros de Saída

- Parâmetros de saída são variáveis de saída, que retornam valores.
- Ao criar uma **PROCEDURE** (**CREATE PROC**), os parâmetros de saída são variáveis acompanhadas do tipo e o indicador **OUTPUT**.
CREATE PROCEDURE *nome_PROCEDURE* *@Variavel_entrada* CHAR(3), *@Variavel_saida* INT **OUTPUT**
...
- **OUTPUT** indica que o parâmetro é um parâmetro de saída.
- Use parâmetros **OUTPUT** para retornar valores do chamador do **PROCEDURE**.
DECLARE *@SAIDA* INT [DECLARANDO UMA VARIÁVEL NA MEMÓRIA RAM, **TCL**]
EXEC *nome_PROCEDURE* *@variavel_entrada* = 'CEL', *@variavel_saida* = *@SAIDA* **OUTPUT**
GO

15.3.4.3 PROCEDURES com parâmetros de entrada e saída

- Utilizar PROCEDURES com parâmetros de entrada e saída potencializa o uso.
- Lembrar de usar **OUTPUT** para variáveis de saída.
- Para salvar o parâmetro de saída e poder utilizar seu valor a qualquer momento, basta associar ele a uma variável salva na memória RAM (declarada fora de qualquer escopo).
DECLARE *@SAIDA* INT [DECLARANDO UMA VARIÁVEL NA MEMÓRIA RAM, **TCL**]
EXEC *GETTIPO* *@TIPO* = 'CEL', *@CONTADOR* = *@SAIDA* **OUTPUT**
...
ou
DECLARE *@SAIDA* INT [DECLARANDO UMA VARIÁVEL NA MEMÓRIA RAM, **TCL**]
EXEC *GETTIPO* 'CEL', *@SAIDA* **OUTPUT**
...
- Sintaxe, com exemplo e comentários entre colchetes:
CREATE PROCEDURE *GETTIPO* *@TIPO* CHAR(3), *@CONTADOR* INT **OUTPUT**
AS
SELECT
@CONTADOR = COUNT(*)

```

FROM TELEFONE01
WHERE TIPO = @TIPO
GO
DECLARE @SAIDA INT [DECLARANDO UMA VARIÁVEL NA MEMÓRIA RAM, TCL]
EXEC GETTIPO @TIPO = 'CEL', @CONTADOR = @SAIDA OUTPUT
SELECT @SAIDA [PROJEÇÃO, imprimindo na tela]
GO
ou
DECLARE @SAIDA INT [DECLARANDO UMA VARIÁVEL NA MEMÓRIA RAM, TCL]
EXEC GETTIPO 'CEL', @SAIDA OUTPUT [APENAS COLOCANDO VALORES E VARIÁVEIS
NAS POSIÇÕES CERTAS]
SELECT @SAIDA [PROJEÇÃO, imprimindo na tela]
GO

```

16 Categorias de comandos

16.1 DML - *Data Manipulation Language* (Linguagem de Manipulação de Dados)

É um conjunto de instruções usada nas consultas e modificações dos dados armazenados nas tabelas do banco de dados.

- **SELECT**

- Recupera linhas do banco de dados e permite a seleção de uma ou várias linhas ou colunas de uma ou várias tabelas.
- Projeção do que quer ter de visualização na tela.
- Sintaxe:
SELECT *
FROM *tabela*
GO

- **INSERT**

- Adiciona registros numa tabela.
- Sintaxe:
INSERT INTO *nome_da_tabela*
VALUES
(*valor_na_coluna_1_registro1*, *valor_na_coluna_2_registro1*,...),
(*valor_na_coluna_1_registro2*, *valor_na_coluna_2_registro2*,...),
...
GO

- **UPDATE**

- Altera os dados de um ou mais registros em uma tabela.
- Sintaxe:
UPDATE *tabela* **SET** *coluna_a_atualizar* = *valor_atualizado*
WHERE *condição* = *valor*
GO

- **DELETE**

- Remove um ou mais registros de uma tabela.
- Sintaxe:
DELETE FROM *tabela*
WHERE *critério_do_que_se_quer_deletar* = *valor*
GO

- **BULK INSERT***

- Importa um arquivo de dados em uma tabela ou exibição do banco de dados em um formato especificado pelo usuário.

- Sintaxe:

```
BULK INSERT tabela_importação  
FROM 'caminho'  
WITH(  
  FIRSTROW = 2,  
  DATAFILETYPE = 'char',  
  FIELDTERMINATOR = '\t',  
  ROWTERMINATOR = '\n'  
)  
GO
```

- * Mais detalhes no “*Modulo 26 PARTE 4 - Importação de arquivo de dados*”.

16.2 DDL - *Data Definition Language* (Linguagem de definição de dados)

É um conjunto de instruções usado para criar e modificar as estruturas dos objetos armazenados no banco de dados.

- **CREATE**

Utilizada para construir um novo banco de dados, tabela, índice ou consulta armazenada.

- **DATABASE**

- * Criação de banco de dados.

- * Sintaxe:

- CREATE DATABASE** *nome_banco_de_dados*
GO

- **TABLE**

- * Criação de tabela.

- * Sintaxe:

- CREATE TABLE** *nome_tabela* (
 coluna1 *tipo* *regra* *retrições*,
 coluna2 *tipo* *regra* *retrições*,
 ...
)

GO

- **DROP**

Remove um banco de dados, tabela, índice ou visão existente.

- **DATABASE**

- * Remove banco de dados.

- * Sintaxe:

- DROP DATABASE** *nome_do_banco_de_dados*
GO

- **TABLE**

- * Remove tabela.

- * Sintaxe:

- DROP TABLE** *nome_da_tabela*
GO

- **ALTER**

- Modifica um objeto existente do banco de dados.
- É possível incluir, eliminar e alterar colunas.
- Para alterar uma tabela existente, é necessário que os registros existentes já sejam compatíveis com a alteração.

* ALTER COLUMN

- Altera o *tipo* e *regras* de uma coluna/campo.
- Sintaxe:
ALTER TABLE *nome_tabela*
ALTER COLUMN *nome_coluna* *modificação_tipo*
GO

* ADD

- Adiciona chaves (primária ou estrangeira) a uma coluna.
- Não é possível adicionar “*auto_increment*”.
- Sintaxe:
ALTER TABLE *tabela*
ADD PRIMARY KEY(*coluna*)
GO
ou
ALTER TABLE *tabela*
ADD FOREIGN KEY(*coluna_da_tabela*)
REFERENCES (*coluna_chave_primaria_de_outra_tabela*)
GO
- O comando **ADD** funciona para adicionar nova coluna.
Sintaxe:
ALTER TABLE *tabela*
ADD *nova_coluna* *tipo*
GO

* DROP COLUMN

- Deleta uma determinada coluna de uma tabela.
- Sintaxe:
ALTER TABLE [*nome_database.*]*nome_tabela*
DROP COLUMN *nome_coluna*
GO

* ALTER DATABASE

- Alterar nome de uma database.

- Sintaxe:
ALTER DATABASE *nome_database*
MODIFY NAME = *novo_nome_database*
GO

- **SP_RENAME**

- Mudar nome da tabela e/ou coluna.

- Sintaxe:

- * Mudar nome da *tabela*:

- SP_RENAME** '*NomeTabelaAntigo*', '*NomeTabelaNovo*'

- GO**

- ou

- * Mudar nome da *coluna*:

- SP_RENAME** '*NomeTabela.NomeColunaAntigo*', '*NovoNomeColuna*', '*COLUMN*'

- GO**

- **TRUNCATE**

- Esvazia imediatamente todo o conteúdo de uma tabela ou objeto que contenha dados.

- É muito mais rápido que um comando DELETE, pois, ao contrário deste, não armazena os dados sendo removidos no log de transações. Por esse motivo, em vários SGBDs é um comando não-transacional e irreversível, não sendo possível desfazê-lo com **ROLLBACK**.

- Sintaxe:

- TRUNCATE TABLE** *nome_tabela*

- GO**

16.3 DCL - *Data Control Language* (Linguagem de Controle de Dados)

São usados para controle de acesso e gerenciamento de permissões para usuários em no banco de dados. Com eles, pode facilmente permitir ou negar algumas ações para usuários nas tabelas ou registros (segurança de nível de linha).

16.3.1 Login

- **CREATE LOGIN**

- A instrução **CREATE LOGIN** cria uma identidade usada para se conectar a uma instância do SQL Server.
- O login é então mapeado para um usuário do banco de dados (portanto, antes de criar um usuário no SQL Server, você deve primeiro criar um login).
- Quatro tipos de logons que você pode criar no SQL Server:

- * Pode-se criar um login usando a autenticação do **Windows**.

```
CREATE LOGIN test_domain/techonthenet  
FROM WINDOWS  
GO
```

Obs.: Este exemplo de **CREATE LOGIN** criaria um novo Login chamado *test_domain/techonthenet* que usa a autenticação do Windows.

- * Pode-se criar um login usando a autenticação do **SQL Server**.

```
CREATE LOGIN nome_login  
WITH PASSWORD = 'senha'  
GO
```

- * Pode-se criar um Login a partir de um **certificate**.

```
CREATE LOGIN nome_login  
FROM CERTIFICATE nome_certificação  
GO
```

- * Pode-se criar um Login a partir de um **asymmetric key**.

```
CREATE LOGIN nome_login  
FROM ASYMMETRIC KEY nome_asym_key  
GO
```

- **ALTER LOGIN**

- A instrução **ALTER LOGIN** modifica uma identidade usada para se conectar a uma instância do SQL Server.
- Você pode usar a instrução **ALTER LOGIN** para:

- * Alterar uma senha

```
ALTER LOGIN nome_login
```

```
WITH PASSWORD = 'nova_senha'  
GO
```

- * Forçar uma alteração de senha

```
ALTER LOGIN nome_login  
WITH PASSWORD = 'nova_senha' MUST_CHANGE, CHECK_EXPIRATION  
= ON  
GO
```

Obs.: Força a alteração da senha após o primeiro logon usando a instrução **ALTER LOGIN** no **SQL Server** (Transact-SQL).

- * Desabilitar um login

```
ALTER LOGIN nome_login DISABLE  
GO
```

- * Habilitar um login

```
ALTER LOGIN nome_login ENABLE  
GO
```

- * Desbloquear um login

```
ALTER LOGIN nome_login  
WITH PASSWORD = 'senha'  
UNLOCK  
GO
```

- * Renomear um login

```
ALTER LOGIN nome_login  
WITH NAME = novo_nome  
GO
```

- * Etc.

- **DROP LOGIN**

- A instrução **DROP LOGIN** é usada para remover uma identidade (ou seja: Login) usada para conectar a uma instância do **SQL Server**.

- Sintaxe:

```
DROP LOGIN nome_login  
GO
```

- *Find logins* no **SQL Server**

- No SQL Server, há uma exibição de catálogo (ou seja: exibição do sistema) chamada `sys.sql_logins`.

- Você pode executar uma consulta nessa exibição do sistema que retorna todos os logins que foram criados no **SQL Server**, bem como informações sobre esses logins.

- Sintaxe:

```
SELECT *
```

```
FROM master.sys.sql_logins  
GO
```

16.3.2 USER - Usuário

- **CREATE USER**

- A instrução **CREATE USER** cria um usuário de banco de dados para fazer logon no SQL Server.
- Um usuário de banco de dados é mapeado para um **LOGIN**, que é uma identidade usada para se conectar a uma instância do SQL Server.
- Comando para criação de usuários.

- Sintaxe:

```
CREATE USER user_nome FOR LOGIN login_nome  
GO
```

- * *user_nome*

- O nome do usuário do banco de dados que você deseja criar.

- * *login_nome*

- O **Login** usado para se conectar à instância do SQL Server.

- Listar usuários:

- No SQL Server, há uma exibição do sistema chamada *sys.database_principals*. Você pode executar uma consulta nessa exibição do sistema que retorna todos os usuários que foram criados no SQL Server, bem como informações sobre esses usuários.

- Sintaxe:

```
SELECT *  
FROM master.sys.database_principals  
GO
```

- Mostrar usuário conectado atual:

- Função **SUSER_NAME()**.

- Função que retorna o usuario logado no banco de dados no momento.

- Útil para usar dentro de **TRIGGERS** para salvar o usuario reponsavel por alguma alteração numa tabela (auditoria).

- Sintaxe:

```
SELECT SUSER_NAME()  
GO
```

- Removendo usuários:

- A instrução **DROP USER** é usada para remover um usuário do banco de dados SQL Server.

– Sintaxe:
DROP USER *user_nome*
GO

16.3.3 Permissões

- **GRANT**

- Permitir que usuários especificados realizem tarefas especificadas.
- Também permite gerenciar permissão para realizar tarefas específicas em database e/ou tabelas específicas.
- Sintaxe:
GRANT *lista_de_privilégios* **ON** [*nome_database.*]*nome_tabela* **TO** *nome_usuario*
GO
ou para dar permissão de root:
GRANT ALL ON * . * **TO** *nome_usuario*
GO
- Revisar as permissões atuais de um usuário:
 - * Para obter informações sobre as permissões dos usuários ou funções, você pode consultar a exibição do catálogo do sistema sys.database_principals.
 - * Esta será uma lista enorme. Você também pode personalizar essa consulta para obter as permissões associadas a um usuário ou função adicionando a condição WHERE.
 - * Sintaxe:
SELECT *pri.name* **AS** *Username*,
pri.type_desc **AS** [*User Type*],
permit.permission_name **AS** [*Permission*],
permit.state_desc **AS** [*Permission State*],
permit.class_desc **Class**,
object_name(*permit.major_id*) **AS** [*Object Name*]
FROM *sys.database_principals* *pri*
LEFT JOIN *sys.database_permissions* *permit*
ON *permit.grantee_principal_id* = *pri.principal_id*
[**WHERE** *name* = '*nome_usuario*']
GO

- **REVOKE**

- Cancela/revoga permissões previamente concedidas.
- Sintaxe:
REVOKE *lista_de_privilégios* **ON** [*nome_database.*]*nome_tabela* **FROM** *nome_usuario*
GO
Obs.: Note que no **REVOKE** é usado **FROM** e no **GRANT** é usado **TO**.

- **DENY**

- O comando é usado para impedir explicitamente que um usuário receba uma permissão específica.

- A instrução **DENY** impede que os usuários executem ações. Isso significa que a instrução remove as permissões existentes das contas de usuário ou impede que os usuários obtenham permissões por meio de sua associação de grupo/função que pode ser concedida no futuro.
 - Todas as opções da instrução **DENY** têm o mesmo significado lógico que as opções com o mesmo nome na instrução **GRANT**.
 - **DENY** tem uma opção adicional, *CASCADE*, que especifica que as permissões serão negadas ao usuário A e a qualquer outro usuário para quem o usuário A passou essa permissão. (Se a opção *CASCADE* não for especificada na instrução **DENY** e a permissão de objeto correspondente tiver sido concedida com **WITH GRANT OPTION**, um erro será retornado).
 - Sintaxe:
DENY *lista_privilegios* **ON** *objeto::database.tabela* **TO** *usuário*
GO
- Privilégios que podem ser CONCEDIDOS à ou REVOCADOS de um usuário:
 - **ALL** — **ALL** não concede todas as permissões para a tabela. Em vez disso, ele concede as permissões ANSI-92 que são **SELECT**, **INSERT**, **UPDATE**, **DELETE** e **REFERENCES**.
 - **CREATE** — permite criar novas tabelas ou bancos de dados.
 - **SELECT** - permite usar o comando **SELECT** para ler os bancos de dados.
 - **DROP** — permite deletar tabelas ou bancos de dados.
 - **DELETE** — permite excluir linhas de tabelas.
 - **INSERT** — permite inserir linhas em tabelas.
 - **UPDATE** — permite atualizar linhas de tabelas.
 - **REFERENCES** - Capacidade de criar uma restrição que se refere à tabela.
 - **ALTER** - Capacidade de executar instruções **ALTER TABLE** para alterar a definição da tabela.
 - **GRANT OPTION** — permite conceder ou remover privilégios de outros usuários.

16.4 TCL - *Tool Command Language* (Linguagem de Comandos de Ferramentas)

São usados para gerenciar as mudanças feitas por instruções DML. Ele permite que as declarações a serem agrupadas em transações lógicas.

- **BEGIN TRANSACTION**

- O comando garante que diversas instruções sejam executadas, porém se alguma for mal sucedida todas falham.
- É possível avaliar o processo de implementação das instruções e seus resultados e caso necessário regressar ao estado anterior às instruções ou confirmar sua implementação.
- Principais instruções que são comuns de serem usadas na transação são as **DML** (**INSERT**, **UPDATE** e **DELETE**).
- Sintaxe:
BEGIN TRANSACTION (ou apenas, **BEGIN**)

- **BACKROLL**

- Regressão para o estado anterior ao início da transação (**BEGIN TRANSACTION**).
- Sintaxe:
BACKROLL
GO

- **COMMIT**

- Confirmação de que as instruções da transação (**BEGIN TRANSACTION**) podem ser implementadas sem problemas.
- Sintaxe:
COMMIT
GO

17 Observações

17.1 Problemas para fazer *login* o SSMS

- Caso o **SSMS** não identifique o usuário “sa” e senha como deveria, seguir os seguintes passos:
 - Desabilitar temporariamente o antivírus do computador.
 - Desabilitar o “**firewall**” do computador.
“Painel de Controle\Sistema e Segurança\Windows Defender Firewall\Personalizar Configurações”
 - Abrir o *instalador* de **SQL Server** e pedir para “**Reparar**”.
 - Ao final da reparação, abrir o **SSMS** novamente e fazer o *login*.

17.2 Abreviações do nome de restrições (CONSTRAINTS) no dicionário de dados - sistema (boas práticas)

- Padronização do nome das restrições salvas no sistema.
- Abreviações do nome das restrições (**CONSTRAINTS**), para salvar no sistema por meio do **ALTER TABLE**.
 - ‘**PK**’ é abreviação de “**PRIMARY KEY**”
 - ‘**FK**’ é abreviação de “**FOREIGN**”
 - ‘**UQ**’ é abreviação de “**UNIQUE**”
 - ‘**CK**’ é abreviação de “**CHECK**”

17.3 Formato da data no sistema

“aaaa-mm-dd hh:mm:ss.mmm”
(ano-mês-dia hora:minuto:segundos.milisegundos)

18 Andamento dos Estudos

18.1 Assunto em andamento

Atualmente estou estudando Módulo 28 - AULA 111.