“Programação para Banco de Dados”
(Parte 2)

Prof. Nathan Cirillo e Silva
Prof. Télvio Orrú
Universidade Paulista UNIP

LPBD@2020

Aula 6

Tratamento de Erros

- Não importa a linguagem em que está programando, **erros existem e devemos lidar com eles**;
- Ao criar uma aplicação, **a própria linguagem de programação usada poderá trata-los**;
- Já em **Scripts T-SQL** é necessário lidar com os erros usando **comandos específicos**;
- Imagine *inserir um registro que viola a chave primária* ou *dividir um valor por zero*.

Como Identificar e Capturar Erros

Função @@ERROR

- Um **modo primitivo** de identificar erros na linguagem T-SQL é **checar o valor da função @@ERROR**;
- Tal função **retorna um número maior do que zero (> 0)** caso identifique algum tipo de erro;
- O **retorno da função** deve ser **atribuído a uma variável**, pois o seu valor **muda a cada declaração**;
- Assim, *caso o seu valor seja maior do que zero*, torna-se possível **tomar uma determinada ação**.

Veja Alguns Exemplos

```
DECLARE @numErro INT;
DROP TABLE NaoExiste;
SET @numErro = @@ERROR;
IF @numErro > 0 BEGIN
    PRINT 'Ocorreu um erro!';
    PRINT @numErro;
    PRINT @@ERROR;
END;
```

Salva o retorno em uma variável!

```
Mensagem 3701, Nível 11, Estado 5, Linha 2
Não é possível 'Descartar' o tabela 'NaoExiste', pois ele não existe ou você não tem permissão.
Ocorreu um erro!
3701
0
```

A variável evita que o valor da função seja perdido.

```
DECLARE @numErro INT;
PRINT 1/0;
SET @numErro = @@ERROR;
IF @numErro > 0 BEGIN
    PRINT 'Ocorreu um erro!';
    PRINT @numErro;
    PRINT @@ERROR;
END;
```

```
Mensagem 8134, Nível 16, Estado 1, Linha 2
Erro de divisão por zero.
Ocorreu um erro!
8134
0
```

O erro em vermelho é gerado automaticamente pelo mecanismo do BD!

Usando o Comando **GOTO**

- A linguagem T-SQL permite utilizar o **comando GOTO** para **pular partes de um código**;
- É necessário **criar uma etiqueta (*label*)** para que seja **chamada via comando GOTO** (*salto*);
- *O código existente entre o comando GOTO e a etiqueta é ignorado, ou seja, pulado;*
- O ideal é usar o GOTO **apenas para tratar erros**.

Veja um Exemplo Prático

Não será Apresentado!

```
DECLARE @numErro INT;
PRINT 'Início do Código';
PRINT 1/0;
SET @numErro = @@ERROR;
IF @numErro > 0
    GOTO ERRO_LABEL;
PRINT 'NÃO HOUVE ERROS!';
ERRO_LABEL: PRINT 'OCORREU UM ERRO!';
```

REALIZA UM SALTO

```
Início do Código
Mensagem 8134, Nível 16, Estado 1, Linha 3
Erro de divisão por zero.
OCORREU UM ERRO!
```

Comando **TRY...CATCH**

- **É recomendado usar o TRY...CATCH** ao invés da função @@ERROR (*método mais antigo*);
- A sintaxe desse comando é **similar a encontrada em outras linguagens de programação**;
- A sua vantagem é a **existência de diversas funções** que retornam detalhes sobre o erro.

```
BEGIN TRY
   <códigos que podem gerar erros>
END TRY
BEGIN CATCH
   <códigos que lidam com os erros>
END CATCH
```

Principais **Funções de Erro**

Função	Propósito
ERROR_NUMBER()	Retorna o número do erro. Era a única informação que tínhamos acesso.
ERROR_SEVERITY()	Retorna a gravidade do erro. O valor deve exceder 10 para ser capturado.
ERROR_STATE()	Retorna o código de estado do erro. Está relacionado a causa do erro.
ERROR_PROCEDURE()	Retorna o nome do *stored procedure* ou *trigger* que causou o erro.
ERROR_LINE()	Retorna o número da linha que causou o erro.
ERROR_MESSAGE()	Retorna a mensagem atual de erro.

Utilizando **TRY...CATCH**

```
BEGIN TRY
    DROP TABLE NaoExiste;
END TRY
BEGIN CATCH
    PRINT ERROR_NUMBER();
    PRINT ERROR_MESSAGE();
    PRINT ERROR_SEVERITY();
END CATCH
```

```
3701
Não é possível 'Descartar' o tabela 'NaoExiste', pois ele não existe
ou você não tem permissão.
11
```

```
BEGIN TRY
    PRINT 50/0;
END TRY
BEGIN CATCH
    PRINT 'Dentro do Bloco: ';
    PRINT ERROR_NUMBER();
    PRINT ERROR_MESSAGE();
END CATCH
    PRINT 'Fora do Bloco: ';
    PRINT ERROR_NUMBER();
```

As funções **não perdem o seu valor** *dentro do bloco CATCH.*

```
tro do Bloco:
8134
Erro de divisão por zero.
Fora do Bloco:
```

Se as funções *estiverem fora do bloco CATCH* os seus **valores são resetados para NULL**.

Erros Não Capturáveis

- Existem erros que a estrutura TRY...CATCH **não é capaz de capturar**, ou seja, **não identifica-os**;
- Se o **nome da tabela ou coluna estiver incorreto em uma consulta**, por exemplo, **nada é feito**;
- Em outras palavras, **as declarações do bloco CATCH não são executadas**;
- Para resolver isso, a consulta pode ser encapsulada em uma *Stored Procedure (SP)*.

Veja um Exemplo Prático!

A tabela mencionada na consulta **NÃO EXISTE!**

```
BEGIN TRY
    PRINT 'PRIMEIRA PARTE!'
    SELECT idade FROM NãoExiste;
END TRY
BEGIN CATCH
    PRINT ERROR_NUMBER();
END CATCH
```

O bloco CATCH não será executado!

```
PRIMEIRA PARTE!
Mensagem 208, Nível 16, Estado 1, Linha 3
Nome de objeto 'NãoExiste' inválido.
```

A consulta foi envolvida em uma *SP!*

```
CREATE PROCEDURE Consulta_Idade
AS SELECT idade FROM NãoExiste;
```

```
BEGIN TRY
    PRINT 'PRIMEIRA PARTE!';
    EXECUTE Consulta_Idade;
END TRY
BEGIN CATCH
    PRINT ERROR_NUMBER();
END CATCH
```

A SP é chamada no bloco TRY!

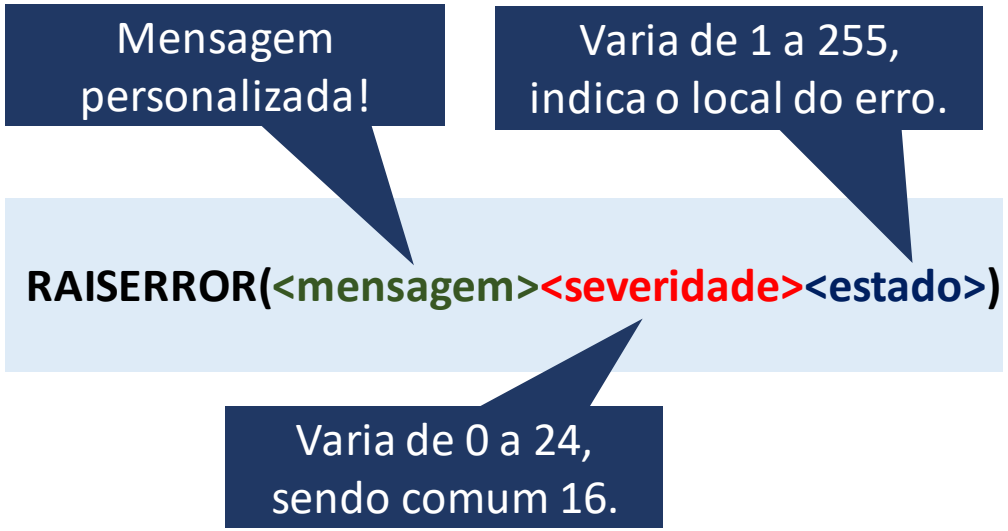
```
PRIMEIRA PARTE!
208
```

Agora o bloco CATCH é executado!

Como Gerar Erros Personalizados

Trabalhando com o **RAISERROR**

- Ao usar o **TRY...CATCH** você **evita que uma mensagem de erro retorne** para a aplicação do usuário;
- *Mas e se for **necessário retornar um erro** e em alguns casos **mesmo quando esse não exista?***;
- *Ao alterar um registro que não existe, isso não gera um erro de BD;*
- Porém você pode querer uma mensagem de erro para isso.



Níveis de Severidade

- **0 a 10** = indicam mensagens não severas ou informacionais;

Ideal para personalizar procedimentos!

- **11 a 16** = indicam erros que podem ser corrigidos pelo usuário, como: sintaxe ou permissão;

- **17 a 19** = indicam erros que devem ser repassados para o administrador, como: falta de memória ou espaço em disco;

- **20 a 24** = indicam erros fatais, onde o processo em execução é finalizado e um registro é gerado (log).

Utilização do **RAISERROR**
(Continua)

```
CREATE PROCEDURE Busca_Nome
@nm NVARCHAR(30)
AS
DECLARE @qtd INT;
SELECT @qtd = COUNT(*) FROM Funcionarios WHERE
nome LIKE @nm+'%';
RETURN @qtd;

GO
```

Retorna o **número de Funcionários** que **começam com o nome especificado!**

Utilização do **RAISERROR**
(Continua)

```
DECLARE @nome NVARCHAR(30) = 'And';
DECLARE @nmaux NVARCHAR(50);
DECLARE @rows INT;
EXECUTE @rows = Busca_Nome @nome;
IF @rows = 0 BEGIN
    SET @nmaux = 'Nenhum nome começa com: ' +@nome;
    RAISERROR(@nmaux,16,1);
END
ELSE BEGIN
    SET @nmaux = 'Há ' + CAST(@rows AS NVARCHAR) + ' nome(s)!';
    RAISERROR(@nmaux,16,1);
END;
```

Chama a SP gerando um erro para ambos os casos!

Utilização do **RAISERROR**

*Existe algum funcionário que o nome comece com **'And'**?*

SIM

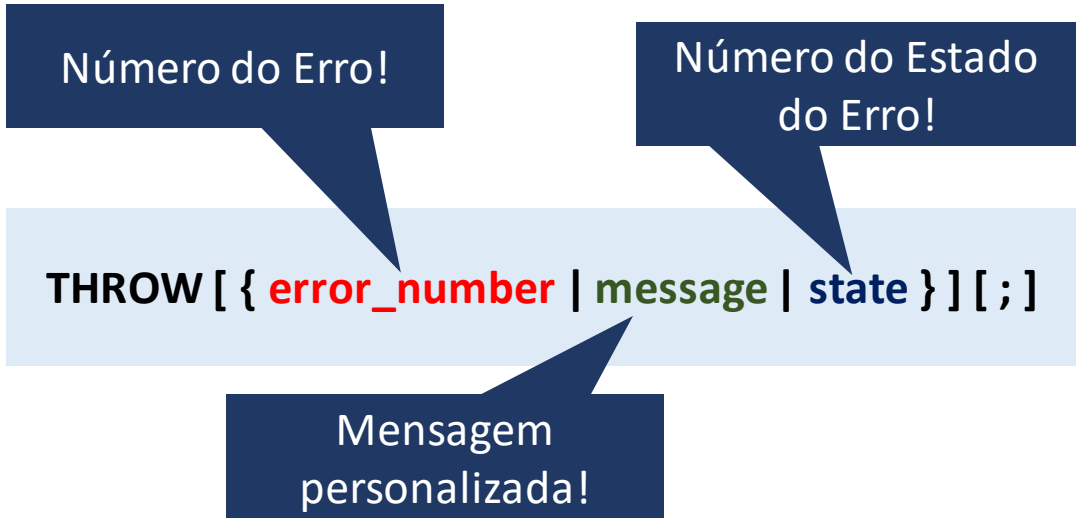
```
Mensagem 50000, Nível 16, Estado 1, Linha 11
Há 1 nome(s)!
```

NÃO

```
Mensagem 50000, Nível 16, Estado 1, Linha 7
Nenhum nome começa com: And
```

Comando **THROW** *VS* **RAISERROR**

- **THROW é um novo comando** disponível apenas **a partir da versão 2012 do SQL Server**;
- É **mais fácil de ser usado** do que o RAISERROR, além de ser **mais customizável**;
- Usado quando o SQL Server **não tem mensagens claras** para o erro;
- No **THROW**, por padrão, o valor da *gravidade sempre será 16*.



Usando o Comando **THROW**

Verifica se existe o funcionário com **idFunc = 1**

```
IF EXISTS(SELECT * FROM Funcionario WHERE idFunc=1)
    THROW 50000,'Usuário com ID 1 já existe!',1;
```

```
Mensagem 50000, Nível 16, Estado 1, Linha 25
Usuário com ID 1 já existe!
```

Se o funcionário existir apresenta uma mensagem!

Já existe um funcionário com o **idFunc = 1**

```
BEGIN TRY
    INSERT INTO Funcionario (idFunc,nome)
                VALUES(1,'Teddy');
END TRY
BEGIN CATCH
    THROW 50001,'Violação de Chave Primária!',1;
END CATCH
```

```
(0 linhas afetadas)
Mensagem 50001, Nível 16, Estado 1, Linha 20
Violação de Chave Primária!
```

Apresenta erro de violação de chave primária!

Mensagens Permanentes de Erro

- As **mensagens de erro do SQL Server** estão armazenadas por padrão na tabela ***sys.messages***;
- Ao **adicionar uma mensagem nessa tabela** podemos **reutilizá-la posteriormente**;
- As **principais SP que manipulam os seus dados** são: *sp_dropmessage* e *sp_addmessage*;
- As novas mensagens devem **começar a partir do valor 50.000**, *pois o restante já está em uso.*

Criando Mensagens Permanentes

Verifica se a mensagem existe e apaga se necessário!

```
IF EXISTS(SELECT * FROM sys.messages WHERE message_id=50002)
    EXEC sp_dropmessage 50002,@LANG='us_english';
GO
```

Cria uma nova mensagem para uso posterior!

```
PRINT 'Criando mensagem personalizada...'
EXEC sp_addmessage 50002, 16, N'Registro não encontrado!',@LANG='us_english';
GO
```

Faz uso da mensagem criada anteriormente!

```
IF NOT EXISTS(SELECT * FROM Funcionario where idFunc=20)
    RAISERROR(50002,16,1);
```

```
Criando mensagem personalizada...
Mensagem 50002, Nível 16, Estado 1, Linha 40
Registro não encontrado!
```

Tabelas Temporárias

- São estruturas **usadas por um curto período de tempo** em scripts T-SQL;
- Elas ficam no **BD do sistema tempdb** e não no BD do usuário;
- Somente a **conexão que criou a tabela temporária** *é capaz de enxerga-la e manipula-la*;
- Esse tipo de tabela pode **conter tudo que uma tabela convencional possui**: *chaves, índices, etc.*

```
CREATE TABLE #NomeTabela
(
    <coluna 1> <tipo de dado>,
    <coluna 2> <tipo de dado>
)
```

Usando uma Tabela Temporária
(Continua)

#
Cria uma tabela temporária local.

```
CREATE TABLE #ProdutosPorCategoria
(
    id int identity primary key,
    categoria varchar(100) not null,
    produto varchar(100) not null
)
GO
```

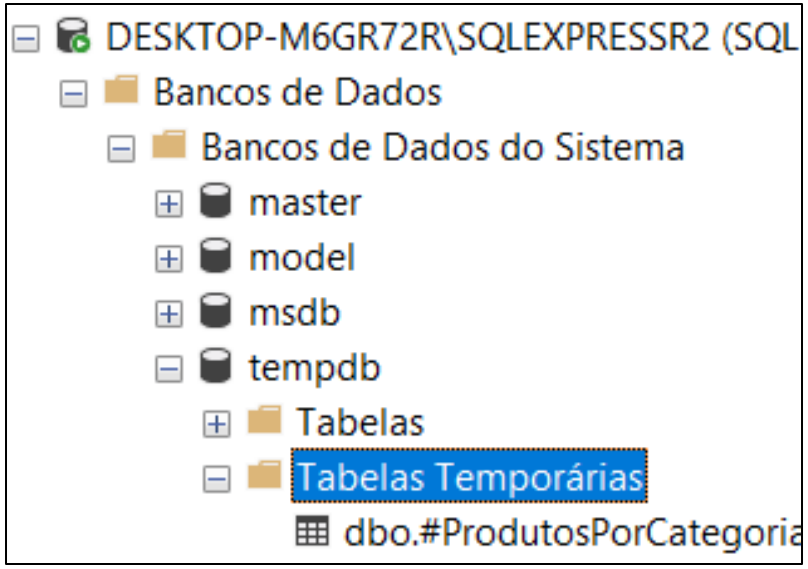
Usando uma Tabela Temporária
(Continua)

Populando a tabela temporária criada!

```
INSERT INTO #ProdutosPorCategoria(categoria,produto)
SELECT A.CategoryName,B.ProductName
FROM Categories AS A LEFT JOIN Products AS B
ON A.CategoryID = B.CategoryID
GO
```

Usando uma Tabela Temporária
(Continua)

```
SELECT * FROM #ProdutosPorCategoria;
GO
```

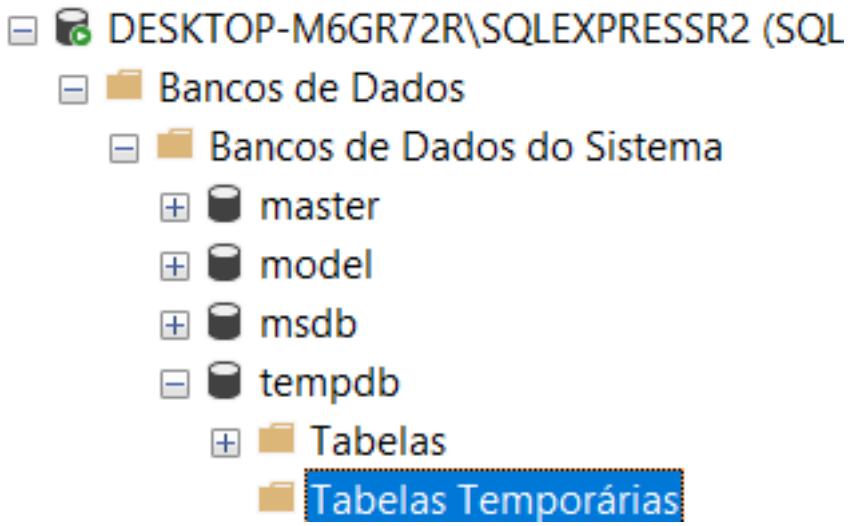


id	categoria	produto
1	Beverages	Chai
2	Beverages	Chang
3	Beverages	Guaraná Fantástica
4	Beverages	Sasquatch Ale
5	Beverages	Steeleye Stout
6	Beverages	Côte de Blaye
7	Beverages	Chartreuse verte
8	Beverages	Ipoh Coffee
9	Beverages	Laughing Lumberjack Lager
10	Beverages	Outback Lager
11	Beverages	Rhönbräu Klosterbier
12	Beverages	Lakkalikööri
13	Condiments	Aniseed Syrup
14	Condiments	Chef Anton's Cajun Seasoning
15	Condiments	Chef Anton's Gumbo Mix
16	Condiments	Grandma's Boysenberry Spread
17	Condiments	Northwoods Cranberry Sauce
18	Condiments	Genen Shouyu
19	Condiments	Gula Malacca
20	Condiments	Sirop d'érable

Usando uma Tabela Temporária

Ao **fechar a conexão** isso já é feito **automaticamente**, porém trata-se de uma **boa prática!**

```
DROP TABLE #ProdutosPorCategoria;
```



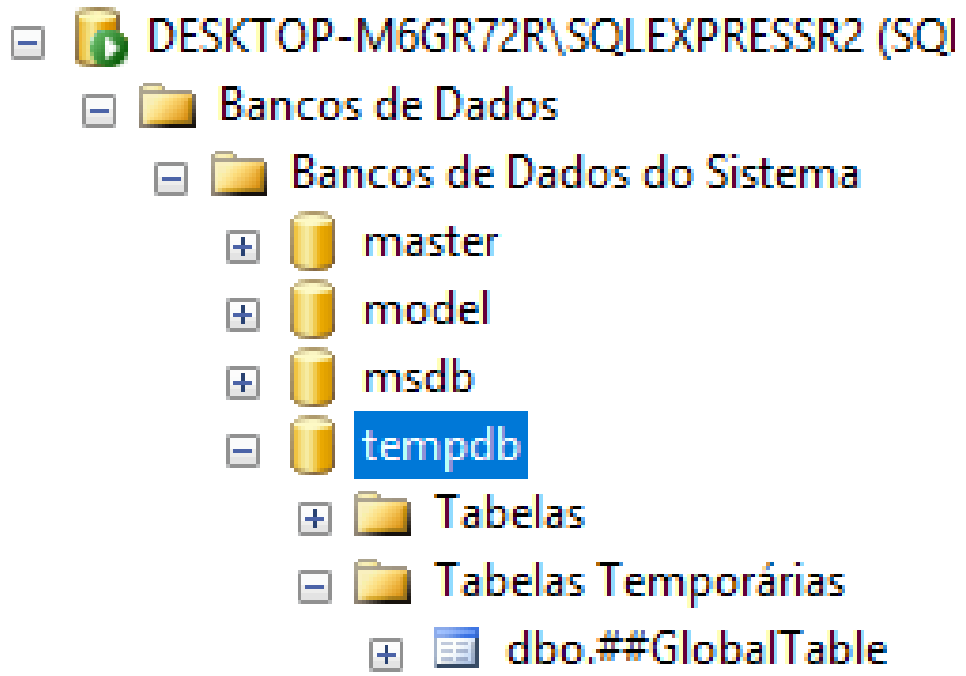
Tabelas Temporárias Globais

- Nas tabelas temporárias locais, o acesso se dá somente de dentro da conexão que a criou;
- Já nas **tabelas temporárias globais**, *qualquer conexão é capaz de ver e manipular a tabela*;
- O mecanismo de banco de dados **apaga a tabela quando a sua última conexão for fechada**;
- *Não há muitas razões para se usar uma tabela temporária global, devido a questões de segurança.*

Criando uma Tabela Temporária Global

##
Cria uma tabela temporária global

```
CREATE TABLE ##GlobalTable
(
    id int identity primary key,
    ativo bit not null,
    descricao varchar(50)
)
```



Mãos à Obra!!!

Envie por e-mail conforme o padrão apresentado na Aula 1