

# Armazenamento de Dados

Os dados podem ser armazenados em vários locais diferentes, que são classificados de acordo com sua velocidade de acesso e sua disponibilidade.

- Memória RAM
- Disco SSD
- Disco Magnético
- Disco Ótico
- Fitas magnéticas



# Armazenamento de Dados

## HIERARQUIA

### Armazenamento Primário

- Trabalhado diretamente pela CPU

Exemplos: memória RAM, cache

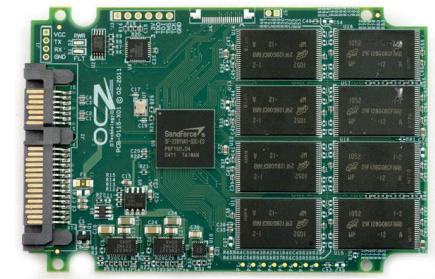
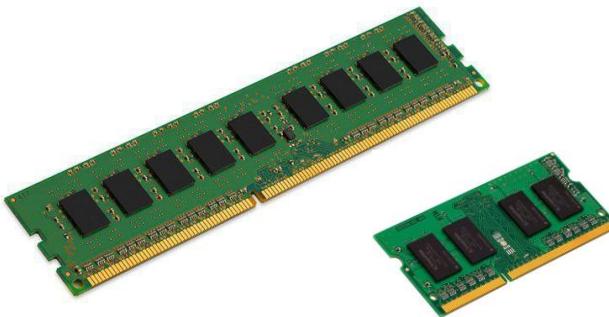
### Armazenamento Secundário

- Geralmente mais lento e, consequentemente, mais barato
- Não é trabalhado diretamente pela CPU
- Necessária intermediação do armazenamento primário

Exemplos: Fita magnética, discos

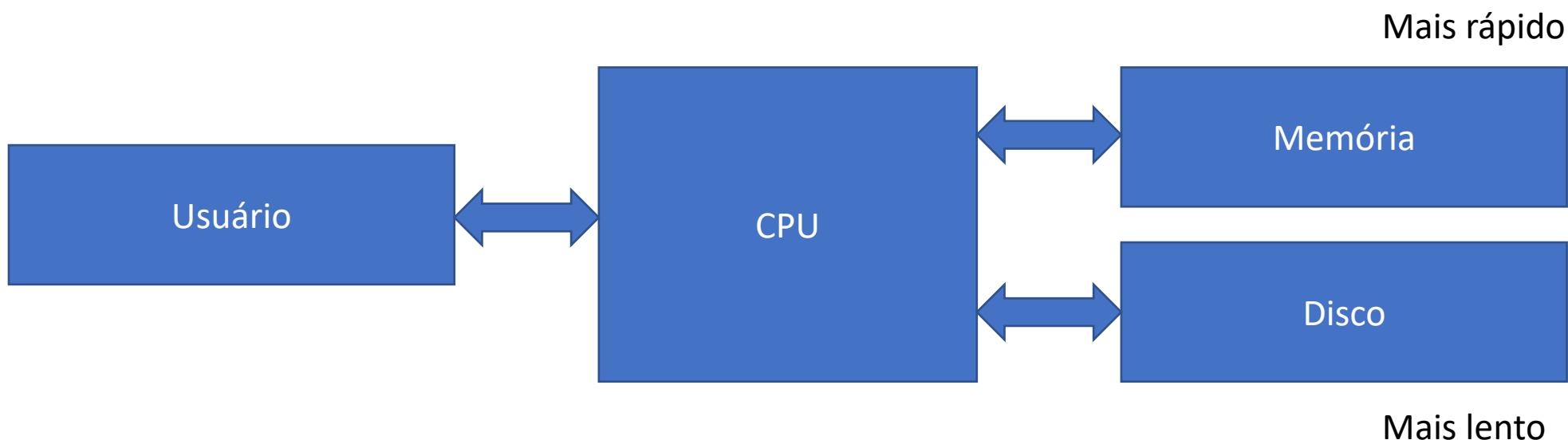
# Armazenamento de Dados

## TIPOS



# Armazenamento de Dados

## PROCESSAMENTO / MANIPULAÇÃO



# Índices

## DEFINIÇÃO

Um **ÍNDICE** é uma estrutura em disco associada a uma tabela (ou view), que agiliza a recuperação dos dados.

Um índice contém chaves criadas de uma ou mais colunas e essas chaves são armazenadas em uma estrutura (árvore B) que habilita o SQL Server a localizar os registros associadas aos valores de chave de forma rápida e eficaz.

- Estrutura de dados
- Organiza registros
- Otimiza certas operações de recuperação

### Dicionário

Definições de [Oxford Languages](#) · [Saiba mais](#)

Pesquise uma palavra



### indexação

/cs/

*substantivo feminino*

ação ou efeito de indexar.

- **BIBLIOLOGIA**

ordenação em forma de índice; classificação.

# índices

## DEFINIÇÃO

Dados sem ordenação / classificação



# índices

## DEFINIÇÃO

Dados com ordenação / classificação por um critério (chave)



# índices

## PORQUE?

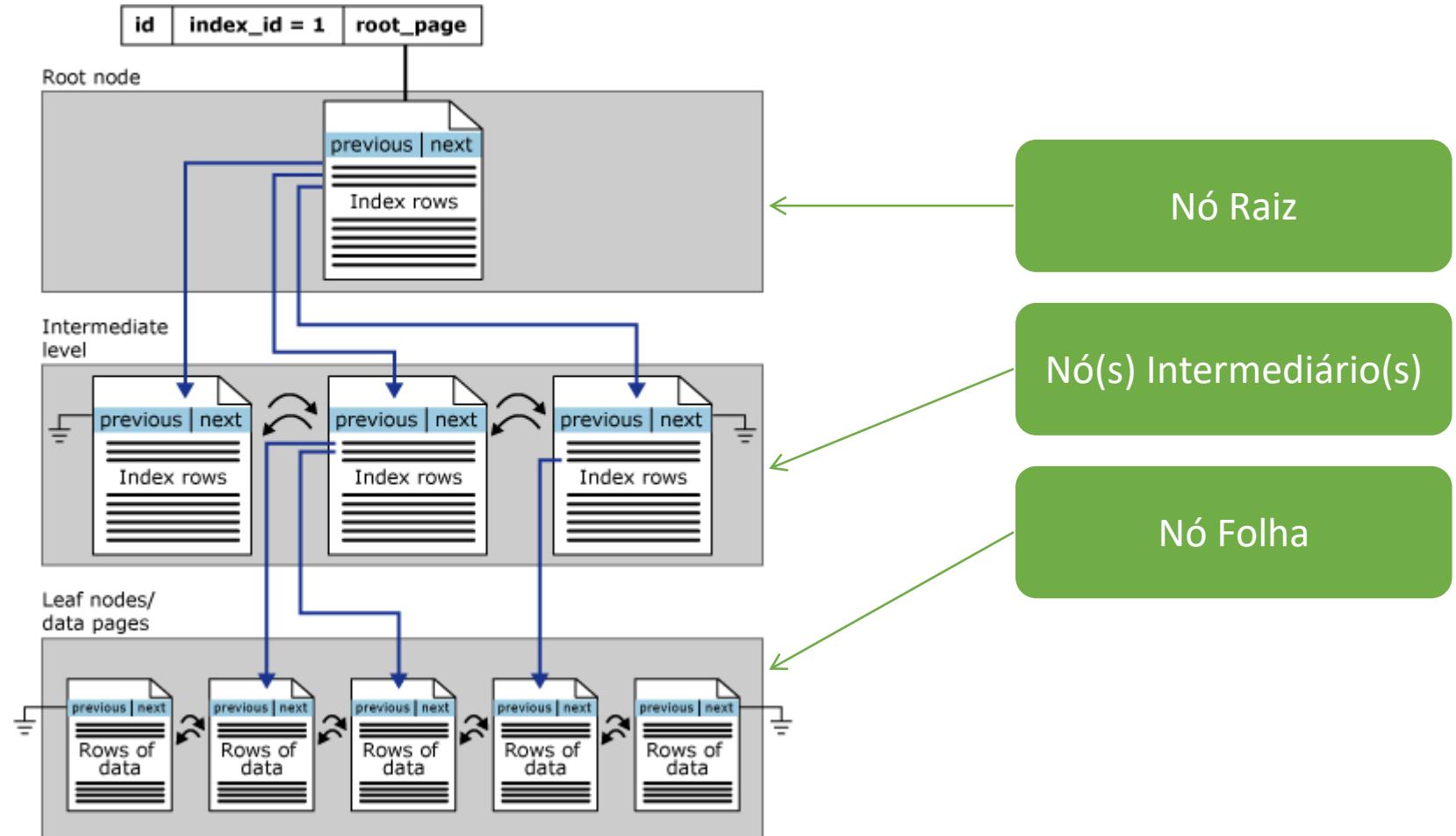
- Quando não conseguimos trabalhar no armazenamento primário, temos que usar o armazenamento secundário.
- Sabemos que o acesso aos dados no armazenamento secundário é lento.
- Precisamos de meios eficientes de acesso aos dados no armazenamento secundário (índices).

**Exemplos:** Procurar um produto em uma lista de materiais escolares X significado de uma palavra em um dicionário.

# Índices

## ÁRVORE BINÁRIA (B-TREE)

**Na maioria dos casos,  
um índice não terá mais  
que 4 ou 5 níveis.**



# Índices

## VIDEO



Breve introdução sobre índices - Heap, Clustered e Nonclustered



FabricioLima  
2,75 mil inscritos

Inscrito ▾

507



Compartilhar

...

<https://www.youtube.com/watch?v=lPwjhtHEfw0>

# índices

## TIPOS E CATEGORIAS DE INDICES

### Categorias de Índices:

- Clusterizado – Somente 1 por tabela.
- Não clusterizado – Posso ter mais de 1 por tabela.
- Únicos – Posso ter mais de 1 por tabela.

### Tipos:

- Simples – Somente 1 coluna.
- Composto – Mais de 1 coluna.

### Tipo especial:

- Cobertura – 1 ou mais colunas.

# índices

## QUANDO CRIAR INDICES?

**CRIE ÍNDICES** em colunas que suportam as seguintes operações em suas queries:

- Search arguments (SARGs) – WHERE
- Joins
- order by ou group by

MAIS IMPORTANTE

**NÃO CRIE ÍNDICES** para:

- Tabelas muito pequenas que podem ser mantidas em cache
- Tabelas médias e pequenas que não requerem acesso direto a linhas aleatórias

# Índices

## CONSIDERAÇÕES SOBRE INDICES

- Muitos índices podem ocupar mais espaço do que a própria tabela.
- Índices agilizam as consultas, mas deixam as operações de escritas mais lentas (INSERT, UPDATE, DELETE).
- Colunas mais seletivas são melhores que as menos seletivas (Data de Nascimento x Sexo)
- Considere a ordem das colunas para índices compostos.
- Elimine funções, operações aritméticas, e outras expressões sobre as colunas de tabelas na cláusula WHERE.

Exemplos:

- **WHERE SUBSTRING(au\_Iname, 1,3) = "Ger"** torna-se utilizável se escrita dessa forma:  
**WHERE au\_Iname LIKE "Ger%"**
- **WHERE price \* 3 > 3000** torna-se utilizável se escrita da forma: **WHERE price = 3000/3**

# índices

## CONSIDERAÇÕES SOBRE INDICES



De forma resumida, o que é coluna mais seletiva e coluna menos seletiva?

- **Coluna mais seletiva:** Contém valores muito diferentes entre si. Consultas usando essa coluna tendem a retornar poucos resultados.
- **Coluna menos seletiva:** Possui valores semelhantes ou repetidos. Consultas nessa coluna podem trazer muitos resultados.

Em resumo, a seletividade de uma coluna está relacionada à variedade de valores únicos que ela contém. Quanto mais valores únicos, mais seletiva é a coluna, e quanto menos valores únicos, menos seletiva é a coluna.

# índices

## SINTAXE – CRIAR INDICES

### Indice Clustered:

```
ALTER TABLE conta ADD CONSTRAINT pk_conta PRIMARY KEY (cd_conta);
```

### Indice NonClustered:

```
CREATE [NONCLUSTERED] INDEX idx1 ON conta (cpf);
```

### Indice Unique:

```
CREATE UNIQUE [NONCLUSTERED] INDEX idx2 ON conta (cpf);
```

### Indice NonClustered com Include (lookup):

```
CREATE [NONCLUSTERED] INDEX idx3 ON conta (cpf) INCLUDE (rg);
```

# índices

## SINTAXE – REMOVER INDICES

**Indice Clustered:**

```
ALTER TABLE conta DROP CONSTRAINT pk_conta;
```

**Indice NonClustered – todos os tipos:**

```
DROP INDEX idx1 ON conta;
```

```
DROP INDEX conta.idx1;
```

# índices

## PROCEDURES DE SISTEMA AUXILIARES

**Mostra todos os índices de uma tabela:**

```
sp_helpindex cliente;
```

index_name	index_description	index_keys
pk_cliente	clustered, unique, primary key located on PRIMARY	cod_cliente

**Mostra o espaço ocupado pelos dados e índices de uma tabela (ou banco de dados):**

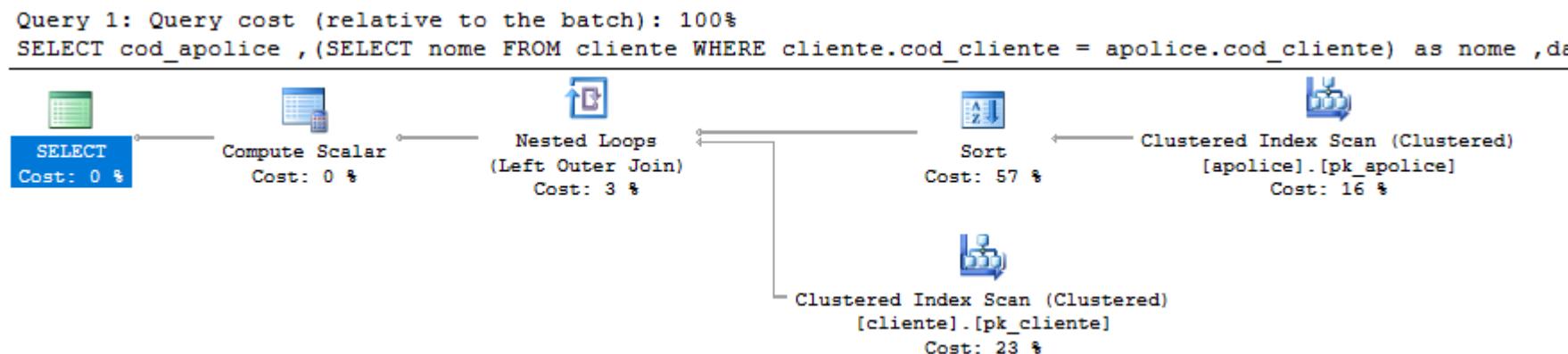
```
sp_spaceused cliente;
```

name	rows	reserved	data	index_size	unused
cliente	9	16 KB	8 KB	8 KB	0 KB

# Planos de Acesso SINTAXE

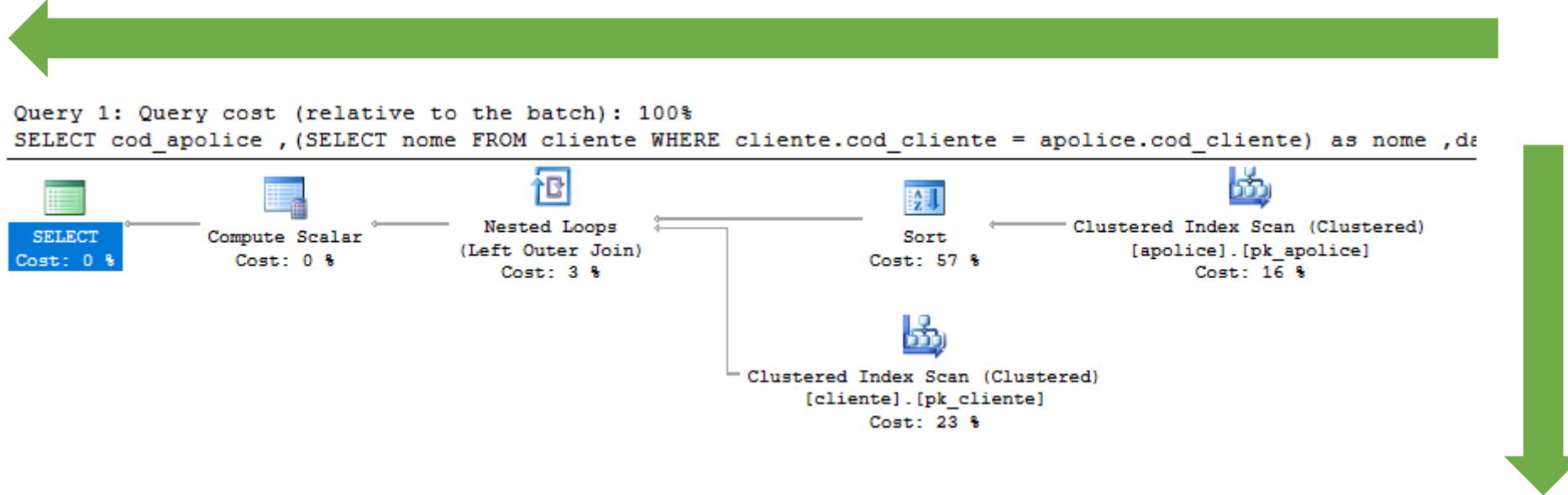
**O PLANO DE EXECUÇÃO** é uma ferramenta para auxiliar a entender como as consultas estão sendo realizadas no banco e interpretadas pelo otimizador de consultas.

Essa ferramenta é de extrema importância para um trabalho de otimização de consulta, pois entendendo como a consulta está sendo feita internamente é possível identificar pontos de melhoria.



# Planos de Acesso

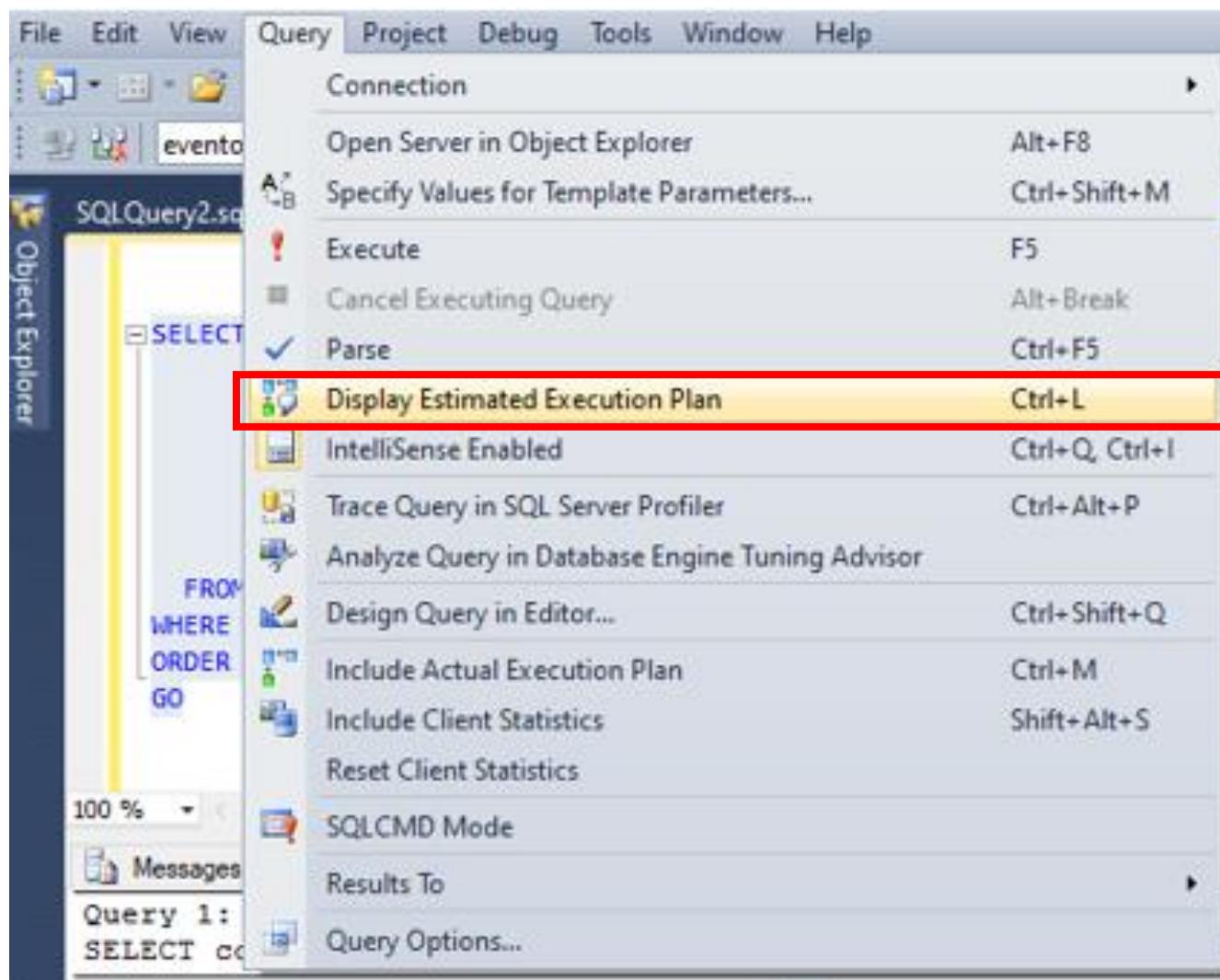
## LEITURA DE UM PLANO DE EXECUÇÃO



# Planos de Acesso

## VISUALIZANDO O PLANO DE EXECUÇÃO DE UMA CONSULTA

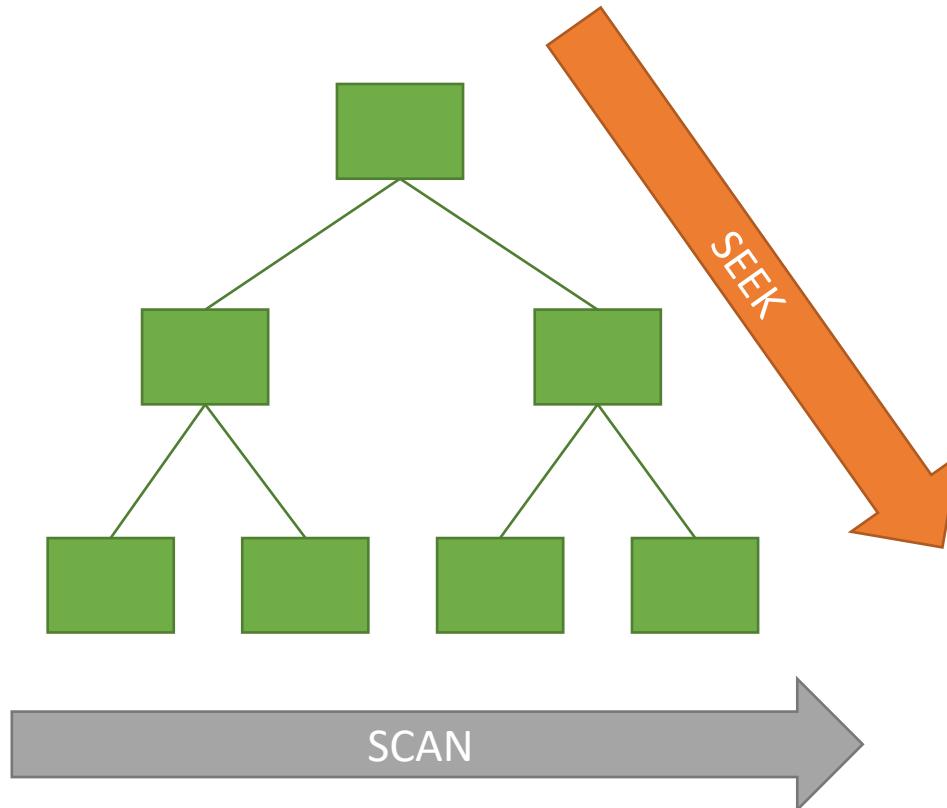
Através do SSMS:



# Planos de Acesso

## SEEK x SCAN

Operações de **SEEK** são geralmente mais rápidas que operações de **SCAN**.



# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Table Scan

- Varre todos os registros folha sequencialmente em busca dos dados.
- Ocorre em todas tabelas sem índice.
- Normalmente esta é uma operação de alto custo (lenta).

The screenshot shows the SQL Server Management Studio interface. In the query window, the following code is displayed:

```
SELECT * FROM MICRODADOS_ENEM_2021_SC
```

The execution plan window is open, showing the following details for Query 1:

Query cost (relative to the batch): 100%

```
SELECT * FROM MICRODADOS_ENEM_2021_SC
```

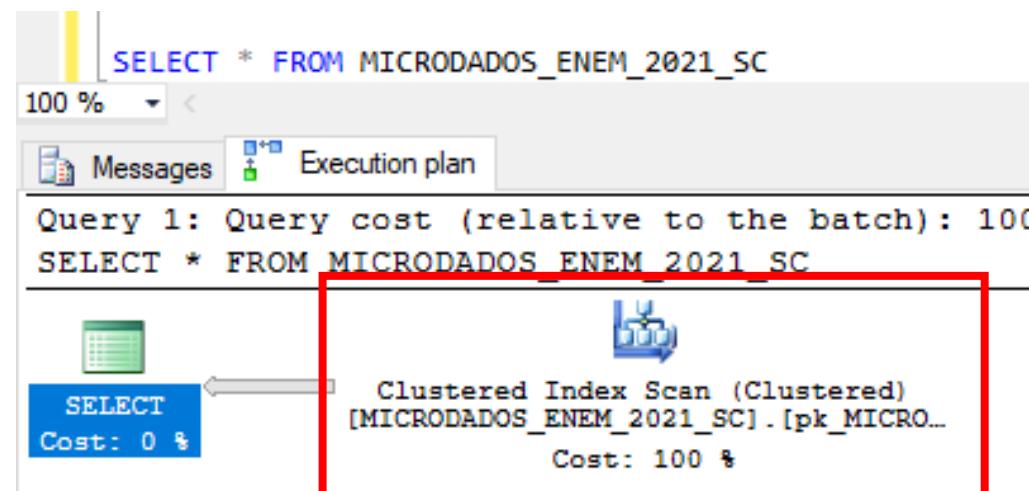
The execution plan diagram shows a single operator node labeled "Table Scan [MICRODADOS\_ENEM\_2021\_SC]". This node is highlighted with a red box. An arrow points from the "SELECT Cost: 0 %" node to the "Table Scan" node.

# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Clustered Index Scan

- Varre todos os registros folha sequencialmente em busca dos dados.
- Ele é muito similar ao Table Scan, só que um pouco mais rápido, pois os dados já estão ordenados no índice (clustered).



# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Clustered Index Seek e NonClustered Index Seek

- Percorre a árvore através dos nós até o registro folha (mais eficiente para retornar registros específicos).

The screenshot shows the SQL Server Management Studio interface with a query window and an execution plan viewer.

Query window content:

```
SELECT * FROM MICRODADOS_ENEM_2021_SC where NU_INSCRICAO = 210051014600
```

Execution plan details:

Query 1: Query cost (relative to the batch): 100%

```
SELECT * FROM MICRODADOS_ENEM_2021_SC where NU_INSCRICAO = 210051014600
```

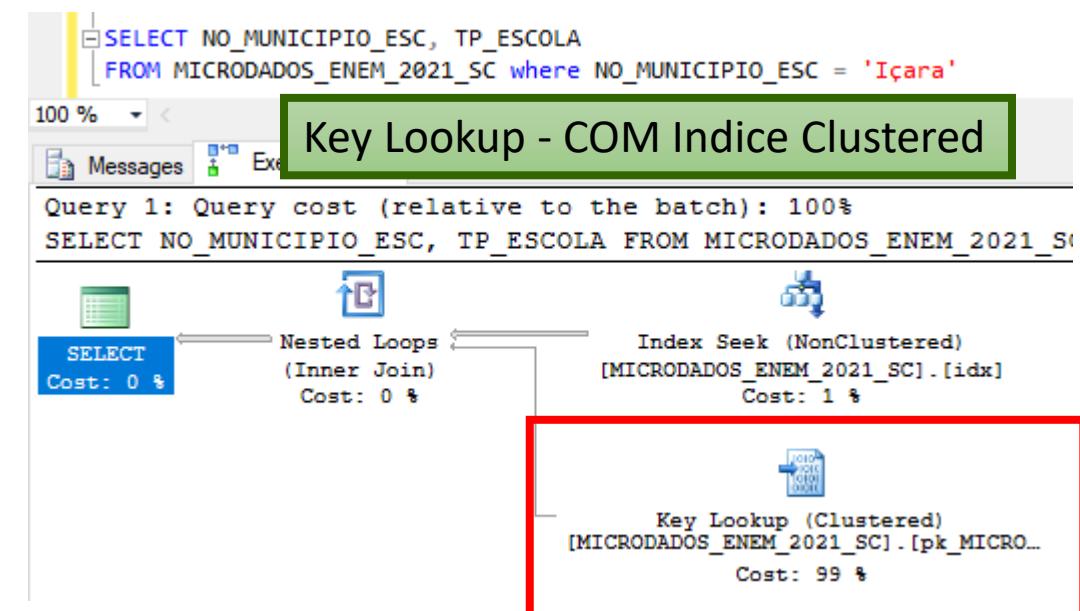
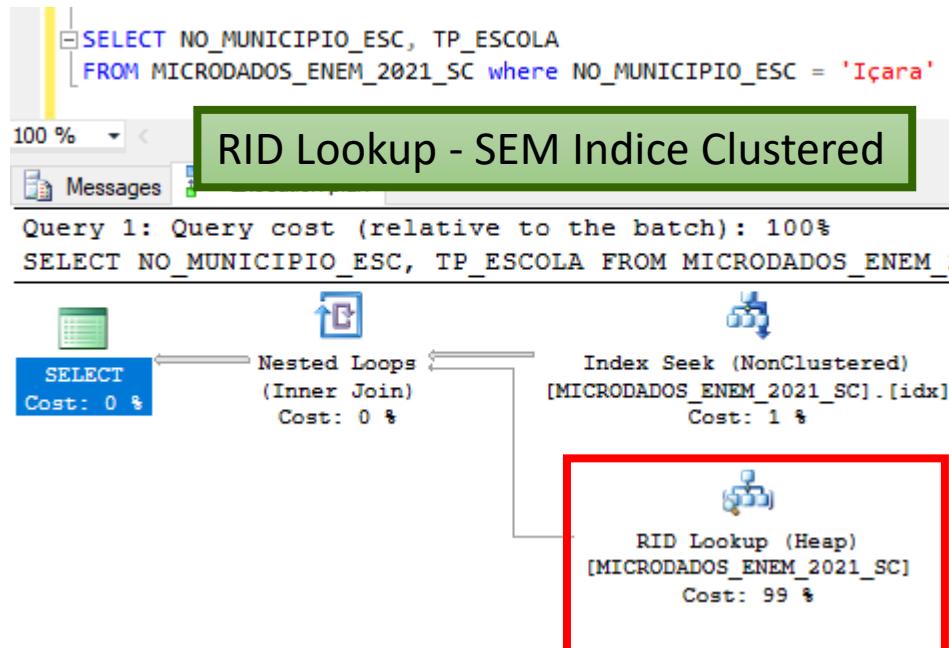
The execution plan shows a **Clustered Index Seek (Clustered)** operation on the **[MICRODADOS\_ENEM\_2021\_SC].[pk\_MICRO...]** index. This step is highlighted with a red box. The cost for this step is 100 %.

# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Key Lookup e RID Lookup

- Operações de Lookup ocorrem quando o dado não é encontrado no índice nonclustered e precisa ir ao índice clustered para encontrá-lo.
- Geralmente ocorrem 2 operações de leituras para uma cada item.

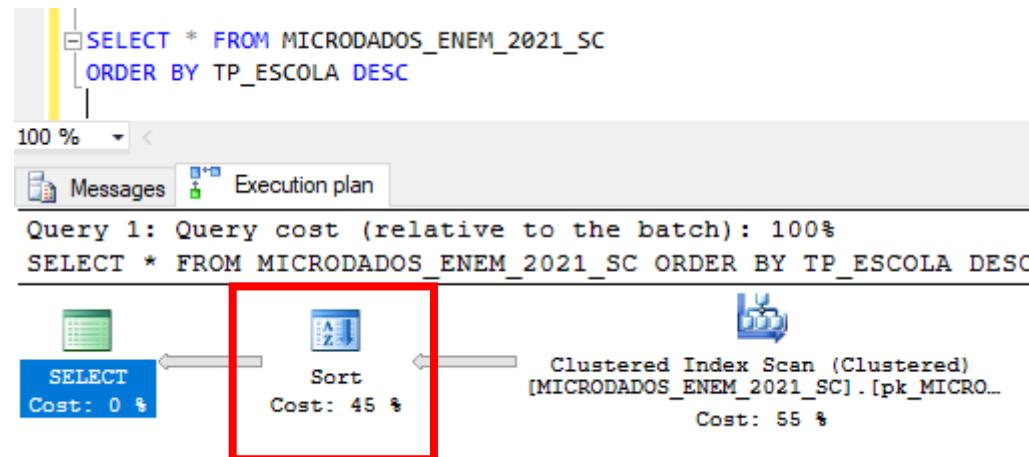


# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Sort

- Operador que costuma ser muito pesado em grandes volumes de dados.
- Ocorrem com operações de ORDER BY ou DISTINCT.

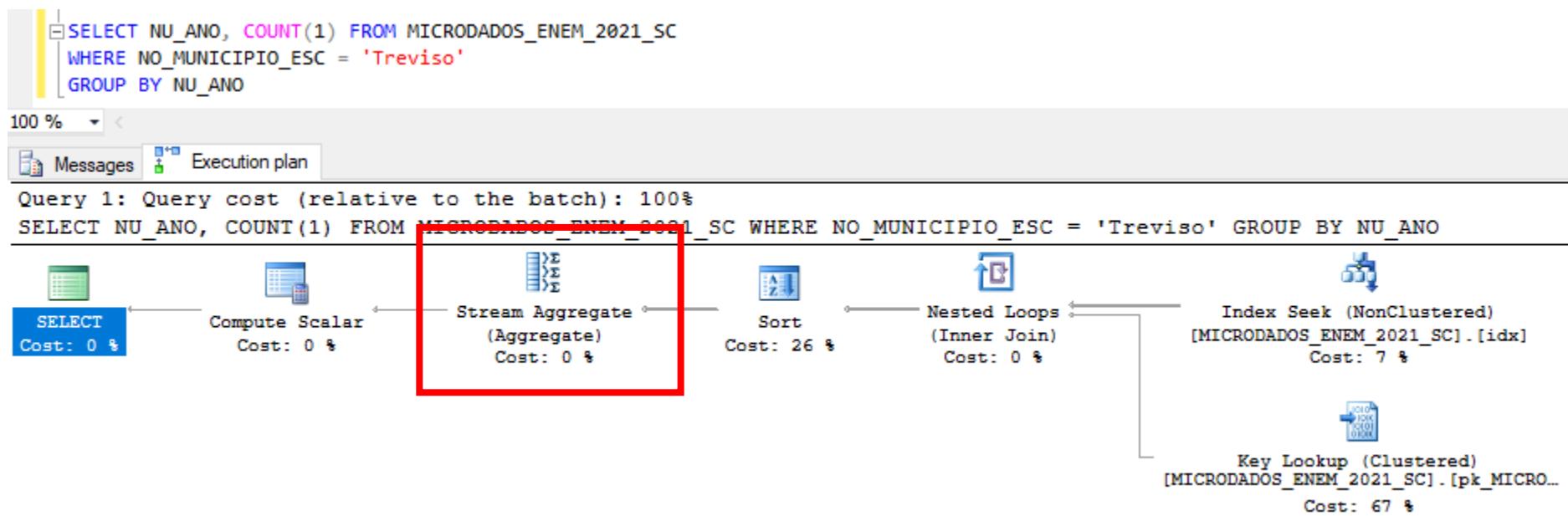


# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Stream Aggregate

- Ocorrem em consultas com agrupamento (GROUP BY, DISTINCT, etc).

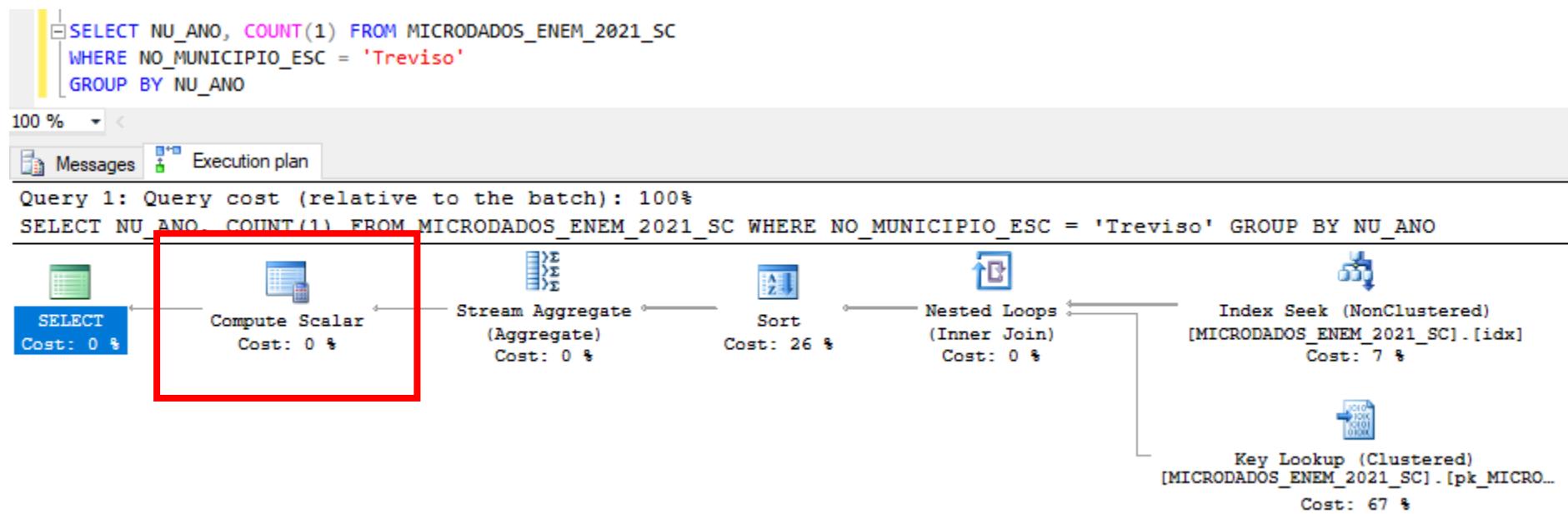


# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Compute Scalar

- Ocorre quando utilizamos consultas com expressões, funções, cálculos matemáticos ou conversões (CAST, CONVERT)

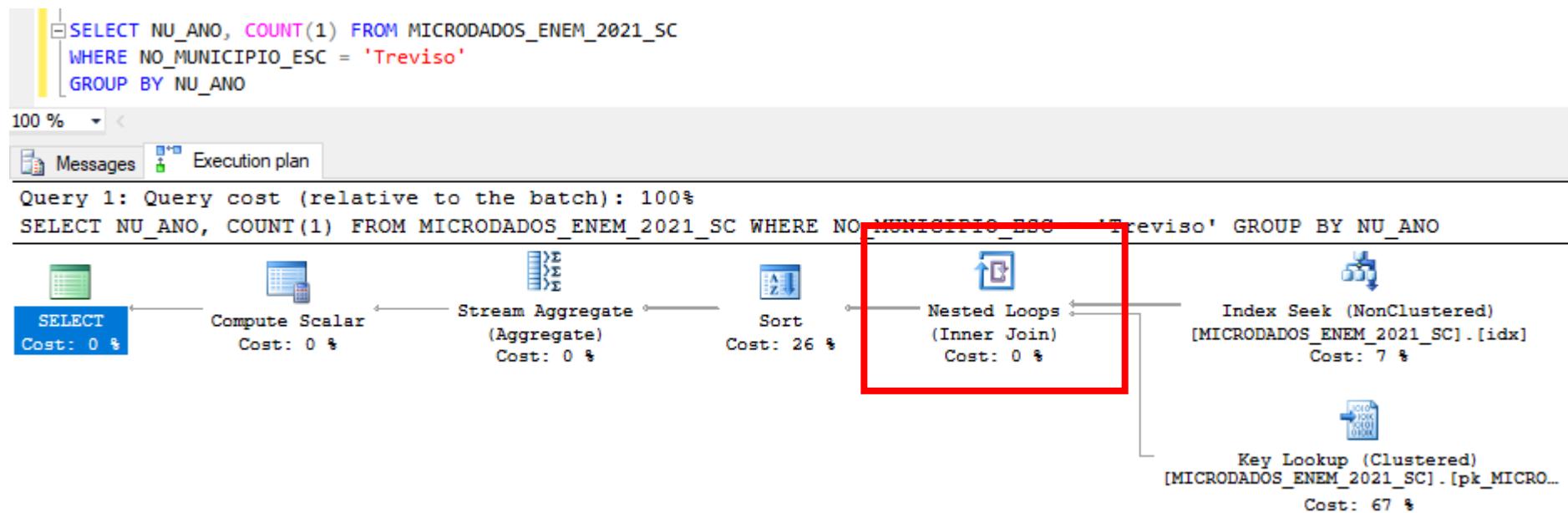


# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Nested Loops

- Ocorre em operações de Junção (JOIN) em cenários de poucos registros.
- Muito eficiente.

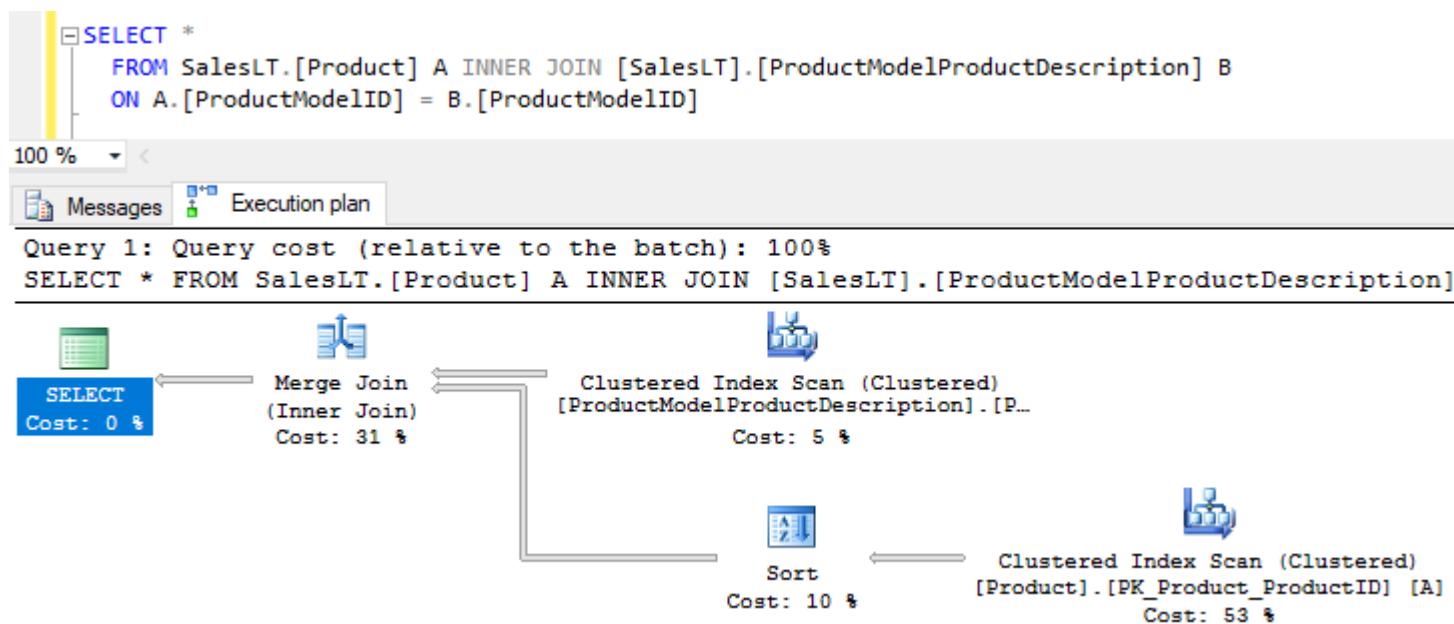


# Planos de Acesso

## OPERADORES DO PLANO DE EXECUÇÃO

### Merge Join

- Ocorre em cenários com muitos registros.
- Merge Join consome mais que o Nested Loop, pois os matches das junções são feitos em memória e o volume de dados costuma ser maior também.



# Planos de Acesso

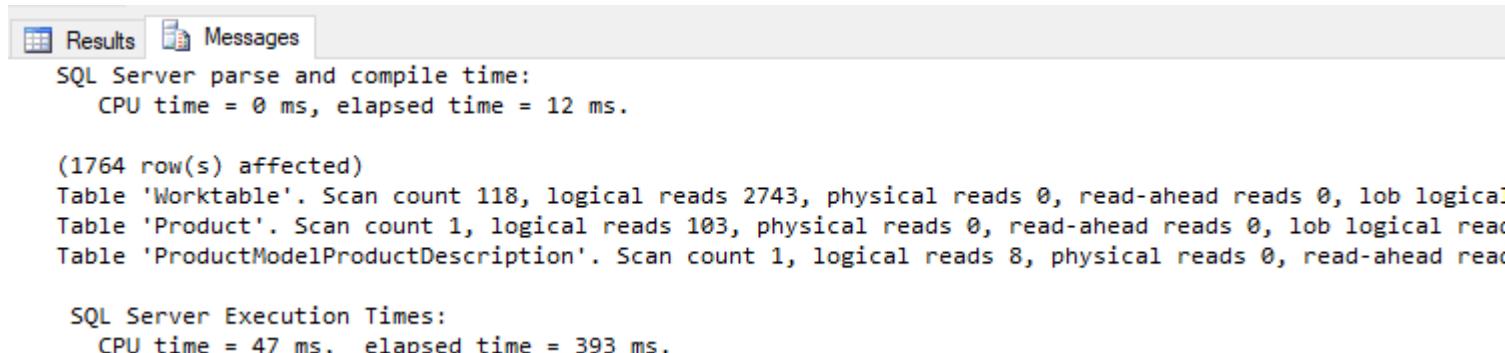
## SET STATISTICS

### STATISTICS IO E STATISTICS TIME

Usados para medir a quantidade de leituras/escritas que são realizadas para cada objeto consultado, bem como o tempo de resposta de cada operação.

```
set statistics io on; --on liga e off desliga  
set statistics time on; --on liga e off desliga
```

```
SELECT *  
  FROM SalesLT.[Product] A INNER JOIN [SalesLT].[ProductModelProductDescription] B  
    ON A.[ProductModelID] = B.[ProductModelID];
```



The screenshot shows the SQL Server Management Studio interface with the 'Results' tab selected. The output window displays the results of the executed SQL query, including parse and compile times, row count affected, and detailed I/O statistics for three tables: Worktable, Product, and ProductModelProductDescription. It also shows the execution times for CPU and elapsed time.

```
SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 12 ms.  
  
(1764 row(s) affected)  
Table 'Worktable'. Scan count 118, logical reads 2743, physical reads 0, read-ahead reads 0, lob logical  
Table 'Product'. Scan count 1, logical reads 103, physical reads 0, read-ahead reads 0, lob logical read  
Table 'ProductModelProductDescription'. Scan count 1, logical reads 8, physical reads 0, read-ahead read  
  
SQL Server Execution Times:  
CPU time = 47 ms, elapsed time = 393 ms.
```

# Planos de Acesso

## DICAS ANÁLISE DE DESEMPENHO

### Algumas dicas para resolução de problemas em análise de desempenho:

- Nunca use \* (SELECT \* FROM...).
- Lookup: Resolva com INCLUDE.
- Index Scan: Valide o uso das colunas na cláusula WHERE e na lista de SELECT.
- Cuidado com o uso de funções e operações matemáticas na WHERE. Elas podem ser ruins para a sua consulta.
- Use de SET STATISTICS IO e TIME, para medir a quantidade de leituras/escritas físicas e lógicas que são realizadas para cada objeto, bem como o tempo de resposta de cada operação.
- Em muitos casos, a estrutura do banco já está otimizada, mas a query foi mal escrita. Sendo assim, a consulta pode deverá ser ajustada para contemplar o uso dos índices existentes.
- Efetue rebuild dos índices e atualização de suas estatísticas frequentemente.

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios Linguagem SQL 3”**

Utilizando os itens vistos em aula:

**ÍNDICES e PLANOS DE ACESSO.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

**Material de apoio:**

*Breve introdução sobre índices - Heap, Clustered e Nonclustered*

<https://www.youtube.com/watch?v=lPwjhtHEfw0>

# Functions

**FUNCTIONS ou FUNÇÕES** como funções em linguagens de programação, funções no SQL Server são rotinas que aceitam parâmetros, executam ações (rotinas, cálculos complexos, etc) e retornam um resultado.

Existem 3 tipos de funções:

## Escopo de estudo

- **Escalar:** Retornam um valor único.
- **Table-Valued:** Retornam uma lista de resultados.
- **Sistema:** Não podem ser alteradas, utilizadas para diversas operações (agregação, matemática, string, etc).

# Functions

## DIFERENÇA ENTRE STORED PROCEDURE E FUNCTIONS

- **Funções** podem ser utilizadas em comandos de SELECT, diferentemente de uma **stored procedure**.
- **Stored procedures** aceitam parâmetros de entrada e saída, **funções** somente de entrada.
- **Funções** sempre retornam um valor, **stored procedure** podem ou não retornar um valor.
- Em **stored procedures** é possível trabalhar com controle transacional (begin tran, commit, rollback) em **funções** não é possível.

# Functions

## CRIANDO UMA FUNÇÃO

### SINTAXE

```
create function function_name (@par1 type, @par2 type, ... ) returns int as
begin
    sql_statements

    return @parameter
end
go
```

### EXEMPLO

```
create function fn_idade (@data datetime) returns int as
begin
    declare @idade int

    select @idade = floor(datediff(day, @data, getdate()) / 365.25)

    return @idade
end
go
```

# Functions

## EXCLUINDO UMA FUNÇÃO

### SINTAXE SIMPLIFICADA

```
drop function function_name
```

### EXEMPLO

```
drop function fn_idade
```

### VALIDANDO A EXISTÊNCIA DA FUNCTION ANTES DE EXCLUIR

```
drop function if exists fn_idade  
go
```

# Functions

## CHAMANDO UMA FUNÇÃO

### SINTAXE

```
select schema.function_name(parametros)  
[ from table_name ... ]
```

Sempre necessário informar o nome do schema ao chamar uma função escalar.

### EXEMPLO

```
select top 5 cd_paciente,  
       nm_paciente,  
       dbo.fn_idade(dt_nascimento) as idade  
from paciente
```

	cd_paciente	nm_paciente	idade
0		PACIENTE NÃO CADASTRADO	NULL
1		MARILENE	58
2		ALINE	33
3		AMARILDO	58
4		ANA	18

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios funções (functions)”**

Utilizando os itens vistos em aula:

**PROGRAMAÇÃO T-SQL, CURSORES e FUNCTIONS.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

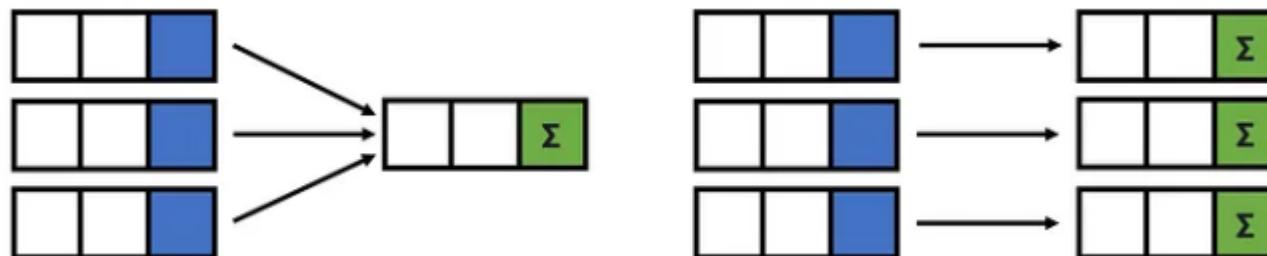
# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

As **FUNÇÕES DE JANELA** são similares às agregações realizadas através de operações com **GROUP BY**.

Com **GROUP BY** as linhas são agrupadas e summarizadas de acordo com a função de agregação escolhida.

Com **FUNÇÕES DE JANELA** as linhas não são agrupadas, ela mantem todas as linhas do resultado (sem agrupar as linhas) e gera uma nova coluna com o resultado da agregação escolhida.



# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

```
SELECT * FROM apolice  
ORDER BY placa;
```

cod_apolice	cod_cliente	data_inicio_vigencia	data_fim_vigencia	valor_cobertura	valor_franquia	placa
202200011	2	2022-10-24	2023-12-24	7865.55	89.16	ALD3834
202200015	4	2022-10-24	2024-02-24	16261.87	180.20	ALD3834
202200009	6	2022-10-24	2024-03-24	17561.01	169.48	ALD3834
202200014	4	2022-10-24	2022-12-24	15040.52	161.51	GQY6753
202200008	7	2022-10-24	2023-10-24	6815.28	145.16	IAC8974
202200013	7	2022-10-24	2024-03-24	2737.30	25.81	IAC8974
202200006	7	2022-10-24	2023-02-24	12595.89	20.45	JIE0952
202200016	3	2022-10-24	2023-06-24	15760.31	132.84	JIE0952
202200003	3	2022-10-24	2024-02-24	19456.46	146.99	JIE0952
202200004	3	2022-10-24	2022-10-24	4615.60	47.77	LVX7086
202200012	1	2022-10-24	2023-05-24	19970.84	157.80	LVX7086
202200005	3	2022-10-24	2023-10-24	19130.12	181.57	LWJ9156
202200007	2	2022-10-24	2023-11-24	19509.51	61.00	MZT1826
202200001	1	2021-10-24	2022-02-24	2565.25	100.25	MZT1826
202200010	1	2021-10-24	2022-08-24	9425.25	68.64	NAP5760
202200002	7	2022-10-24	2023-02-24	16081.90	28.79	NEM5116

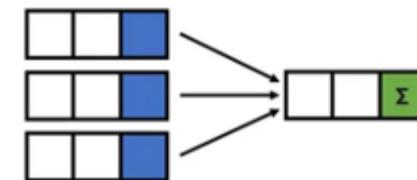
# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

```
SELECT placa,  
       COUNT(placa) AS qtde  
FROM apolice  
GROUP BY placa  
ORDER BY qtde DESC
```

placa	qtde
ALD3834	3
JIE0952	3
LVX7086	2
MZT1826	2
IAC8974	2
NAP5760	1
NEM5116	1
LWJ9156	1
GQY6753	1

GROUP BY



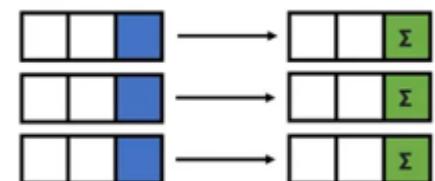
# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

```
SELECT *,  
       COUNT(placa) OVER(PARTITION BY placa) as qtde  
FROM apolice  
ORDER BY qtde DESC
```

cod_apolice	cod_cliente	data_inicio_vigencia	data_fim_vigencia	valor_cobertura	valor_franquia	placa	qtde
202200011	2	2022-10-24	2023-12-24	7865.55	89.16	ALD3834	3
202200015	4	2022-10-24	2024-02-24	16261.87	180.20	ALD3834	3
202200009	6	2022-10-24	2024-03-24	17561.01	169.48	ALD3834	3
202200006	7	2022-10-24	2023-02-24	12595.89	20.45	JIE0952	3
202200016	3	2022-10-24	2023-06-24	15760.31	132.84	JIE0952	3
202200003	3	2022-10-24	2024-02-24	19456.46	146.99	JIE0952	3
202200004	3	2022-10-24	2022-10-24	4615.60	47.77	LVX7086	2
202200012	1	2022-10-24	2023-05-24	19970.84	157.80	LVX7086	2
202200007	2	2022-10-24	2023-11-24	19509.51	61.00	MZT1826	2
202200001	1	2021-10-24	2022-02-24	2565.25	100.25	MZT1826	2
202200008	7	2022-10-24	2023-10-24	6815.28	145.16	IAC8974	2
202200013	7	2022-10-24	2024-03-24	2737.30	25.81	IAC8974	2
202200010	1	2021-10-24	2022-08-24	9425.25	68.64	NAP5760	1
202200002	7	2022-10-24	2023-02-24	16081.90	28.79	NEM5116	1
202200005	3	2022-10-24	2023-10-24	19130.12	181.57	LWJ9156	1
202200014	4	2022-10-24	2022-12-24	15040.52	161.51	GQY6753	1

### WINDOW FUNCTIONS



# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

### POR QUE USAR?

Uma grande vantagem das funções de janela é que elas **permitem que você trabalhe com valores agregados e não agregados de uma só vez** porque as **linhas não são agrupadas** (recolhidas juntas).

As funções da janela também são simples de usar e ler. Dessa forma, eles podem **reduzir a complexidade de suas consultas**, o que facilita a manutenção no futuro.

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

FUNÇÕES DE JANELA são permitidas no SELECT e no ORDER BY.  
Não são permitidas em cláusulas FROM, WHERE, GROUP BY ou HAVING.

### ORDEM REAL DE EXECUÇÃO

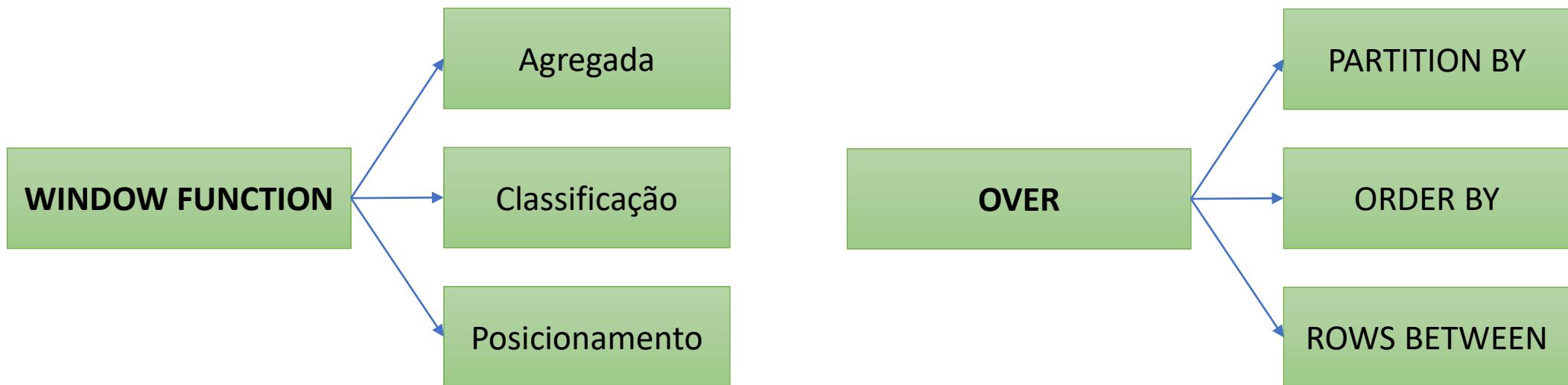
1. FROM JOIN
2. WHERE
3. GROUP BY
4. AGGREGATE FUNCTIONS
5. HAVING
- 6. WINDOW FUNCTIONS**
7. SELECT
8. DISTINCT
9. UNION
10. ORDER BY
11. TOP

**IMPORTANTE:** Se você precisa o resultado de uma FUNÇÃO DE JANELA dentro de uma WHERE ou GROUP BY, por exemplo, você pode usar uma CTE.

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

```
SELECT <coluna1>, <coluna2>, <coluna3>, *  
      <WINDOW FUNCTION> OVER ( PARTITION BY <colunas> ORDER BY <colunas> ROWS  
      BETWEEN <opcao_rows_between> AND <opcao_rows_between>)  
FROM <tabela>
```



# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

Existem três tipos de **FUNÇÕES DE JANELA**:

### AGREGADA

- AVG()
- MAX()
- MIN()
- SUM()
- COUNT()

### CLASSIFICAÇÃO

- ROW\_NUMBER()
- RANK()
- DENSE\_RANK()
- PERCENT\_RANK()
- NTILE()

### POSICIONAMENTO

- LAG()
- LEAD()
- FIRST\_VALUE()
- LAST\_VALUE()
- NTH\_VALUE()

**Analíticas  
Valores**

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

**FUNÇÕES AGREGADAS:** podemos usar essas funções para calcular várias agregações, como média, número total de linhas, valores máximos ou mínimos ou soma total dentro de cada janela ou partição.

**FUNÇÕES DE CLASSIFICAÇÃO:** essas funções são usadas para classificar linhas dentro de sua partição. PARTITION BY opcional, ORDER BY obrigatório.

**FUNÇÕES DE POSICIONAMENTO:** essas funções permitem selecionar valores de linhas anteriores ou seguintes dentro da partição ou o primeiro ou último valor dentro da partição.

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

As funções da janela podem ter os seguintes argumentos na cláusula **OVER**:

**PARTITION BY** divide o conjunto de resultados da consulta em partições.

**ORDER BY** define a ordem lógica das linhas dentro de cada partição do conjunto de resultados.

**ROWS BETWEEN** limita as linhas dentro da partição com a especificação de pontos iniciais e finais na partição. Ele requer o argumento ORDER BY para ordenar as linhas dentro das especificações do **ROWS BETWEEN**.

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

### EXEMPLOS DE FUNÇÕES AGREGADAS

```
SELECT cod_apolice, placa,  
       COUNT(placa) OVER() as qtde  
FROM apolice  
order by placa  
;
```

cod_apolice	placa	qtde
202200011	ALD3834	16
202200015	ALD3834	16
202200009	ALD3834	16
202200014	GQY6753	16
202200008	IAC8974	16
202200013	IAC8974	16
202200006	JIE0952	16
202200016	JIE0952	16

```
SELECT cod_apolice, placa,  
       COUNT(placa) OVER(PARTITION BY placa) as qtde  
FROM apolice  
;
```

cod_apolice	placa	qtde
202200011	ALD3834	3
202200015	ALD3834	3
202200009	ALD3834	3
202200014	GQY6753	1
202200008	IAC8974	2
202200013	IAC8974	2
202200006	JIE0952	3
202200016	JIE0952	3

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

### EXEMPLOS DE FUNÇÕES CLASSIFICAÇÃO

```
SELECT cod_apolice, placa,  
       ROW_NUMBER() OVER(ORDER BY placa) as qtde  
FROM apolice  
order by placa  
;
```

cod_apolice	placa	qtde
202200011	ALD3834	1
202200015	ALD3834	2
202200009	ALD3834	3
202200014	GQY6753	4
202200008	IAC8974	5
202200013	IAC8974	6
202200006	JIE0952	7

```
SELECT cod_apolice, placa,  
       ROW_NUMBER() OVER(PARTITION BY placa ORDER BY placa) as qtde  
FROM apolice  
;
```

cod_apolice	placa	qtde
202200011	ALD3834	1
202200015	ALD3834	2
202200009	ALD3834	3
202200014	GQY6753	1
202200008	IAC8974	1
202200013	IAC8974	2
202200006	JIE0952	1

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

### EXEMPLOS DE FUNÇÕES CLASSIFICAÇÃO

```
SELECT cod_apolice, placa,  
       RANK() OVER(ORDER BY placa) as qtde  
FROM apolice  
;
```

cod_apolice	placa	qtde
202200011	ALD3834	1
202200015	ALD3834	1
202200009	ALD3834	1
202200014	GQY6753	4
202200008	IAC8974	5
202200013	IAC8974	5
202200006	JIE0952	7
-----		

```
SELECT cod_apolice, placa,  
       DENSE_RANK() OVER(ORDER BY placa) as qtde  
FROM apolice  
;
```

cod_apolice	placa	qtde
202200011	ALD3834	1
202200015	ALD3834	1
202200009	ALD3834	1
202200014	GQY6753	2
202200008	IAC8974	3
202200013	IAC8974	3
202200006	JIE0952	4

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

### EXEMPLOS DE FUNÇÕES POSICIONAMENTO

```
SELECT cod_apolice, placa,  
       FIRST_VALUE(cod_apolice)  
    OVER(partition BY placa  
        ORDER BY placa, cod_apolice ASC) as qtde  
FROM apolice  
;
```

cod_apolice	placa	qtde
202200009	ALD3834	202200009
202200011	ALD3834	202200009
202200015	ALD3834	202200009
202200014	GQY6753	202200014
202200008	IAC8974	202200008
202200013	IAC8974	202200008
202200003	JIE0952	202200003
202200006	JIE0952	202200003
202200016	JIE0952	202200003

```
SELECT cod_apolice, placa,  
       LAST_VALUE(cod_apolice)  
    OVER(PARTITION BY placa ORDER BY placa, cod_apolice ASC  
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as qtde  
FROM apolice  
;
```

cod_apolice	placa	qtde
202200009	ALD3834	202200015
202200011	ALD3834	202200015
202200015	ALD3834	202200015
202200014	GQY6753	202200014
202200008	IAC8974	202200013
202200013	IAC8974	202200013
202200003	JIE0952	202200016
202200006	JIE0952	202200016
202200016	JIE0952	202200016

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

**ROWS BETWEEN** *<inicio\_da\_linha>* **AND** *<fim\_da\_linha>*

Nos itens *<inicio\_da\_linha>* e *<fim\_da\_linha>*, temos as seguintes opções:

**UNBOUNDED PRECEDING** — todas as linhas antes da linha atual na partição, ou seja, a primeira linha da partição.

**[#] PRECEDING** — # número de linhas antes da linha atual

**CURRENT ROW** — a linha atual

**[#] FOLLOWING** — # número de linhas depois da linha atual

**UNBOUNDED FOLLOWING** — todas as linhas depois da linha atual, ou seja, a última linha da partição.

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

Alguns exemplos de como ler **ROWS BETWEEN**:

**ROWS BETWEEN 3 PRECEDING AND CURRENT ROW** — isso significa olhar para trás nas 3 linhas anteriores até a linha atual.

**ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING** — isso significa olhar da primeira linha da partição para 1 linha após a linha atual

**ROWS BETWEEN 5 PRECEDING AND 1 PRECEDING** — isso significa retroceder nas 5 linhas anteriores até 1 linha antes da linha atual

**ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING** — isso significa olhar da primeira linha da partição até a última linha da partição

**IMPORTANTE:** Sempre que você adiciona uma cláusula **ORDER BY**, o SQL define a janela padrão como **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**.

# Select

## FUNÇÕES DE JANELA (WINDOW FUNCTIONS)

### SINTAXE SIMPLIFICADA

```
SELECT <coluna1>, <coluna2>, <coluna3>, *  
      <WINDOW FUNCTION> OVER ( PARTITION BY <colunas> ORDER BY <colunas> ROWS  
      BETWEEN <opcao_rows_between> AND <opcao_rows_between>)  
FROM <tabela>
```

# Select

## EXERCICIOS

Resolver a lista de exercícios do arquivo:

**“Exercícios Linguagem SQL 2”**

Utilizando os itens vistos em aula (quando possível):

**JOINS, SUBCONSULTA ANINHADA / CORRELACIONADA, CTE e WINDOW FUNCTIONS.**

Procure gerar mais uma versão da mesma consulta utilizando os tópicos acima citados.

# Triggers

**TRIGGER ou GATILHO** são um tipo especial de procedimento armazenado executado automaticamente quando um evento ocorre no servidor de banco de dados.

Existem 3 grupos de triggers:

## Escopo de estudo

- **Triggers de DML** – Triggers que reagem a comandos de DML como: INSERT, UPDATE e DELETE.
- **Triggers de DDL** – Triggers que reagem a comandos de DDL como: CREATE, ALTER e DROP.
- **Triggers de Logon** – Triggers que reagem a eventos de logon.

# Triggers

**TRIGGER DE DML** são as triggers mais utilizadas em um banco de dados relacional.

No SQL Server existem 2 tipos principais de triggers de DML:

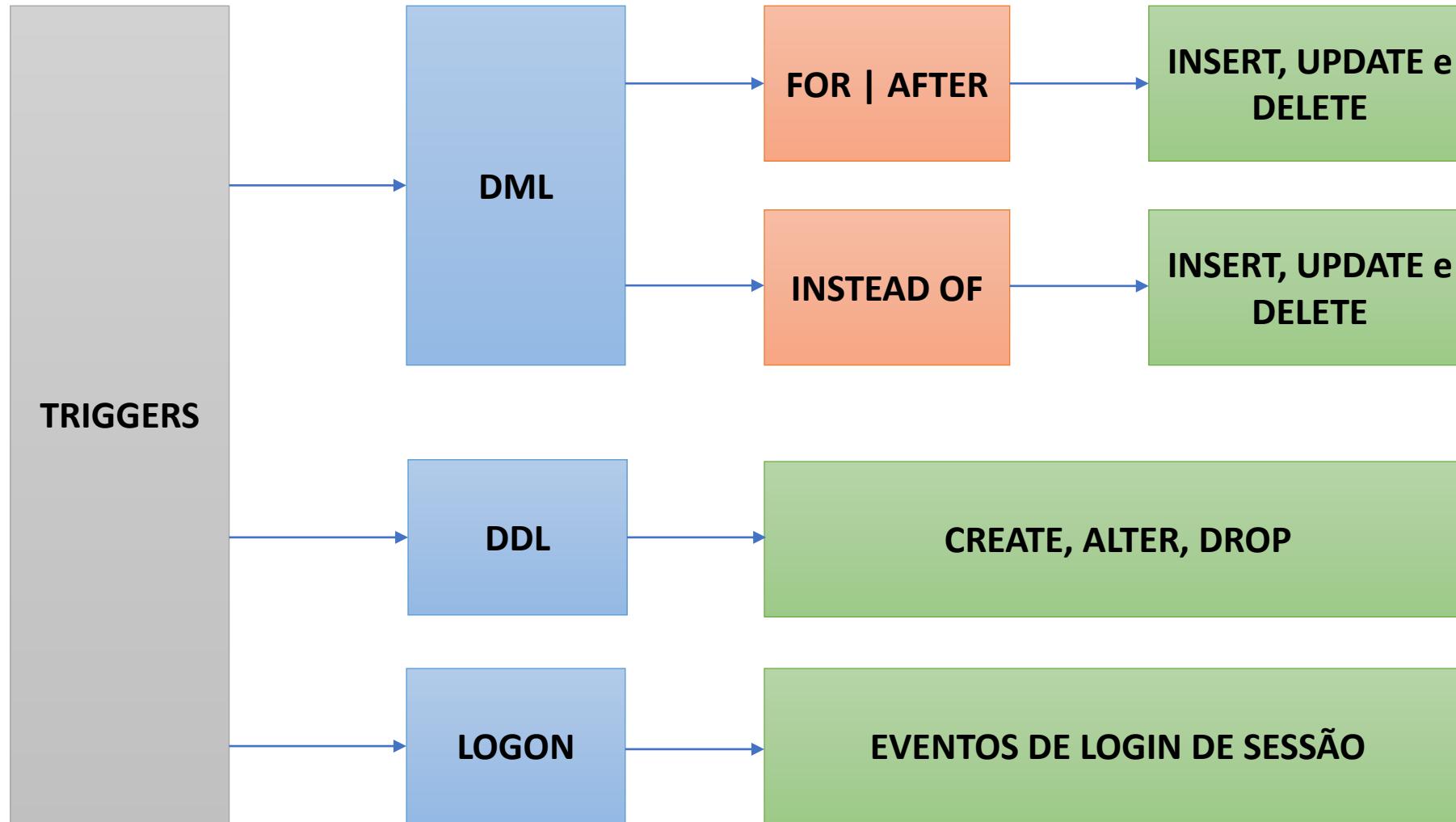
## FOR | AFTER

Especifica que a trigger de DML é disparada apenas quando todas as operações especificadas na instrução SQL de gatilho foram iniciadas com êxito.

## INSTEAD OF

Especifica que a trigger de DML será iniciada *em vez* da instrução SQL de gatilho, substituindo as ações das instruções de gatilho. Não é possível especificar INSTEAD OF para gatilhos DDL ou de logon.

# Triggers



# Triggers

## CRIANDO UMA TRIGGER

### SINTAXE

```
create trigger trigger_name on table_name
for {insert | update | delete} [, {insert | update | delete}]
...
as
sql_statements
```

### EXEMPLO

```
create trigger ti_sales on sales for insert as
begin
    if datename (dw, getdate()) = 'Terça-Feira'
    begin
        raiserror ('Vendas não podem ser feitas na terça.', 16,1)
        rollback tran
        return
    end
end
```

AFTER INSERT  
INSTEAD OF INSERT

# Triggers

## EXCLUINDO UMA TRIGGER

### SINTAXE SIMPLIFICADA

```
DROP TRIGGER trigger_name
```

### EXEMPLO

```
DROP TRIGGER ti_sales
```

### VALIDANDO A EXISTÊNCIA DA TRIGGER ANTES DE EXCLUIR

```
DROP TRIGGER IF EXISTS ti_sales  
go
```

# Triggers

## DESABILITANDO E HABILITANDO UMA TRIGGER

### SINTAXE

```
alter table table_name
{enable | disable} trigger trigger_name
```

OU

```
{enable | disable} trigger trigger_name on table_name
```

### EXEMPLO

```
ALTER TABLE sales DISABLE TRIGGER ti_sales
DISABLE TRIGGER ti_sales ON sales
```

# Triggers

## PROCEDURES DE SISTEMAS PARA TRIGGERS

**sp\_depends {table\_name | trigger\_name}**

- Quando informado uma tabela, ele lista todos os objetos (incluindo triggers) no mesmo banco de dados que fazem referência a tabela
- Quando informada uma trigger, lista todas as tabelas no mesmo banco de dados referenciadas pela trigger

**sp\_help trigger\_name**

- Mostra informações sobre uma determinada trigger

**sp\_helptext triggers\_name**

- Mostra o texto (código) usado para criar a trigger

**sp\_rename old\_triggers\_name, new\_trigger\_name**

- Renomeia o nome da stored trigger

# Triggers

## TABELAS INSERTED E DELETED

**INSERTED** e **DELETED** são duas tabelas virtuais automaticamente criadas, sempre que uma trigger é disparada, dentro do escopo de execução da trigger.

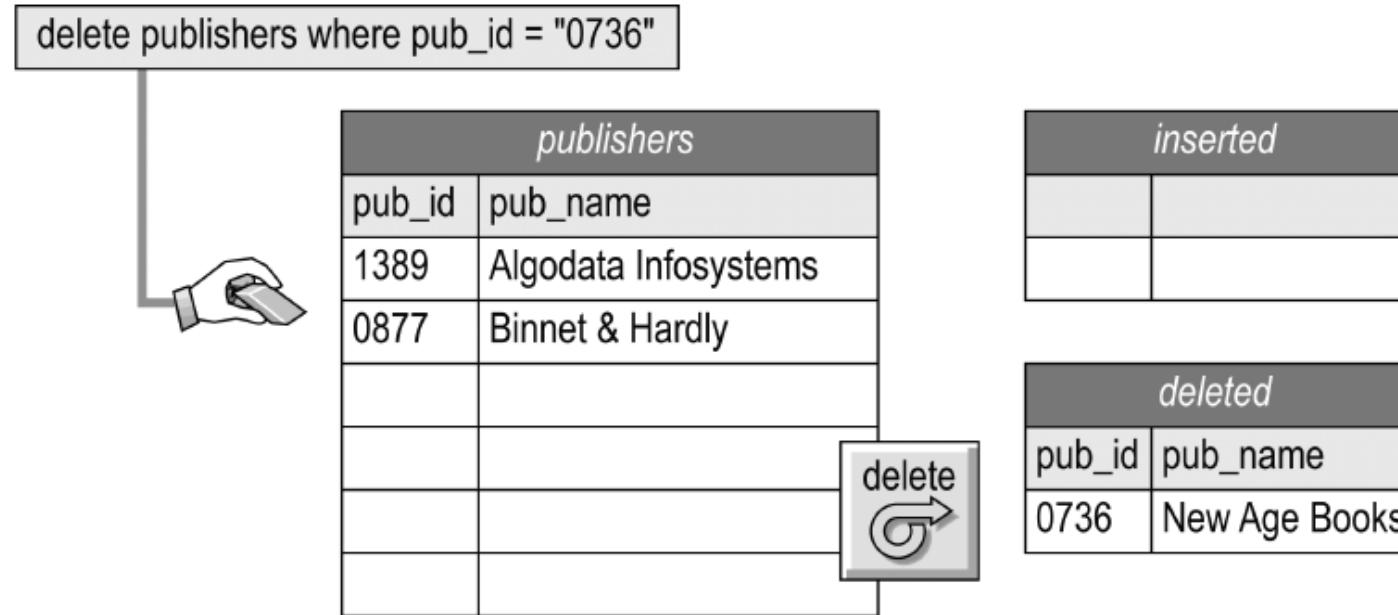
- *inserted* guarda os registros adicionados a tabela.
- *deleted* guarda os registros que foram removidos da tabela.

Evento DML	tabela <b>INSERTED</b> contém	tabela <b>DELETED</b> contém
INSERT	Linhas a serem inseridas	vazio
UPDATE	novas linhas modificadas pelo UPDATE	linhas existentes modificadas pelo UPDATE
DELETE	vazio	Linhas a serem excluídas

# Triggers

## DELETE

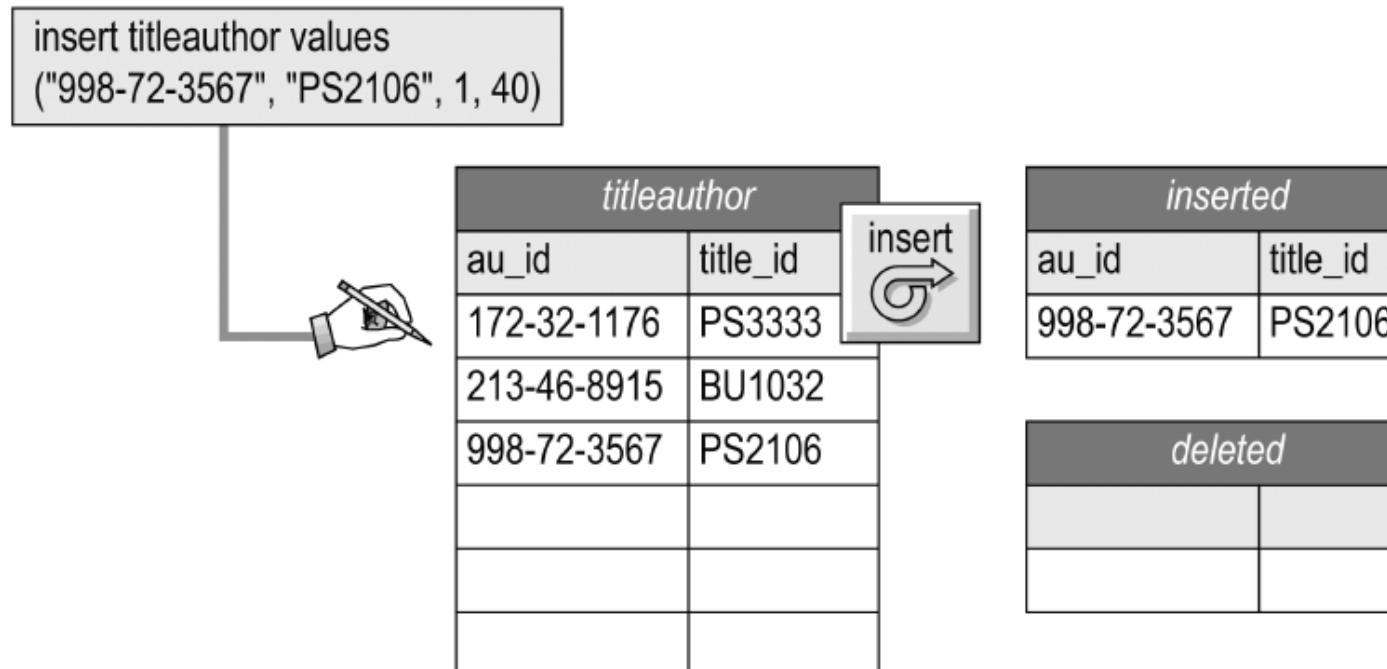
Um comando de *delete* adiciona linhas a tabela *deleted*.



# Triggers

## INSERT

Um comando de *insert* adiciona linhas a tabela *inserted*.



# Triggers

## UPDATE

Um comando de *update* adiciona linhas as tabelas *inserted* e *deleted*.

```
update publishers set pub_id = "9988"  
from publishers where pub_id = "9999"
```



publishers	
pub_id	pub_name
1389	Algodata Infosystems
0877	Binnet & Hardly
9988	Tech Books



inserted	
pub_id	pub_name
9988	Tech Books

deleted	
pub_id	pub_name
9999	Tech Books

# Triggers

## CONSIDERAÇÕES TABELAS INSERTED E DELETED

- Ambas as tabelas *inserted* e *deleted* possuem as mesmas colunas da tabela da trigger.
- A trigger pode selecionar os dados de ambas as tabelas (outros processos não podem).
- As triggers não podem modificar os dados das tabelas *inserted* e *delete* (somente leitura).

# Triggers

## MELHORES PRÁTICAS

Os itens a seguir devem ser levados em consideração para um eficaz desenvolvimento de triggers:

- *@@ROWCOUNT ou ROWCOUNT\_BIG() no começo do código.*
- **IF UPDATE**
- RAISERROR ou THROW

# Triggers

## MELHORES PRÁTICAS

```
DROP TRIGGER IF EXISTS Person.reminder;
GO

CREATE TRIGGER tu_reminder ON Person.Address AFTER UPDATE AS
BEGIN
    IF (ROWCOUNT_BIG() = 0)
        RETURN;
    IF ( UPDATE (StateProvinceID) OR UPDATE (PostalCode) )
    BEGIN
        RAISERROR ('Notify Customer Relations', 16, 1);
        --OU
        THROW 50000,N'Notify Customer Relations',1
    END;
END
GO
```

# Triggers

## MELHORES PRÁTICAS - IF UPDATE

- **IF UPDATE** é uma condição que permite uma trigger checar se há uma alteração numa coluna específica (somente INSERT ou UPDATE).
- Ele só pode ser usado com trigger
- Normalmente usado para verificar se os valores em uma coluna de chave primária foram alterados

### SINTAXE

```
if update (column_name) [ {and | or} update (column_name) ] ...
```

# Triggers

## MELHORES PRÁTICAS – RAISERROR E THROW

**RAISERROR** e **THROW** gera uma exceção no código que pode ser capturado pela aplicação para tratamento de erros (transfere a execução para um bloco CATCH de uma construção TRY CATCH)

```
THROW 50000, N'An error occurred', 1;
```

```
Msg 50000, Level 16, State 1, Line 11  
An error occurred
```

```
RAISERROR (N'An error occurred', 16, 1)
```

```
Msg 50000, Level 16, State 1, Line 13  
An error occurred
```

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios gatilhos (triggers)”**

Utilizando os itens vistos em aula:

**PROGRAMAÇÃO T-SQL, STORED PROCEDURE, CURSORES e TRIGGERS.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.



**SATC**  
EDUCAÇÃO E TECNOLOGIA

## **BANCO DE DADOS**

Engenharia de Software – 3<sup>a</sup> fase

Prof. Jorge Luiz da Silva

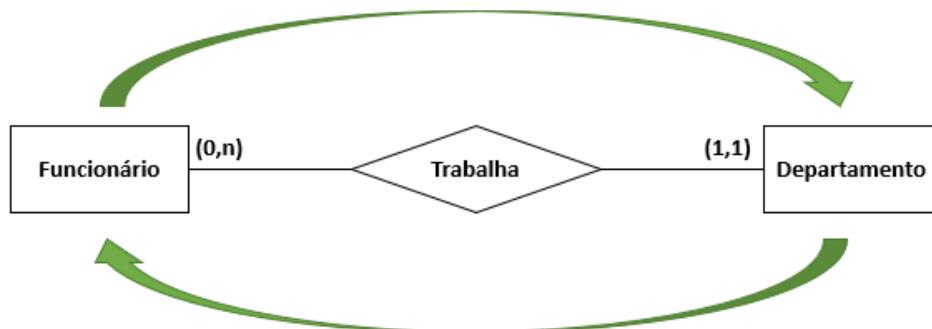
## Categorias

**DDL** : Linguagem de Definição de Dados  
**DQL** : Linguagem de Consulta de Dados  
**DML** : Linguagem de Manipulação de Dados  
**DCL** : Linguagem de Controle de Dados  
**DTL** : Linguagem de Transação de Dados

## Comandos

**DDL**  
CREATE | DROP | ALTER | RENAME  
**DQL**  
SELECT  
**DML**  
INSERT | UPDATE | DELETE  
**DCL**  
GRANT | REVOKE  
**DTL**  
COMMIT | ROLLBACK | BEGIN TRAN

## Modelo ER Conceitual



## Operadores

**Aritméticos** bit a bit  
+ - \* / % & | ^

**Comparação**  
= < > <= >= <> != !> !<

**Lógicos**  
AND | OR | NOT | IN | LIKE | ALL  
SOME | ANY | EXISTS | BETWEEN

**Compostos**  
+= -= \*= /= %= &= ^= |=

## Palavras-chave

WHERE | DISTINCT | TOP | FROM  
AS | ORDER BY | ASC | DESC | CASE  
DEFAULT | VALUES | SET

## Objetos

TABLE | VIEW | PROCEDURE | INDEX  
TRIGGER | SEQUENCE | FUNCTION

## Constraints

NOT NULL | UNIQUE | PRIMARY KEY  
FOREIGN KEY | CHECK | DEFAULT

## Funções de Agregação

AVG | MIN | MAX | COUNT | SUM

## Palavras-chave Agregação

GROUP BY | HAVING

## Joins

INNER JOIN



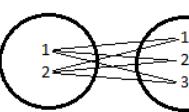
LEFT JOIN



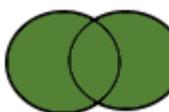
RIGHT JOIN



CROSS JOIN

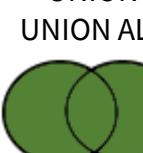


FULL JOIN

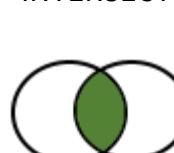


## Operações de Conjunto

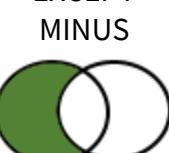
UNION



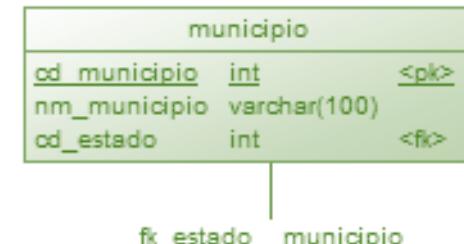
INTERSECT



EXCEPT



## Modelo ER Físico



## Exemplos DDL

### Cria uma tabela

```
CREATE TABLE aluno (
    cd_aluno int NOT NULL,
    nm_aluno varchar(100),
    idade int NOT NULL
);
```

### Adiciona uma coluna

```
ALTER TABLE aluno
ADD is_ativo bit;
```

### Modifica uma coluna

```
ALTER TABLE aluno
ALTER COLUMN idade tinyint;
```

### Remove uma coluna

```
ALTER TABLE aluno
DROP COLUMN idade;
```

### Remove uma tabela

```
DROP TABLE aluno;
```

## Exemplos DML

### Insere dados na tabela

```
INSERT INTO aluno (cd_aluno, nm_aluno, idade)
VALUES (1, 'Fulano de Tal', 23);
```

### Insere dados a partir de uma consulta

```
INSERT INTO aluno (cd_aluno, nm_aluno, idade)
SELECT codigo, nome, idade FROM colaborador;
```

### Atualiza dados de uma tabela

```
UPDATE aluno SET idade = 36 WHERE cd_aluno = 123;
```

### Apaga alguns dados de uma tabela

```
DELETE FROM aluno WHERE is_ativo = 0;
```

### Apaga todos dados de uma tabela

```
DELETE FROM aluno;
* TRUNCATE TABLE aluno;
```

### Insere dados na tabela

```
INSERT INTO aluno (cd_aluno, nm_aluno, idade)
VALUES (1, 'Fulano de Tal', 23);
```

## Exemplos DQL

### Lista todos as linhas e colunas

```
SELECT * FROM aluno;
```

### Filtre linhas de um tabela

```
SELECT * FROM aluno
WHERE cd_aluno = 123;
```

```
SELECT * FROM aluno
WHERE cd_aluno = 123
AND is_ativo = 1;
```

### Filtre colunas de uma tabela

```
SELECT cd_aluno, nm_aluno
FROM aluno
WHERE cd_aluno = 123
AND is_ativo = 1;
```

### Lista máximo de 10 linhas

```
SELECT TOP 10 cd_aluno, nm_aluno
FROM aluno
WHERE is_ativo = 1;
```

### Conta quantidade de linhas

```
SELECT COUNT(*) FROM aluno;
```

### Valores max e min de uma coluna

```
SELECT MAX(idade), MIN(idade) FROM aluno;
```

### Soma valores de uma coluna

```
SELECT SUM(idade) FROM aluno;
```

### Média dos valores de uma coluna

```
SELECT AVG(idade) FROM aluno;
```

### Ordena resultado (ascendente)

```
SELECT * FROM aluno
ORDER BY nm_aluno ASC;
```

### Ordena resultado (descendente)

```
SELECT * FROM aluno
ORDER BY nm_aluno DESC;
```

### Busca linhas e colunas de 2 tabelas

```
SELECT * FROM aluno INNER JOIN aula
ON aluno.cd_aluno = aula.cd_aluno
WHERE is_ativo = 1;
```

## Exemplos de Constraints

### Remove uma Constraint

```
ALTER TABLE aluno DROP CONSTRAINT pk_aluno;
```

### Cria uma Check Constraint

```
ALTER TABLE aluno ADD CONSTRAINT chk_1 CHECK (is_ativo IN (0, 1));
```

### Cria um Default Constraint

```
ALTER TABLE aluno ADD CONSTRAINT df_ativo DEFAULT 1 FOR is_ativo;
```

### Agregação e filtro na agregação

```
SELECT turma, COUNT(1) as quantidade
FROM turma
WHERE curso LIKE 'Engenharia%'
GROUP BY turma
HAVING quantidade > 40;
```

# Select

## JUNÇÕES (JOINS)

**JUNÇÕES** são utilizados quando precisamos realizar uma consulta utilizando mais de uma tabela, ligando as tabelas através de campos chaves.

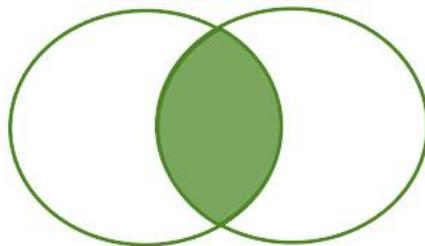
Basicamente temos 5 tipos de JOIN no SQL Server (podendo mudar de um SGBD para outro):

- INNER JOIN
- LEFT [ OUTER ] JOIN
- RIGHT [ OUTER ] JOIN
- FULL [ OUTER ] JOIN
- CROSS JOIN

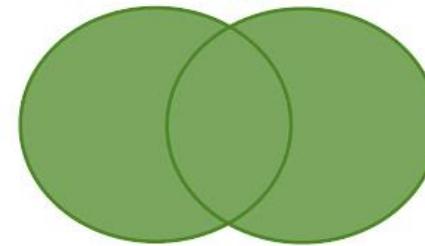
# Select

## JUNÇÕES (JOINS)

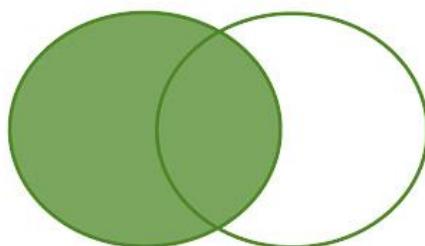
Inner Join



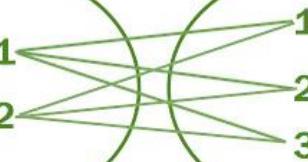
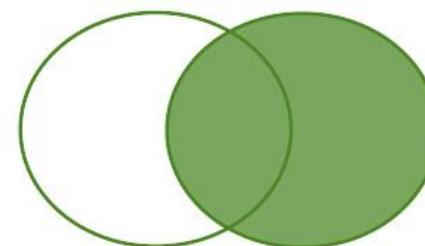
Full Join



Left Join



Right Join

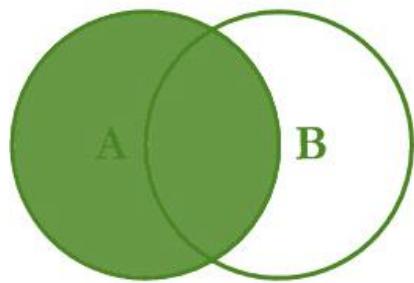


Cross Join

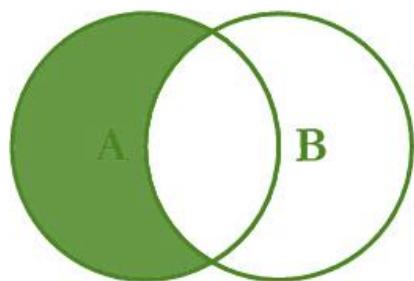
# Select

## JUNÇÕES (JOINS)

### SQL JOINS



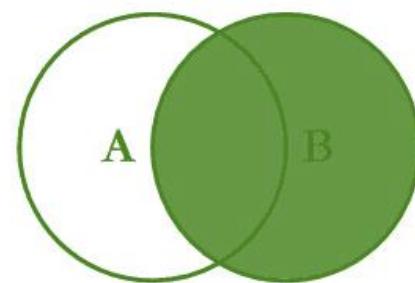
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



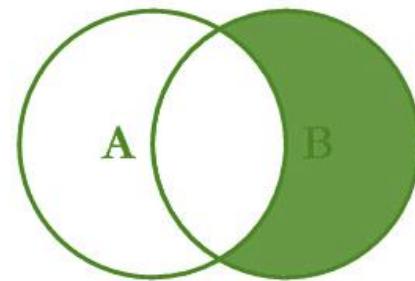
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



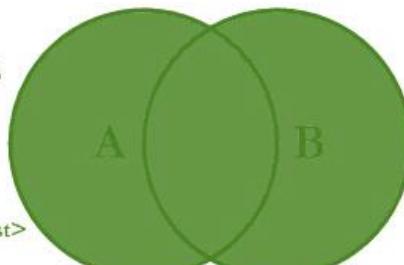
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
WHERE A.Key IS NULL
OR B.Key IS NULL
```

# Select

## JUNÇÕES (JOINS)

### SINTAXE SIMPLIFICADA

```
SELECT column_list  
FROM table1 INNER JOIN table2  
ON table1.column_from_table1 = table2.column_from_table2  
go
```

```
SELECT column_list  
FROM table1 LEFT JOIN table2  
ON table1.column_from_table1 = table2.column_from_table2  
go
```

# Select

## JUNÇÕES (JOINS)

### EXEMPLO

```
SELECT *
FROM apolice INNER JOIN cliente
ON apolice.cod_cliente = cliente.cod_cliente
go
```

```
SELECT *
FROM apolice LEFT JOIN cliente
ON apolice.cod_cliente = cliente.cod_cliente
go
```

# Select

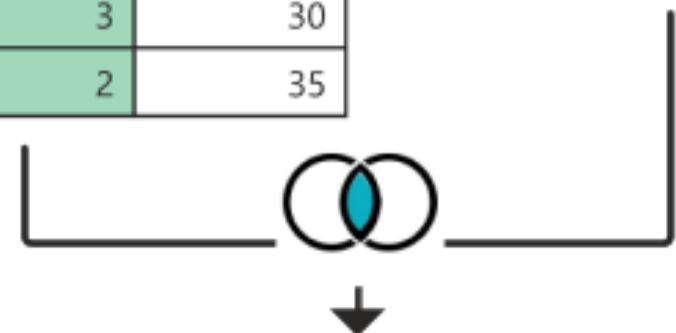
## JUNÇÕES (INNER JOIN)

Left Table

Date	CountryID	Units
1/1/2020	1	40
1/2/2020	1	25
1/3/2020	3	30
1/4/2020	2	35

Right Table

ID	Country
3	Panama
4	Spain



Merged Table

Date	CountryID	Units	Country
1/3/2020	3	30	Panama

-- selecionando todas as colunas da tabela1  
SELECT \* FROM sale

-- selecionando todas as colunas da tabela2  
SELECT \* FROM country

-- junção interna tabela1 e tabela2  
SELECT Date, CountryID, Units, Country  
FROM sale  
INNER JOIN country  
ON sale.CountryID = country.ID;

# Select

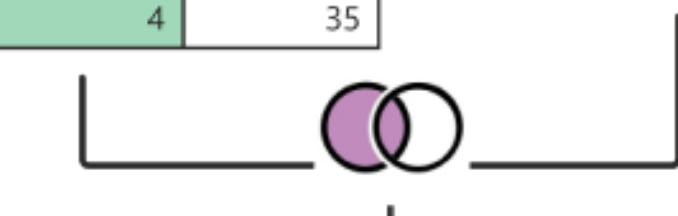
## JUNÇÕES (LEFT JOIN)

Left Table

Date	CountryID	Units
1/1/2020	1	40
1/2/2020	1	25
1/3/2020	3	30
1/4/2020	4	35

Right Table

ID	Country
1	USA
2	Canada
3	Panama



Merged Table

Date	CountryID	Units	Country
1/1/2020	1	40	USA
1/2/2020	1	25	USA
1/3/2020	3	30	Panama
1/4/2020	4	35	null

--selecionando todas as colunas da tabela1  
SELECT \* FROM sale

--selecionando todas as colunas da tabela2  
SELECT \* FROM country

--junção externa esquerda entre tabela1 e tabela2  
SELECT Date, CountryID, Units, Country  
FROM sale  
LEFT JOIN country  
ON sale.CountryID = country.ID;

# Select

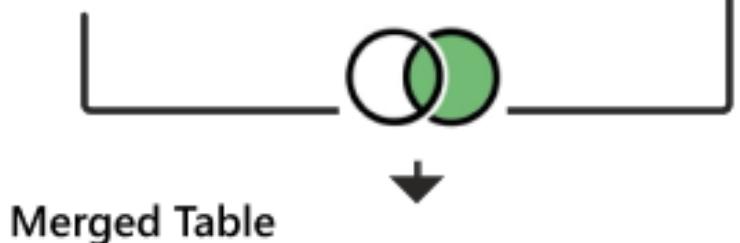
## JUNÇÕES (RIGHT JOIN)

Left Table

Date	CountryID	Units
1/1/2020	1	40
1/2/2020	1	25
1/3/2020	3	30
1/4/2020	4	35

Right Table

ID	Country
3	Panama



Merged Table

Date	CountryID	Units	Country
1/3/2020	3	30	Panama

--selecionando todas as colunas da tabela1  
SELECT \* FROM sale

--selecionando todas as colunas da tabela2  
SELECT \* FROM country

--junção externa esquerda entre tabela1 e tabela2  
SELECT Date, CountryID, Units, Country  
FROM sale  
RIGHT JOIN country  
ON sale.CountryID = country.ID;

# Select

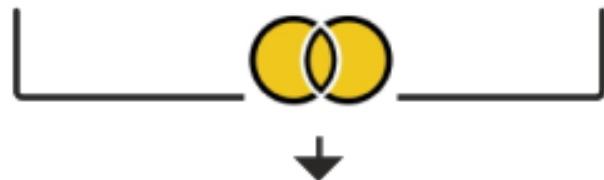
## JUNÇÕES (FULL JOIN)

Left Table

Date	CountryID	Units
1/1/2020	1	40
1/2/2020	1	25
1/3/2020	3	30
1/4/2020	2	35

Right Table

ID	Country
1	USA
2	Canada
3	Panama
4	Spain



Merged Table

Date	CountryID	Units	Country
1/1/2020	1	40	USA
1/2/2020	1	25	USA
1/4/2020	2	35	Canada
1/3/2020	3	30	Panama
null	null	null	Spain

--selecionando todas as colunas da tabela1  
SELECT \* FROM sale

--selecionando todas as colunas da tabela2  
SELECT \* FROM country

--junção externa esquerda entre tabela1 e tabela2  
SELECT Date, CountryID, Units, Country  
FROM sale  
FULL JOIN country  
ON sale.CountryID = country.ID;

# Select

## JUNÇÕES (JOINS)

### OBSERVAÇÕES

- Nomes de colunas de ligação do **JOIN** não precisam ser iguais, precisam ser parte de uma FK (recomendado).
- **JOINS** entre mais de duas tabelas devem listar todas tabelas envolvidas na cláusula FROM, com seus devidos conectivos de relacionamento **ON**, mesmo que não estejam visíveis na lista do SELECT.

# Select

## SQL ALIAS

### SINTAXE SIMPLIFICADA

```
SELECT column_list  
FROM real_table_name alias_name
```

ALIAS é um apelido curto que damos para as tabelas para facilitar a escrita, evitar repetição do nome da tabela na consulta.

### EXEMPLO

```
SELECT ld.dt_lancamento, tl.sc_lancamento, ld.vl_lancamento  
FROM lancamento_diario ld  
INNER JOIN tipo_lancamento tl  
ON ld.cd_tp_lancamento = tl.cd_tp_lancamento  
go
```

# Select

## SUBCONSULTAS (SUBQUERIES)

**SUBCONSULTAS** são utilizados quando queremos incluir uma consulta dentro de outra consulta.

Basicamente temos em 2 tipos principais de subconsultas:

- Subconsulta aninhada
- Subconsulta correlacionada

### Consulta externa

```
SELECT cd_conta  
FROM pessoa_conta  
WHERE cd_pessoa IN (SELECT cd_pessoa Consulta interna  
                      FROM PESSOA  
                      WHERE nm_pessoa LIKE 'JOAO%')
```

# Select

## SUBCONSULTA ANINHADA

**SUBCONSULTAS ANINHADAS** são consultas auxiliares/secundárias que fornecem dados para a consulta principal.

Uma subconsulta pode incluir uma ou mais subconsultas.

```
SELECT StateProvinceID, AddressID  
FROM Person.Address  
WHERE AddressID IN  
(SELECT AddressID  
FROM Person.Address  
WHERE StateProvinceID = 39);  
GO
```

A mesma consulta usando join

```
SELECT e1.StateProvinceID, e1.AddressID  
FROM Person.Address AS e1  
INNER JOIN Person.Address AS e2  
ON e1.AddressID = e2.AddressID  
AND e2.StateProvinceID = 39;  
GO
```

# Select

## SUBCONSULTA CORRELACIONADA

**SUBCONSULTAS CORRELACIONADAS**, também conhecida como uma subconsulta repetitiva, a subconsulta correlacionada depende da consulta externa para obter seus valores.

Isso significa que a subconsulta é executada repetidamente, uma vez para cada linha que pode ser selecionada pela consulta externa.

--Liste todas as cidades cujo ranking é MAIOR o ranking médio de todas as cidades do mesmo país (planilha).

```
SELECT *
FROM city main_city
WHERE rating > (
    SELECT AVG(rating)
    FROM city average_city
    WHERE average_city.country_id = main_city.country_id
);
```

Planilha

# Select

## SUBCONSULTA NA LISTA DO SELECT

É possível utilizar subconsulta dentro da *select list*, também conhecidas como **subconsultas de expressão**.

Normalmente é utilizado para trocar um código por um nome ou vice-versa. Pode ser tanto alinhada quando correlacionada.

```
SELECT cod_apolice  
      ,cod_cliente  
      ,placa  
  FROM apolice
```

```
SELECT cod_apolice  
      ,(SELECT nome from cliente where cliente.[cod_cliente] = apolice.cod_cliente)  
      ,placa  
  FROM apolice
```

# Select

## EXISTS E NOT EXISTS

A cláusula **EXISTS** faz uma validação se existe algum resultado para a subconsulta informada. Caso tenha algum resultado (retorno positivo), o resultado da consulta principal é listado.

É bastante comum a utilização de EXISTS e NOT EXISTS com subconsultas correlacionadas e são equivalentes a subconsultas com IN e NOT IN.

```
select * from carro where placa not in (select placa from sinistro)
```

```
select * from carro where not exists (select * from sinistro  
where sinistro.placa = carro.placa)
```

```
SELECT *  
FROM country  
WHERE EXISTS (  
    SELECT *  
    FROM mountain  
    WHERE country_id = country.id  
);
```

Planilha

# Select

## CTE (COMMON TABLE EXPRESSION)

Define um conjunto de resultados temporários que você pode referenciar várias vezes dentro do escopo de uma instrução SQL.

Uma **CTE** é usada principalmente em uma instrução **SELECT**.

```
with carro_cor as (
    select cor, count(*) as qtde from carro group by cor
)
select * from carro_cor where qtde > 2;
```

A mesma consulta com **HAVING**

```
select cor, count(*) as qtde
from carro
group by cor
having count(*) > 2;
```

# Select

## EXERCICIOS

Resolver a lista de exercícios do arquivo:

**“Exercícios Linguagem SQL”**

Utilizando os itens vistos em aula (quando possível):

**JOINS, SUBCONSULTA ANINHADA / CORRELACIONADA, EXISTS e CTE.**

Procure gerar mais uma versão da mesma consulta utilizando os tópicos acima citados.

Uma query criada com subconsulta de IN, crie uma outra com EXISTS, outra com JOIN (INNER ou LEFT) e por último com CTE, onde todas deverão gerar o mesmo resultado.

# Otimização SQL

## REGRA GERAL

“Otimização SQL refere-se ao processo de **melhorar o desempenho de consultas SQL**, tanto em termos de **tempo de resposta** quanto de consumo de **recursos de hardware** (disco, memória e CPU).”

# Otimização SQL

## DICAS

### #1: DESNORMALIZE sempre que tiver oportunidade!

#### Modelo Lógico Normalização

- **NORMALIZAÇÃO** é o processo que visa organizar os dados em um banco de dados.
- Este processo inclui **regras** para criação de tabelas e seus relacionamentos, a fim de **proteger os dados** e tornar o banco de dados mais **flexível**, eliminando a **redundância** e a **dependência inconsistente**.

# Otimização SQL

## DICAS

**#2: EVITE múltiplos JOINS!**

# Otimização SQL

## DICAS

**#3: Crie e use INDICES!**

# Otimização SQL

## DICAS

**#4: Escolha o TIPO DE DADO adequado!**

# Otimização SQL

## DICAS

**#5: DROP INDEX antes de OPERAÇÕES MASSIVAS!**

# Otimização SQL

## DICAS

**#6: Mantenha as suas TRANSAÇÕES PEQUENAS!**

# Otimização SQL

## DICAS

**#7: NUNCA use SELECT \*!**

### #8: Sempre analise PLANO DE EXECUÇÃO + ESTATISTICAS das consultas (IO e TIME)

# Qual o mais rápido...  
Subconsultas ou JOIN?  
EXISTS ou IN?  
NOT EXISTS ou NOT IN?  
JOIN ou IN ?

# Otimização SQL

## DICAS

**#9: REBUILD de índices e ESTATÍSTICAS de tabelas**

# Otimização SQL

## DICAS

### #10: PDCA – Processo de melhoria continua



# Revisão para a Avaliação A1

Junções e subconsultas e CTE

Funções de Janela

Armazenamento de dados, indexação e planos de acesso

Processamento de transações, controle de concorrência

Otimização de consultas e Revisão para a prova

# Stored Procedures

## DEFINIÇÃO

**STORED PROCEDURE** ou **PROCEDIMENTO ARMAZENADO** é um grupo de uma ou mais instruções do Transact-SQL executado de forma única.

- Aceita parâmetros de entrada e saída.
- Encapsulam tarefas repetitivas.
- Reduz o tráfego de rede.
- São capazes de utilizar os comandos como IF e ELSE, WHILE, CASE, tabelas temporárias, cursores, variáveis, dentro outros.
- Permitem chamar outros procedimentos armazenados dentro dele (execução aninhada).
- Permite utilizar os comandos do grupo DML e alguns do DDL.

# Stored Procedures

## VANTAGENS E DESVANTAGENS

### PRINCIPAIS VANTAGENS

Desempenho/Performance,  
Segurança,  
Reutilização de código.

### PRINCIPAL DESVANTAGEM

Dependência da tecnologia do SGBD.

# Stored Procedures

## TIPOS DE STORED PROCEDURES

### DEFINIDAS PELO USUÁRIO

- Procedimentos de usuário que são executados explicitamente

### TRIGGERS

- Procedimentos de usuário que são executados automaticamente quando há alguma modificação na tabela vinculada (disparo por evento)

### SYSTEM PROCEDURES

- Procedimentos de sistema que lêem ou modificam uma ou mais tabelas de sistema

# Stored Procedures

## CRIANDO UMA STORED PROCEDURE

### SINTAXE

```
CREATE {PROC | PROCEDURE} procedure_name AS  
BEGIN  
    statements  
END  
go
```

### EXEMPLO

```
CREATE PROCEDURE pr_atualiza_titulo AS  
BEGIN  
    UPDATE titulo  
        SET preco = preco * 0.95  
        WHERE total < 3000  
END  
go
```

# Stored Procedures

## EXCLUINDO UMA STORED PROCEDURE

### SINTAXE

```
DROP PROCEDURE procedure_name
```

### EXEMPLO

```
DROP PROCEDURE pr_atualiza_titulo
```

### VALIDANDO A EXISTÊNCIA DA PROCEDURE ANTES DE EXCLUIR

```
IF EXISTS(SELECT 1 FROM sys.procedures  
          WHERE name = 'pr_atualiza_titulo')  
    DROP PROCEDURE pr_atualiza_titulo  
  
go
```

Versão 2014 ou inferior

```
DROP PROCEDURE IF EXISTS pr_atualiza_titulo  
go
```

Versão 2016 ou superior

# Stored Procedures

## EXECUTANDO UMA STORED PROCEDURE

### SINTAXE

```
[exec | execute] procedure_name
```

### EXEMPLO

```
execute pr_atualiza_titulo  
go
```

# Stored Procedures

## VARIÁVEIS EM STORED PROCEDURE

- Stored procedures podem criar e usar variáveis locais.
- As variáveis existirão somente durante a execução da stored procedure.
- As variáveis de uma procedure não poderão ser usadas por outros processos concorrente.

### EXEMPLO

```
CREATE PROC pr_soma AS
BEGIN
    DECLARE @valor_1 int, @valor_2 int

    SELECT @valor_1 = 2, @valor_2 = 3
    --SET @valor_1 = 2
    --SET @valor_2 = 3

    select @valor_1 + @valor_2
END
go
```

```
EXEC pr_soma
go

-----
5
(1 row affected)
```

# Stored Procedures

## PARAMETROS DE ENTRADA

### SINTAXE

```
CREATE PROCEDURE procedure_name  
    (@variavel tipo, @variavel tipo, ... ) as  
BEGIN  
    --statements  
END
```

### EXEMPLO

```
CREATE PROC pr_soma (@valor_1 int, @valor_2 int) AS  
BEGIN  
    SELECT @valor_1 = 2, @valor_2 = 3  
    --SET @valor_1 = 2  
    --SET @valor_2 = 5  
  
    SELECT @valor_1 + @valor_2  
END  
go
```

# Stored Procedures

## PARAMETROS DE ENTRADA : POSIÇÃO X OUTRAS VARIÁVEIS

### EXEMPLO

```
create proc myproc (@val1 int, @val2 int) as  
    ...
```

### PARAMETROS PASSADOS PELA POSIÇÃO

```
exec myproc 10, 20
```

### PARAMETROS PASSADOS POR VARIÁVEL

```
declare @valor1 int, @valor2 int  
  
set @valor1 = 2  
set @valor2 = 2  
  
exec myproc @valor1, @valor2
```

# Stored Procedures

## PARAMETROS DE ENTRADA : DEFAULT

Um valor *default* é atribuído ao parâmetro para os casos onde nenhum valor é fornecido ao parâmetro no momento de sua execução.

### EXEMPLO

```
create proc pr_state_authors (@state char(2) = 'CA') as  
select au_lname, au_fname, state  
from authors  
where state = @state
```

```
exec proc_state_authors --Nao foi passado nenhum parametro
```

au_lname	au_fname	state
White	Johnson	CA
Green	Marjorie	CA
...		

# Stored Procedures

## PARAMETROS DE SAIDA (OUTPUT)

### SINTAXE

```
CREATE PROCEDURE procedure_name
    (@variavel tipo, @variavel tipo OUTPUT, ... ) as
BEGIN
    --statements
END
```

### EXEMPLO

```
CREATE PROC pr_new_price (@title_id int,
                        @new_price numeric(14,2) output) as
BEGIN
    SELECT @new_price = price FROM titles
    WHERE title_id = @title_id

    SELECT @new_price = @new_price * 1.15
END
```

# Stored Procedures

## PARAMETROS DE SAIDA (OUTPUT) - EXECUÇÃO

### SINTAXE

```
[exec | execute] procedure_name variable output
```

### EXEMPLO

```
declare @title_id char(6), @new_price numeric(14,2)

select @title_id = 'PC8888'

exec proc_new_price @title_id, @new_price output

select @new_price
go
```

-----

23.00

# Stored Procedures

## STORED PROCEDURES DE SISTEMAS

**sp\_depends {table\_name | procedure\_name}**

- Quando informado uma tabela, ele lista todos os objetos (incluindo procedures) no mesmo banco de dados que fazem referencia a tabela
- Quando informado uma proc, lista todas as tabelas no mesmo banco de dados referenciadas pela proc

**sp\_help procedure\_name**

- Mostra informações sobre uma determinada stored procedure

**sp\_helptext procedure\_name**

- Mostra o texto (código) usado para criar a stored procedure

**sp\_rename old\_proc\_name, new\_proc\_name**

- Renomeia o nome da stored procedure

# Stored Procedures

## LIMITAÇÕES

As instruções a seguir não podem ser usadas em qualquer lugar no corpo de um stored procedure.

CREATE	SET	USE
CREATE AGGREGATE	SET SHOWPLAN_TEXT	USE <i>database_name</i>
CREATE DEFAULT	SET SHOWPLAN_XML	
CREATE RULE	SET PARSEONLY	
CREATE SCHEMA	SET SHOWPLAN_ALL	
CREATE ou ALTER TRIGGER		
CREATE ou ALTER FUNCTION		
CREATE ou ALTER PROCEDURE		
CREATE ou ALTER VIEW		

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios Stored Procedure 1”**

Utilizando os itens vistos em aula:

**PROGRAMAÇÃO T-SQL E STORED PROCEDURE.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

# Processamento de Transações

## DEFINIÇÃO

### O QUE É UMA TRANSAÇÃO?

É o conjunto de uma ou mais operações executadas integralmente para garantir a consistência dos dados de um banco de dados (unidade lógica única).

### EXEMPLO

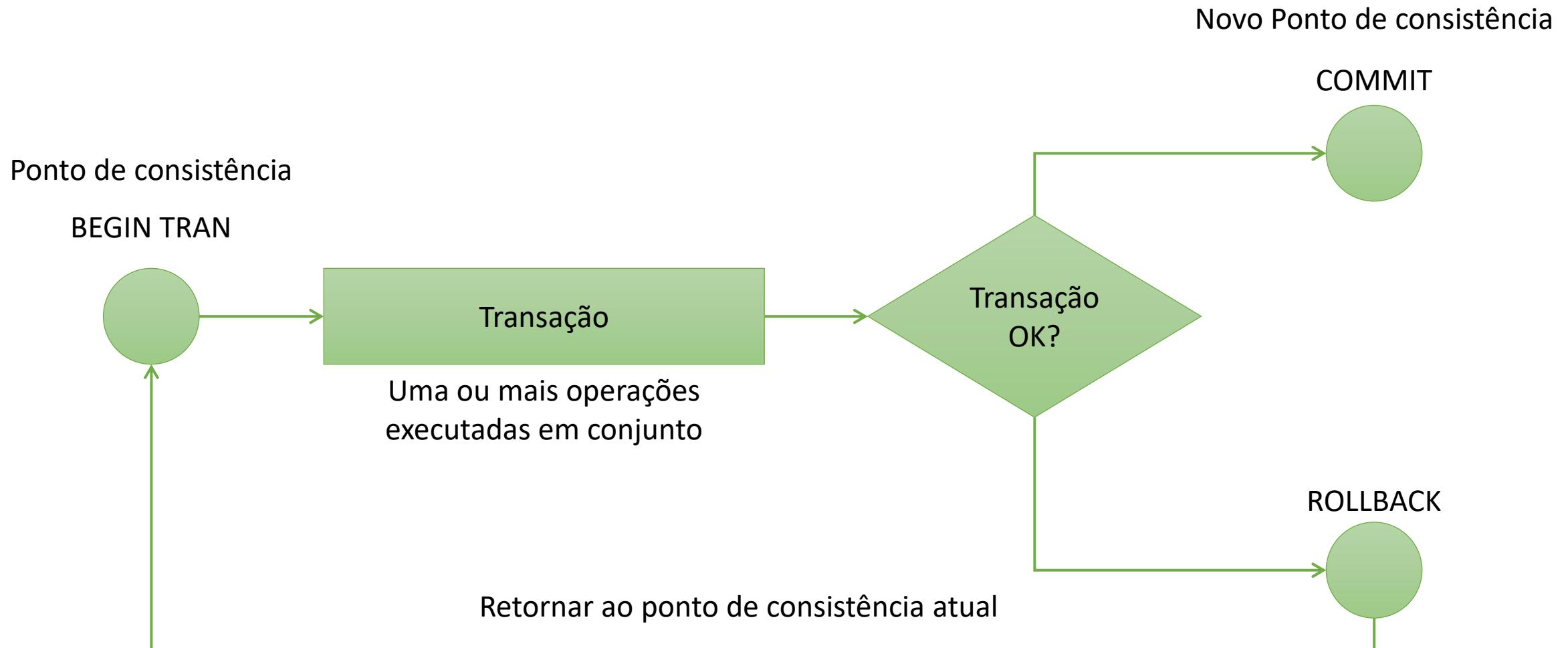
Transferência entre contas bancárias

1. Operação de débito na conta de origem
2. Atualização de saldo na conta de origem
3. Operação de crédito na conta de destino
4. Atualização de saldo na conta de destino

TODAS as operações  
devem ser confirmadas  
ou NENHUMA.

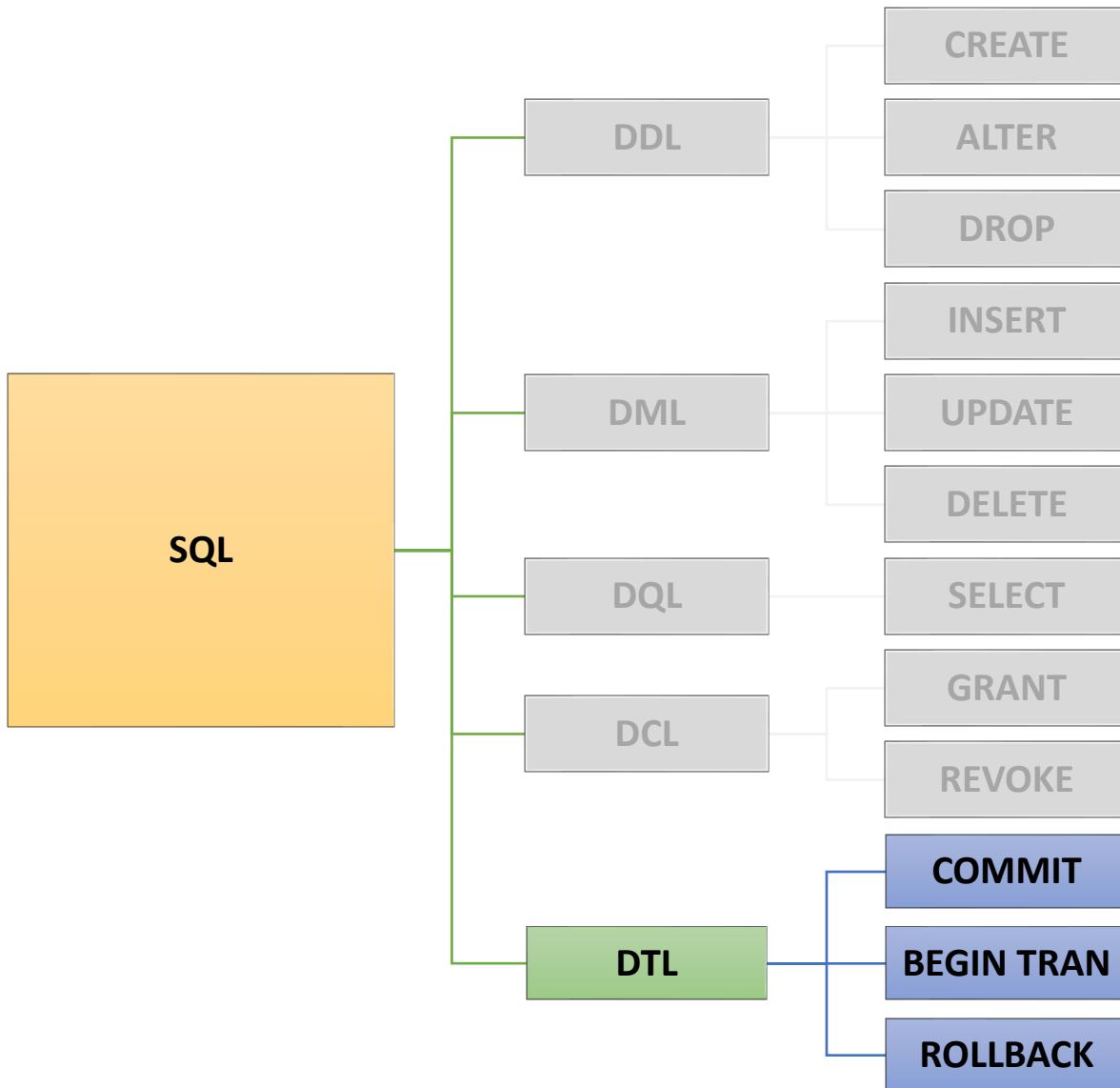
# Processamento de Transações

## FLUXO



# Processamento de Transações

## GRUPOS SQL



# Processamento de Transações

## GRUPOS SQL

**BEGIN TRANSACTION (BEGIN TRAN):** Inicia uma transação de forma explícita, marcando na linha de tempo, um ponto de consistência.

**COMMIT TRANSACTION (COMMIT):** Finaliza a transação confirmando todas as alterações no banco de dados de forma permanente, gerando um novo ponto de consistência.

**ROLLBACK TRANSACTION (ROLLBACK):** Desfaz todas as operações da transação, retornando ao ponto de consistência atual.

# Processamento de Transações

## COMANDOS DTL

### EXEMPLO

```
BEGIN TRAN;

INSERT INTO lancamento VALUES (1, 52620, 'DOC C', '2023-03-01', 100.00, 'D');

UPDATE saldo SET vl_saldo = vl_saldo - 100
WHERE conta = 52620 AND data = '2023-03-01';

INSERT INTO lancamento VALUES (1, 81626, 'DOC C', '2023-03-01', 100.00, 'C');

UPDATE saldo SET vl_saldo = vl_saldo + 100
WHERE conta = 81626 AND data = '2023-03-01';

COMMIT;
OU
ROLLBACK;
```

# Processamento de Transações

## OBSERVAÇÕES

Um **COMMIT** confirma a transação aberta por **BEGIN TRAN** mais atual dentro da mesma sessão. Um **ROLLBACK** desfaz todas as transações abertas.  
É possível ter mais um **BEGIN TRAN** dentro de outro **BEGIN TRAN** (transações aninhadas).

A variável **@@TRANCOUNT** mostra o nível de transações dentro de uma sessão, ou seja, mostra o número de instruções **BEGIN TRAN** que ocorreram na conexão atual.

```
BEGIN TRAN  
SELECT @@TRANCOUNT  
--1  
BEGIN TRAN  
SELECT @@TRANCOUNT  
--2  
BEGIN TRAN  
SELECT @@TRANCOUNT  
--3
```

```
COMMIT;  
SELECT @@TRANCOUNT  
--2  
COMMIT;  
SELECT @@TRANCOUNT  
--1  
COMMIT;  
SELECT @@TRANCOUNT  
--0
```

```
BEGIN TRAN  
SELECT @@TRANCOUNT  
--1  
BEGIN TRAN  
SELECT @@TRANCOUNT  
--2  
BEGIN TRAN  
SELECT @@TRANCOUNT  
--3
```

```
ROLLBACK;  
SELECT @@TRANCOUNT  
--0
```

# Processamento de Transações

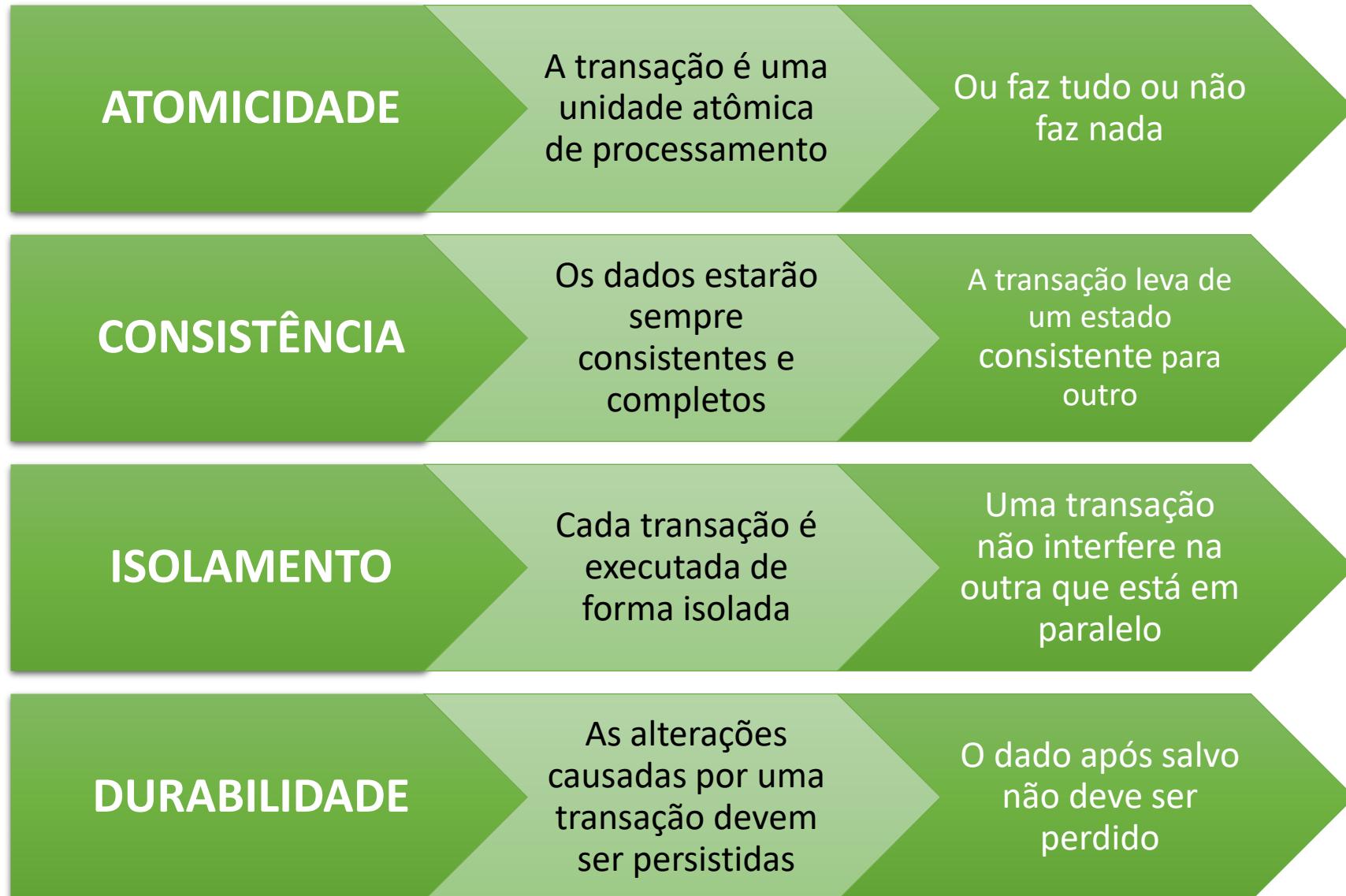
## FALHAS

### TIPOS DE FALHAS

- Falhas de computador, hardware, software ou rede.
- Erro durante execução de operação na transação: estouro de variáveis, condições de verificação não tratadas.
- Condições de exceção detectadas pela transação (necessitam o cancelamento da mesma): saldo Insuficiente em conta
- Falta de energia, ar-condicionado, acidentes.

# Processamento de Transações

## PROPRIEDADES DA TRANSAÇÃO



# Controle de Concorrência

## DEFINIÇÃO

**CONTROLE DE CONCORRÊNCIA** São técnicas utilizadas para garantir que transações concorrentes sejam executadas adequadamente.

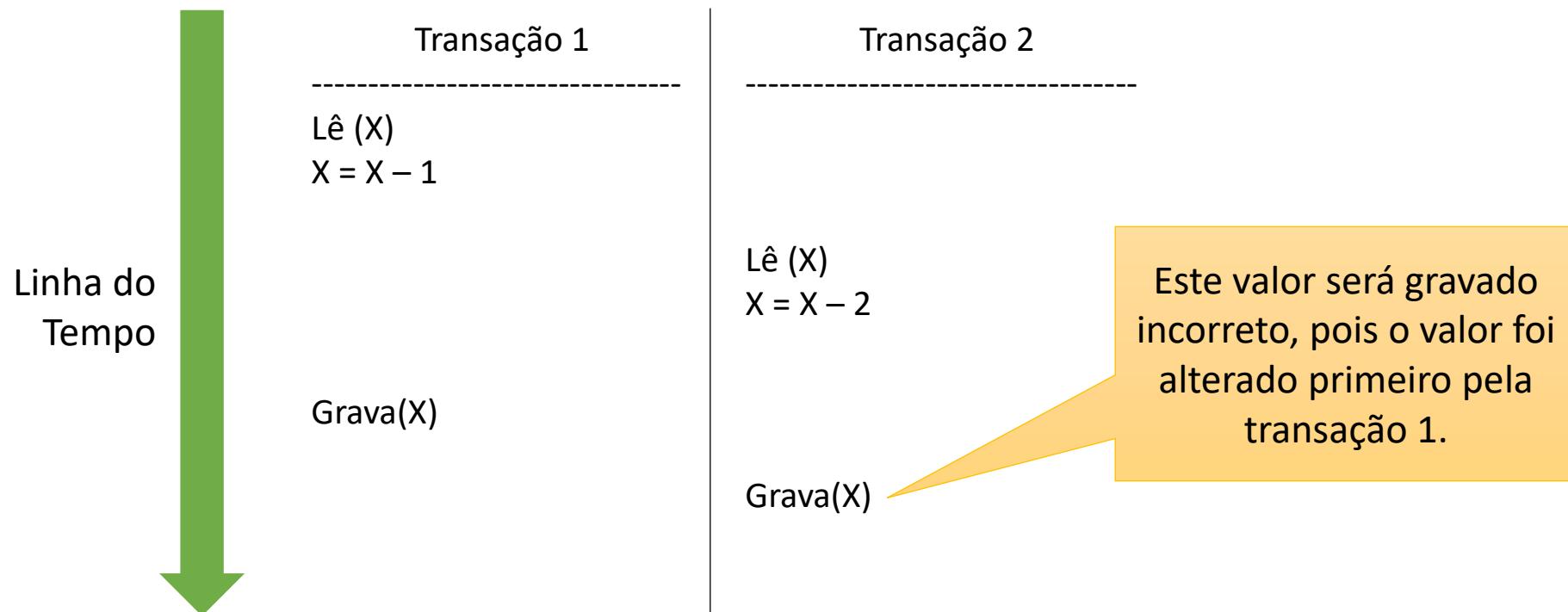
Existem vários problemas que ocorrem quando **não temos** controle de transações concorrentes, as duas principais são:

- Problema da perda de atualização;
- Problema da atualização temporária (leitura suja);

# Controle de Concorrência

## CENÁRIOS DE PRINCIPAIS PROBLEMAS

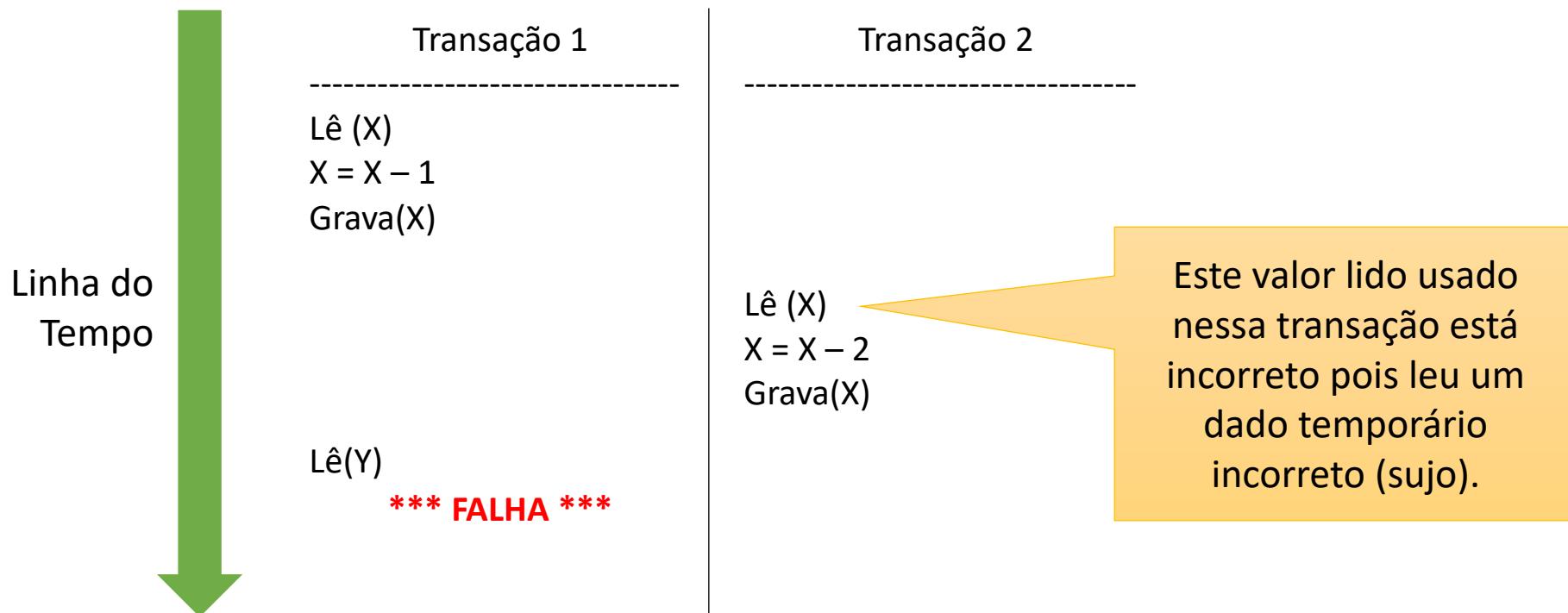
**PROBLEMA DA PERDA DE ATUALIZAÇÃO** ocorre quando duas ou mais transações acessam e alteram os mesmos dados, de modo que torne incorreto o valor de alguma operação no banco de dados.



# Controle de Concorrência

## CENÁRIOS DE PRINCIPAIS PROBLEMAS

**PROBLEMA DA ATUALIZAÇÃO TEMPORÁRIA (LEITURA SUJA)** Ocorre quando uma transação atualiza um item e, por algum motivo, ela falha; neste meio tempo, uma outra transação utiliza o dado atualizado (antes da falha) para seguir suas operações.



# Controle de Concorrência

## NIVEIS DE ISOLAMENTOS

### READ UNCOMMITTED

- Permite a leitura de dados não efetivados.
- Há ganho de performance, mas perda de segurança.

### READ COMMITTED

- Nível de isolamento Padrão.
- Leitura só pode ser feita em dados efetivados.

**NIVEL DE ISOLAMENTO PADRÃO  
SQL SERVER**

### REPEATABLE READ

- Registros lidos não serão alterados por outros processos, garantindo releituras idênticas.

### SERIALIZABLE

- A mais restrita de todas.
- Inserções ou deleções não podem ser feitas em conjuntos de registros lidos.

# Controle de Concorrência

## PARADIGMA DO CONTROLE DE CONCORRÊNCIA

**PARADIGMA:** Impedir que múltiplas transações acessem os itens concorrentemente sem controle.

**“Se uma transação precisa de um dado que está sendo manipulado por outra transação, então ela é forçada a esperar até o dado seja liberado pela primeira transação”**

# Controle de Concorrência

## TERMOS CONCEITUAIS

**LOCK:** Ocorre quando uma sessão (Ex: sessão 121) está realizando alguma alteração em algum objeto e o SQL Server aplica uma trava nesse objeto para impedir outras sessões tentem acessar ou modificar o dado até que seja liberado a trava (commit ou rollback da sessão 121).

**BLOCK:** Cenário parecido com o Lock, mas com a diferença que nesse cenário, existe um lock no objeto e uma ou mais sessões estão tentando ler ou alterar esse objeto, ficando assim, aguardando a liberação do lock.

**DEADLOCK:** Quando dois ou mais processos (sessões) tentam acessar um mesmo objeto, aplicando locks nesse recurso. Sendo assim, esses processos tentam realizar a mesma ação, ao mesmo tempo, no mesmo objeto, e um processo fica aguardando o outro remover o lock para continuar a operação.

# Controle de Concorrência

## EXPLICANDO O DEADLOCK

### Sessão 58

```
-- Passo 1 - Vou iniciar uma transação e deixá-la aberta (sem COMMIT ou ROLLBACK)
BEGIN TRANSACTION
```

```
-- Passo 2 - Vou travar a Tabela1
UPDATE dbo.Tabela1
SET Nome = 'Teste'
WHERE Id = 1
```

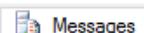
```
-- Passo 5 - Vou tentar travar a Tabela (já possui lock na outra sessão)
UPDATE dbo.Tabela2
SET Nome = 'Teste'
WHERE Id = 1
```

### Sessão 55

```
-- Passo 3 - Vou iniciar uma transação e deixá-la aberta (sem COMMIT ou ROLLBACK)
BEGIN TRANSACTION
```

```
-- Passo 4 - Vou travar a Tabela2
UPDATE dbo.Tabela2
SET Nome = 'Teste'
WHERE Id = 1
```

```
-- Passo 6 - Ao tentar travar a Tabela 1, irá ocorrer o deadlock
UPDATE dbo.Tabela1
SET Nome = 'Deadlock!!!'
WHERE Id = 1
```



Msg 1205, Level 13, State 45, Line 10  
Transaction (Process ID 55) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

# Controle de Concorrência

## TIPOS DE BLOQUEIOS

Existe vários tipos de bloqueios, os dois mais utilizados são:

**SHARED LOCKS (S):** Também conhecidos como **locks de leitura**, permitem que várias transações leiam um recurso ao mesmo tempo, mas impedem que outras transações o modifiquem enquanto o lock estiver ativo.

**EXCLUSIVE LOCKS (X):** Também conhecidos como **locks de escrita**, impedem que outras transações accessem ou modifiquem um recurso enquanto um lock exclusivo está ativo.

# Controle de Concorrência

## TIPOS DE BLOQUEIOS

COMO VERIFICAR OS TIPOS DE LOCKS DE TRANSAÇÃO NO BANCO DE DADOS?

```
SELECT * FROM sys.dm_tran_locks;  
go
```

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type
1	DATABASE		6		0	0	S	LOCK
2	DATABASE		6		0	0	S	LOCK
3	OBJECT		6		485576768	0	IS	LOCK
4	PAGE		6	1:1288	72057594042974208	0	S	LOCK

	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type
1	DATABASE		6		0	0	S	LOCK
2	DATABASE		6		0	0	S	LOCK
3	OBJECT		6		485576768	0	X	LOCK

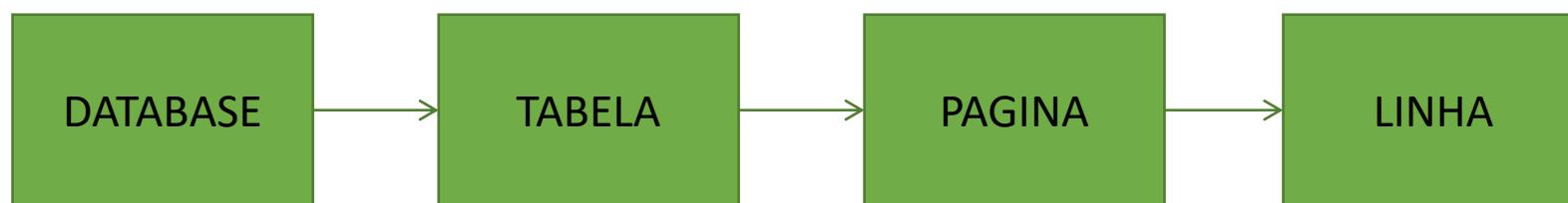
# Controle de Concorrência

## HIERARQUIA DE LOCKS

**LOCKS EM OBJETOS:** São locks que protegem objetos de banco de dados, como tabelas, índices, procedimentos armazenados, entre outros. Eles são o nível mais alto de locks e cobrem todo o objeto.

**LOCKS EM PÁGINAS:** São locks que protegem as páginas de dados do objeto. Cada página pode ser compartilhada por várias transações, mas apenas uma transação pode modificar a página ao mesmo tempo.

**LOCKS EM LINHAS:** São locks que protegem linhas individuais de uma tabela. Row locks são usados principalmente para minimizar o tempo de bloqueio e permitir o acesso concorrente aos dados.



# Controle de Concorrência

## HINTS

**HINT** (Dica) é uma regra que diz ao otimizador do SQL Server o que deve estar contemplado no plano de execução da consulta - de forma explícita - ao rodar uma instrução SQL e, obviamente, o otimizador deverá atender.

Existe vários tipos de hints para gestão do controle de concorrência, mas iremos abordar os dois mais comuns:

- NOLOCK;
- HOLDLOCK;

# Controle de Concorrência

## HINTS

### **NOLOCK**

Quando esta opção é selecionada, o SQL Server não adiciona nenhum bloqueio ao ler ou modificar dados. Nesse caso, é possível que o usuário faça a leitura dos dados da transação incompleta, conhecidos como “leitura suja”.

### **EXEMPLO DE USO**

```
SELECT * FROM MICRODADOS_ENEM_2021_SC WITH (NOLOCK)
```

# Controle de Concorrência

## HINTS

### **HOLDLOCK**

Quando esta opção é selecionada, o SQL Server irá bloquear os dados para qualquer alteração até que bloqueio seja liberado.

### **EXEMPLO DE USO**

```
SELECT * FROM MICRODADOS_ENEM_2021_SC WITH (HOLDLOCK)
```

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios Linguagem SQL 4”**

Utilizando os itens vistos em aula:

**PROCESSAMENTO DE TRANSAÇÕES E CONTROLE DE CONCORRÊNCIA.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

# Stored Procedure

## INTRODUÇÃO A PROGRAMAÇÃO TRANSACT-SQL

**Transact-SQL ou Transact-SQL** fornece uma linguagem de programação com recursos que permitem armazenar temporariamente valores em variáveis, aplicar a execução condicional de comandos, passar parâmetros para procedimentos armazenados e controlar o fluxo de seus programas.

```
declare @aux varchar(120)
declare @aux2 varchar(25)
declare @fim varchar(25)

--Pegar o primeiro nome
--set @fim=left(@nome,charindex(' ',@nome))
set @fim=substring(@nome, 0, 4)

set @aux=@fim --Inicializa variável
set @aux2='' --Inicializa variável

--Enquanto tem nomes do meio
while len(@aux)>0
begin
    --pega primeiro nome
    set @aux=left(@nome,charindex(' ',@nome))
    --remove primeiro nome
    set @nome=ltrim(replace(@nome,@aux,''))
```

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

**Tabelas Temporárias** são diferentes de tabelas permanentes, pois elas existem apenas durante um curto espaço de tempo.

Usamos tabelas temporárias quando os resultados da manipulação dos dados precisam ser armazenados temporariamente em uma tabela ou quando não temos permissão para criar tabelas permanentes para manipulação dos dados.

No SQL Server, temos três tipos de tabelas temporárias:

- Locais,
- Globais,
- Tipo variável.

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

**Tabelas Temporárias Locais** são utilizadas somente dentro de uma sessão (ou procedimento), outras conexões ou transações não terão acesso a esta tabela, apenas a conexão ao qual ela foi executada.

Bastante utilizada em procedimentos armazenados, funções, triggers e scripts de manipulação de dados.

O procedimento para criação da tabela temporária local é similar ao da tabela permanente (mesmos comandos de DDL).

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

### SINTAXE SIMPLIFICADA

```
CREATE TABLE #table_name (
    column_name datatype [ NULL | NOT NULL ] ,
    column_name datatype [ NULL | NOT NULL ] ,...
    column_name datatype [ NULL | NOT NULL ] )
```

### EXEMPLO

```
create table #tabela_teste
(id_aluno    integer,
nm_aluno    char(10),
nm_cidade   char(15))
go
```

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

**Tabelas Temporárias Globais** são utilizadas por qualquer sessão do SQL Server, até que o servidor seja reiniciado.

O procedimento de criação da tabela temporária local é similar ao da tabela permanente (mesmos comandos de DDL).

São menos frequentemente utilizadas do que as tabelas temporárias locais.

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

### SINTAXE SIMPLIFICADA

```
CREATE TABLE ##table_name (
    column_name datatype [ NULL | NOT NULL ] ,
    column_name datatype [ NULL | NOT NULL ] ,...
    column_name datatype [ NULL | NOT NULL ] )
```

### EXEMPLO

```
create table ##tabela_teste
(id_aluno    integer,
nm_aluno     char(10),
nm_cidade   char(15))
go
```

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

**Tabelas Tipo Variáveis** são semelhantes as tabelas temporárias locais com algumas limitações.

- Não são afetadas por rollback (temp table locais são afetadas por rollback).
- Utilizadas normalmente com uma pequena quantidade de dados (algumas centenas de dados).
- Não é possível criar índices nelas.

Bastante utilizada em procedimentos armazenados, funções, triggers e scripts de manipulação de dados.

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS

### SINTAXE SIMPLIFICADA

```
DECLARE @table_variable_name TABLE (
    column_list
) ;
```

### EXEMPLO

```
DECLARE @tabela_teste TABLE
(id_aluno    integer,
nm_aluno    char(10),
nm_cidade   char(15))
go
```

# Programação Transact-SQL

## TABELAS TEMPORÁRIAS - RESUMO

### TEMPORÁRIA LOCAL (SESSÃO)

```
CREATE TABLE #table_name (
    column_list
);
```

### TEMPORÁRIA GLOBAL

```
CREATE TABLE ##tabela_teste (
    column_list
);
```

### TEMPORÁRIA TIPO VARIÁVEL

```
DECLARE @table_variable_name TABLE (
    column_list
);
```

# Programação Transact-SQL

## COMENTÁRIOS

### SINTAXE

```
/* uma ou várias linhas de comentários  
...  
*/  
  
-- Uma única linha de comentário
```

Teclas de atalho do SSMS:

Adicionar Comentário  
**Control + K Control + C**

Remover Comentário  
**Control + K Control + U**

### EXEMPLO

```
-- Converte a data atual para o formato dd/mm/yyyy  
SELECT convert(char(10), getdate(), 103)  
go
```

# Programação Transact-SQL

## VARIÁVEIS

**VARIÁVEIS LOCAIS** – Usadas somente dentro da sessão (ou procedimento) ao qual foi criada.

**VARIÁVEIS GLOBAIS** – Utilizada por qualquer sessão do SQL Server.

# Programação Transact-SQL

## VARIÁVEIS LOCAIS

- Uma variável local é um local nomeado na memória definido pelo usuário para armazenar um valor.
- Variáveis são usadas em stored *procedures*, *triggers* e *scripts SQL* e são definidas com o comando *DECLARE*.
- As variáveis devem ser precedidas por um único @.
- O tempo de vida de uma variável permanece até o término da execução do *stored procedure* ou *trigger*.
- As variáveis locais devem ser declaradas antes da sua utilização.
- Quando declarada, o valor da variável local permanece NULL até que seja atribuído algum valor para ela.

# Programação Transact-SQL

## VARIÁVEIS LOCAIS

### SINTAXE

```
DECLARE @nome_variável datatype,  
        @nome_variável datatype,...
```

### EXEMPLOS

```
DECLARE @cd_usuario char(10),  
        @cd_tipo      int
```

# Programação Transact-SQL

## VISUALIZANDO VARIÁVEIS LOCAIS

### SINTAXE

```
SELECT @nome_variavel
```

### EXEMPLOS

```
DECLARE @cd_usuario char(10) = 'Teste'  
SELECT @cd_usuario
```

# Programação Transact-SQL

## ATRIBUINDO VALORES AS VARIÁVEIS LOCAIS

### SINTAXE

```
SELECT variable_name = column_name  
FROM table_name  
[WHERE condition]
```

```
SELECT variable_name = expression  
[, variable_name = expression ...]
```

```
SET variable_name = expression
```

# Programação Transact-SQL

## ATRIBUINDO VALORES AS VARIÁVEIS LOCAIS

### EXEMPLOS

```
DECLARE @number int,  
        @copy int,  
        @sum int  
  
SET @number = 10  
SELECT @copy = @number, @sum = @number + 100
```

```
DECLARE @AD_id char(11)  
  
SELECT @AD_id = au_id  
      FROM authors  
     WHERE au_fname = 'Ann' and au_lname = 'Dull'
```

# Programação Transact-SQL

## VARIÁVEIS LOCAIS

### OBSERVAÇÕES

- Um único comando de *SELECT* pode ser usado para atribuir um valor a uma variáveis ou para consultar dados dela, mas não ambos.
- O tempo de vida da variável é determinado pelo tempo de duração da *stored procedure* ou *trigger* em que ela foi declarada, e não está disponível para outro processo.
- Se uma atribuição é feita em um comando *SELECT* que retorna mais de uma linha, a variável local recebe o valor da última linha retornada.

# Programação Transact-SQL

## VARIÁVEIS LOCAIS

```
-- Declara uma variável local
DECLARE @pub_var CHAR(4)

-- Atribui um valor a partir de uma consulta
SELECT @pub_var = pub_id
FROM publishers
WHERE pub_name = "New Age Books"

-- Visualiza o valor da variável
SELECT @pub_var

-- Utiliza a variável como parte do select
SELECT title_id, title, pub_id
FROM titles
WHERE pub_id = @pub_var

-- Terminados do batch
go
```

# Programação Transact-SQL

## VARIÁVEIS GLOBAIS

Uma variável global é uma local nomeado na memória definido e mantido pelo SQL Server.

Regras para as variáveis globais:

- Nomes começam com “@”
- Não podem ser criadas por usuários
- Não podem ser atribuidos valores a elas por usuários

# Programação Transact-SQL

## VARIÁVEIS GLOBAIS

Variável	Descrição
@@rowcount	Número de linhas processadas pelo último comando
@@error	Número do erro reportado pelo último comando
@@trancount	Nível de aninhamento das transações
@@servername	Nome do servidor conectado
@@version	Versão do SQL Server e tipo de S.O.
@@spid	ID do processo corrente
@@nestlevel	Níveis de aninhamento de <i>stored procedures</i> e <i>triggers</i>
@@identity	Último valor de um identity utilizado em um insert
@@fetch_status	Situação do último comando fetch em um cursor

# Programação Transact-SQL

## PRINT COM STRINGS OU VARIÁVEIS

- O comando PRINT é usado para passagem mensagem para o client.
- As mensagens podem incluir até 1024 caracteres.

### SINTAXE

```
PRINT { "user_message" | variable_name }
```

### EXEMPLOS

```
PRINT N'This user has SET NOCOUNT turned ON.';
```

```
DECLARE @temp_title_id varchar(100)
SET @temp_title_id = N'There is no user by that name.'
PRINT @temp_title_id
```

# Programação Transact-SQL

## BEGIN... END

BEGIN e END são usados para agrupar instruções Transact-SQL dentro de um contexto de execução.

Normalmente utilizados dentro de IF, WHILE e também para executar códigos em batch.

### SINTAXE

```
BEGIN  
    Código_SQL  
END
```

### EXEMPLOS

```
DECLARE @Iteration int = 0;  
WHILE @Iteration < 10  
BEGIN  
    SELECT FirstName, MiddleName  
    FROM dbo.DimCustomer WHERE LastName = 'Adams';  
    SET @Iteration += 1 ;  
END;
```

**Não seria executado**

Se o bloco BEGIN...END não for incluído, o exemplo a seguir ficará em um loop contínuo.

# Programação Transact-SQL

## IF... ELSE

### SINTAXE SIMPLIFICADA

```
IF <expressão_booleana>
BEGIN
    <comando>
END
ELSE
BEGIN
    <comando>
END
```

# Programação Transact-SQL

## IF... ELSE

### EXEMPLOS

CONDIÇÃO

```
IF @cd_contador is null  
    SELECT @cd_contador = 1  
ELSE
```

Se CONDIÇÃO for VERDADEIRO, executa este item

```
    SELECT @cd_contador = @cd_contador + 1
```

Se CONDIÇÃO for FALSO, executa este item

```
IF @cd_contador IS NULL  
BEGIN  
    SELECT @cd_tipo = 2  
    SELECT @cd_contador = 1  
END  
    SELECT @cd_contador = @cd_contador + 1
```

# Programação Transact-SQL

## IF EXISTS

### SINTAXE SIMPLIFICADA

```
IF [ NOT ] EXISTS ( select_statement )
    code_to_execute_when_condition_is_true
[ ELSE
    code_to_execute_when_condition_is_false ]
```

### EXEMPLO

```
IF EXISTS (SELECT * FROM MICRODADOS_ENEM_2021_SC WHERE NO_MUNICIPIO_ESC
= 'Treviso')
    PRINT 'OK'
ELSE
    PRINT 'NAO OK'
```

# Programação Transact-SQL

## WHILE

### SINTAXE

```
WHILE condition
      block_to_execute
```

### EXEMPLO

```
DECLARE @CNT INT
SET @CNT=0

WHILE ( @CNT <= 5)
BEGIN
    PRINT @CNT
    SET @CNT = @CNT + 1
END
```

```
WHILE (SELECT AVG(price) FROM titles) < 40
BEGIN
    UPDATE titles
    SET price = price + 2
END
```

# Programação Transact-SQL

## BREAK

Usado para sair de um loop **while**.

### EXEMPLO

```
WHILE (SELECT AVG(price) FROM titles) > 20
BEGIN
    UPDATE titles
    SET price = price / 2
    IF (SELECT MAX(price) from titles) < 40
        BREAK
END
```

# Programação Transact-SQL

## CONTINUE

Retorna o fluxo da execução para topo do loop **while**

### EXEMPLO

```
WHILE @price < 20.00
BEGIN
    SELECT @price = @price + 1.00
    IF (SELECT COUNT(price) FROM titles WHERE price = @price) >= 5
        CONTINUE
    ELSE
        UPDATE titles SET price = price * 1.10 WHERE price = @price
END
```

# Programação Transact-SQL

## RETURN

Causa a saída (finalização) de um código SQL

### EXEMPLO

```
DECLARE @avg_price numeric(14,2)

SELECT @avg_price = AVG(price) FROM titles

IF @avg_price < $10
    RETURN
WHILE @avg_price < $20
BEGIN
    UPDATE titles SET price = price * 1.05
    SELECT @avg_price = AVG(price) FROM titles
END
```

# Programação Transact-SQL

## CASE WHEN... THEN...

**CASE** é um comando usado para definir condições que serão testadas durante a execução de um código e, caso sejam atendidas, entregarão um determinado resultado.

### SINTAXE SIMPLES

```
CASE
    WHEN condição1 THEN resultado1
    WHEN condição2 THEN resultado2
    WHEN condiçãoN THEN resultadoN
    ELSE resultado
END;
```

# Programação Transact-SQL

## CASE WHEN... THEN...

### EXEMPLO

```
SELECT ProductNumber,
       Category = CASE ProductLine
                     WHEN 'R' THEN 'Road'
                     WHEN 'M' THEN 'Mountain'
                     WHEN 'T' THEN 'Touring'
                     WHEN 'S' THEN 'Other sale items'
                     ELSE 'Not for sale'
                   END,
       Name
  FROM Production.Product
 ORDER BY ProductNumber;
GO
```

```
SELECT ProductNumber,
       Name,
       "Price Range" = CASE
                     WHEN ListPrice = 0 THEN 'Mfg item - not for resale'
                     WHEN ListPrice < 50 THEN 'Under $50'
                     WHEN ListPrice >= 50 AND ListPrice < 250 THEN 'Under $250'
                     WHEN ListPrice >= 250 AND ListPrice < 1000 THEN 'Under $1000'
                     ELSE 'Over $1000'
                   END
  FROM Production.Product
 ORDER BY ProductNumber;
GO
```

# Exercícios

Resolver a lista de exercícios do link abaixo:

<https://learn.microsoft.com/pt-br/training/modules/get-started-transact-sql-programming/6-exercise-program-transact-sql?ns-enrollment-type=learningpath&ns-enrollment-id=learn.wwl.program-transact-sql>

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

# Stored Procedures

## DEFINIÇÃO

**STORED PROCEDURE** ou **PROCEDIMENTO ARMAZENADO** é um grupo de uma ou mais instruções do Transact-SQL executado de forma única.

- Aceita parâmetros de entrada e saída.
- Encapsulam tarefas repetitivas.
- Reduz o tráfego de rede.
- São capazes de utilizar os comandos como IF e ELSE, WHILE, CASE, tabelas temporárias, cursores, variáveis, dentro outros.
- Permitem chamar outros procedimentos armazenados dentro dele (execução aninhada).
- Permite utilizar os comandos do grupo DML e alguns do DDL.

# Stored Procedures

## VANTAGENS E DESVANTAGENS

### PRINCIPAIS VANTAGENS

Desempenho/Performance,  
Segurança,  
Reutilização de código.

### PRINCIPAL DESVANTAGEM

Dependência da tecnologia do SGBD.

# Stored Procedures

## TIPOS DE STORED PROCEDURES

### DEFINIDAS PELO USUÁRIO

- Procedimentos de usuário que são executados explicitamente

### TRIGGERS

- Procedimentos de usuário que são executados automaticamente quando há alguma modificação na tabela vinculada (disparo por evento)

### SYSTEM PROCEDURES

- Procedimentos de sistema que lêem ou modificam uma ou mais tabelas de sistema

# Stored Procedures

## CRIANDO UMA STORED PROCEDURE

### SINTAXE

```
CREATE {PROC | PROCEDURE} procedure_name AS  
BEGIN  
    statements  
END  
go
```

### EXEMPLO

```
CREATE PROCEDURE pr_atualiza_titulo AS  
BEGIN  
    UPDATE titulo  
        SET preco = preco * 0.95  
        WHERE total < 3000  
END  
go
```

# Stored Procedures

## EXCLUINDO UMA STORED PROCEDURE

### SINTAXE

```
DROP PROCEDURE procedure_name
```

### EXEMPLO

```
DROP PROCEDURE pr_atualiza_titulo
```

### VALIDANDO A EXISTÊNCIA DA PROCEDURE ANTES DE EXCLUIR

```
IF EXISTS(SELECT 1 FROM sys.procedures  
          WHERE name = 'pr_atualiza_titulo')  
    DROP PROCEDURE pr_atualiza_titulo  
  
go
```

Versão 2014 ou inferior

```
DROP PROCEDURE IF EXISTS pr_atualiza_titulo  
go
```

Versão 2016 ou superior

# Stored Procedures

## EXECUTANDO UMA STORED PROCEDURE

### SINTAXE

```
[exec | execute] procedure_name
```

### EXEMPLO

```
execute pr_atualiza_titulo  
go
```

# Stored Procedures

## VARIÁVEIS EM STORED PROCEDURE

- Stored procedures podem criar e usar variáveis locais.
- As variáveis existirão somente durante a execução da stored procedure.
- As variáveis de uma procedure não poderão ser usadas por outros processos concorrente.

### EXEMPLO

```
CREATE PROC pr_soma AS
BEGIN
    DECLARE @valor_1 int, @valor_2 int

    SELECT @valor_1 = 2, @valor_2 = 3
    --SET @valor_1 = 2
    --SET @valor_2 = 3

    select @valor_1 + @valor_2
END
go
```

```
EXEC pr_soma
go

-----
5
(1 row affected)
```

# Stored Procedures

## PARAMETROS DE ENTRADA

### SINTAXE

```
CREATE PROCEDURE procedure_name  
    (@variavel tipo, @variavel tipo, ... ) as  
BEGIN  
    --statements  
END
```

### EXEMPLO

```
CREATE PROC pr_soma (@valor_1 int, @valor_2 int) AS  
BEGIN  
    SELECT @valor_1 = 2, @valor_2 = 3  
    --SET @valor_1 = 2  
    --SET @valor_2 = 5  
  
    SELECT @valor_1 + @valor_2  
END  
go
```

# Stored Procedures

## PARAMETROS DE ENTRADA : POSIÇÃO X OUTRAS VARIÁVEIS

### EXEMPLO

```
create proc myproc (@val1 int, @val2 int) as  
    ...
```

### PARAMETROS PASSADOS PELA POSIÇÃO

```
exec myproc 10, 20
```

### PARAMETROS PASSADOS POR VARIÁVEL

```
declare @valor1 int, @valor2 int  
  
set @valor1 = 2  
set @valor2 = 2  
  
exec myproc @valor1, @valor2
```

# Stored Procedures

## PARAMETROS DE ENTRADA : DEFAULT

Um valor *default* é atribuído ao parâmetro para os casos onde nenhum valor é fornecido ao parâmetro no momento de sua execução.

### EXEMPLO

```
create proc pr_state_authors (@state char(2) = 'CA') as  
select au_lname, au_fname, state  
from authors  
where state = @state
```

```
exec proc_state_authors --Nao foi passado nenhum parametro
```

au_lname	au_fname	state
White	Johnson	CA
Green	Marjorie	CA
...		

# Stored Procedures

## PARAMETROS DE SAIDA (OUTPUT)

### SINTAXE

```
CREATE PROCEDURE procedure_name
    (@variavel tipo, @variavel tipo OUTPUT, ... ) as
BEGIN
    --statements
END
```

### EXEMPLO

```
CREATE PROC pr_new_price (@title_id int,
                        @new_price numeric(14,2) output) as
BEGIN
    SELECT @new_price = price FROM titles
    WHERE title_id = @title_id

    SELECT @new_price = @new_price * 1.15
END
```

# Stored Procedures

## PARAMETROS DE SAIDA (OUTPUT) - EXECUÇÃO

### SINTAXE

```
[exec | execute] procedure_name variable output
```

### EXEMPLO

```
declare @title_id char(6), @new_price numeric(14,2)

select @title_id = 'PC8888'

exec proc_new_price @title_id, @new_price output

select @new_price
go
```

---

23.00

# Stored Procedures

## STORED PROCEDURES DE SISTEMAS

**sp\_depends {table\_name | procedure\_name}**

- Quando informado uma tabela, ele lista todos os objetos (incluindo procedures) no mesmo banco de dados que fazem referencia a tabela
- Quando informado uma proc, lista todas as tabelas no mesmo banco de dados referenciadas pela proc

**sp\_help procedure\_name**

- Mostra informações sobre uma determinada stored procedure

**sp\_helptext procedure\_name**

- Mostra o texto (código) usado para criar a stored procedure

**sp\_rename old\_proc\_name, new\_proc\_name**

- Renomeia o nome da stored procedure

# Stored Procedures

## LIMITAÇÕES

As instruções a seguir não podem ser usadas em qualquer lugar no corpo de um stored procedure.

CREATE	SET	USE
CREATE AGGREGATE	SET SHOWPLAN_TEXT	USE <i>database_name</i>
CREATE DEFAULT	SET SHOWPLAN_XML	
CREATE RULE	SET PARSEONLY	
CREATE SCHEMA	SET SHOWPLAN_ALL	
CREATE ou ALTER TRIGGER		
CREATE ou ALTER FUNCTION		
CREATE ou ALTER PROCEDURE		
CREATE ou ALTER VIEW		

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios Stored Procedure 1”**

Utilizando os itens vistos em aula:

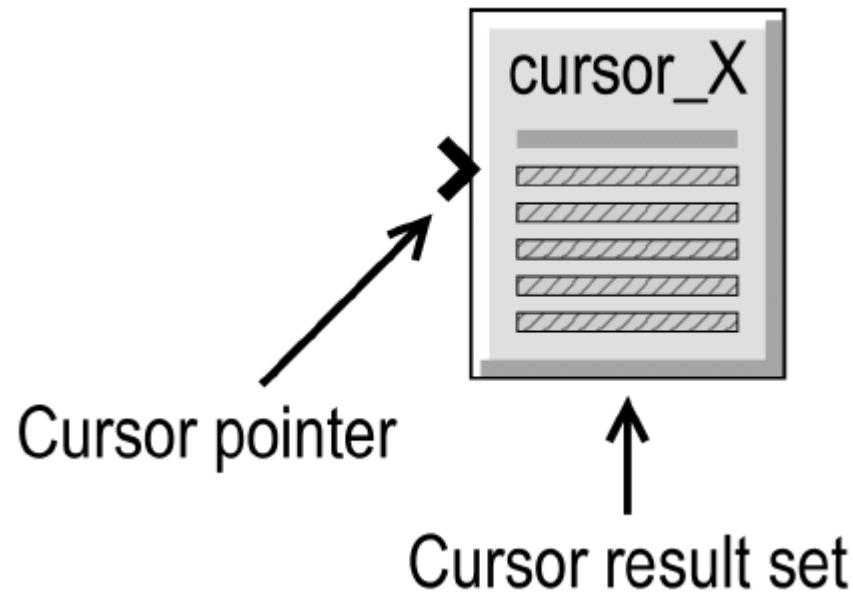
**PROGRAMAÇÃO T-SQL E STORED PROCEDURE.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

# Stored Procedures

## CURSORES

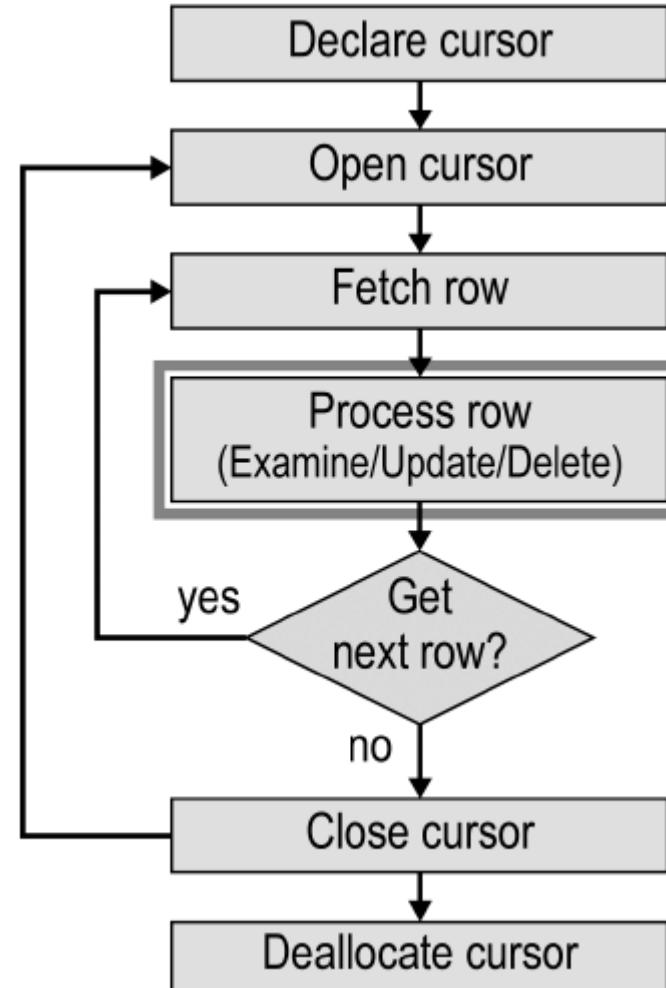
- Um cursor é um mecanismo que processa o resultado de uma query definida, linha a linha.
- O **cursor result set** é o conjunto de linhas retornadas pela query definida.
- O **cursor pointer** aponta para uma linha dentro do result set.



# Stored Procedures

## CICLO DE VIDA DE UM CURSOR

1. Declarar o cursor
2. Abrir o cursor
3. Buscar cada linha
4. Fechar o cursor
5. Desalocar o cursor



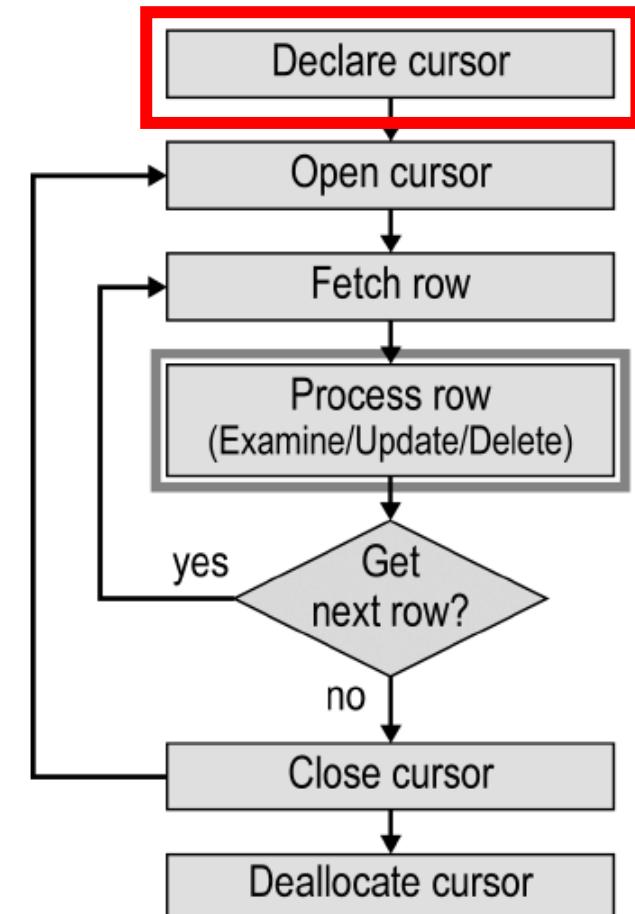
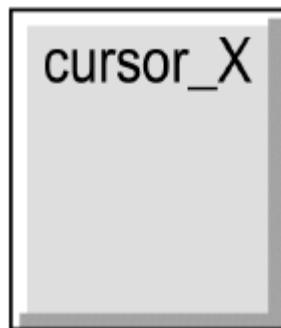
# Stored Procedures

## CURSORES - DECLARE

### PASSO 1: DECLARAR O CURSOR

Quando você declara um cursor:

- Você especifica a query
- O SQL reserva recursos de memória para o cursor
- O SQL utilizará a consulta definida para montar o conjunto de linhas que serão retornadas.



# Stored Procedures

## CURSORES - DECLARE

### SINTAXE

```
declare cursor_name cursor  
for select_statement
```

### EXEMPLO

```
Nome do cursor  
declare cur_biz_book cursor  
      for select title, title_id from titles  
          where type = 'business'  
  
SELECT do cursor
```

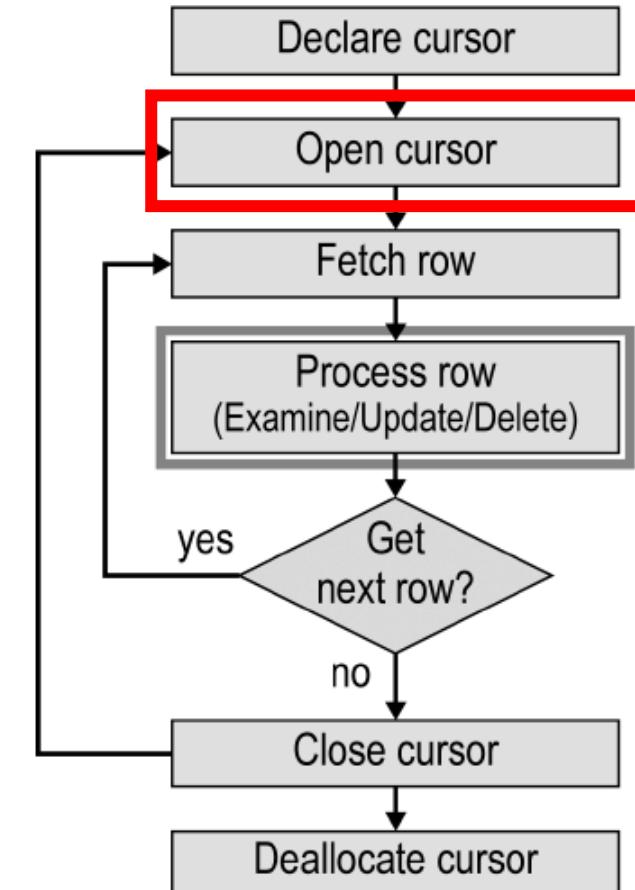
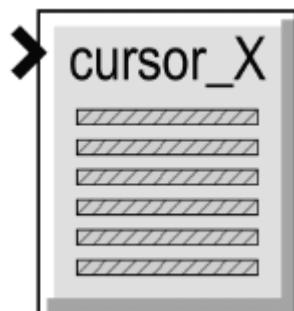
# Stored Procedures

## CURSORES - OPEN

### PASSO 2: ABRIR O CURSOR

Quando você abre o cursor:

- O SQL inicia a criação do result set
- O ponteiro é posicionado antes da primeira linha do result set



# Stored Procedures

## CURSORES - OPEN

### SINTAXE

```
open cursor_name
```

### EXEMPLO

```
declare cur_biz_book cursor  
for select title, title_id from titles  
      where type = 'business'
```

```
open cur_biz_book
```

# Stored Procedures

## CURSORES - FETCH

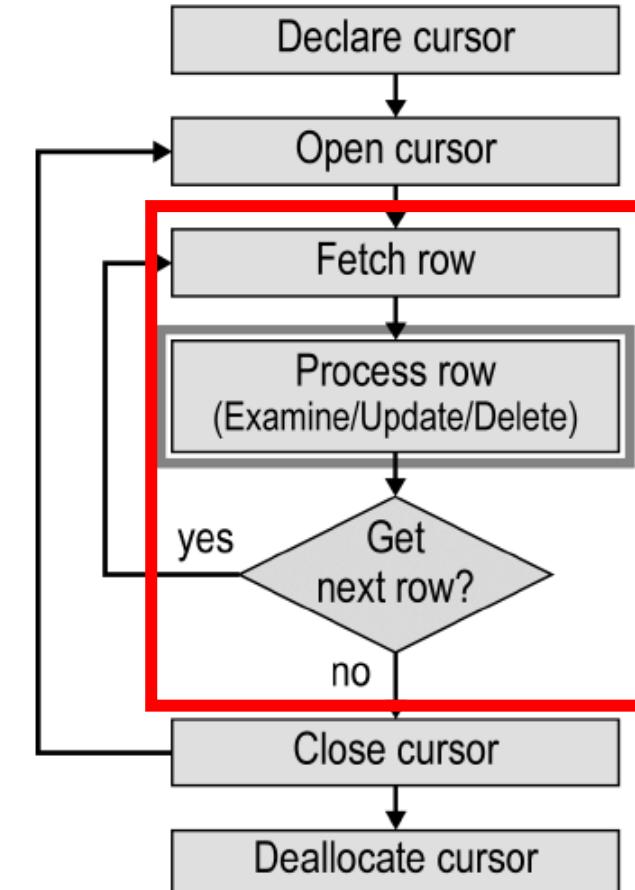
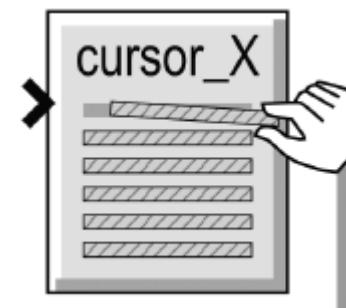
### PASSO 3: BUSCAR CADA LINHA

Quando você executa um **fetch**:

- O ponteiro se move para a próxima linha válida do **result set**
- Ele retorna o próxima linha válida do **result set**

#### VARIÁVEL GLOBAL `@@FETCH_STATUS`

- Retorna 0 se sucesso
- Retorna -1 se falha ou fim das linhas do **result set**.



# Stored Procedures

## CURSORES – FETCH COM @@FETCH\_STATUS

### SINTAXE

```
fetch cursor_name [ into fetch_target_list ]
```

### EXEMPLO

```
declare cur_biz_book cursor
    for select title, title_id from titles
        where type = 'business'
open cur_biz_book
declare @title char(80), @title_id char(6)

while 0=0
begin
    fetch cur_biz_book into @title, @title_id
    if @@FETCH_STATUS <> 0
        break

    --continua códigos sql
end
```

### Outra maneira de efetuar FETCH

```
fetch cur_biz_book into @title, @title_id;

while @@FETCH_STATUS = 0
begin

    -- código sql

    fetch cur_biz_book into @title, @title_id
end
```

# Stored Procedures

## CURSORES - CLOSE E DEALLOCATE

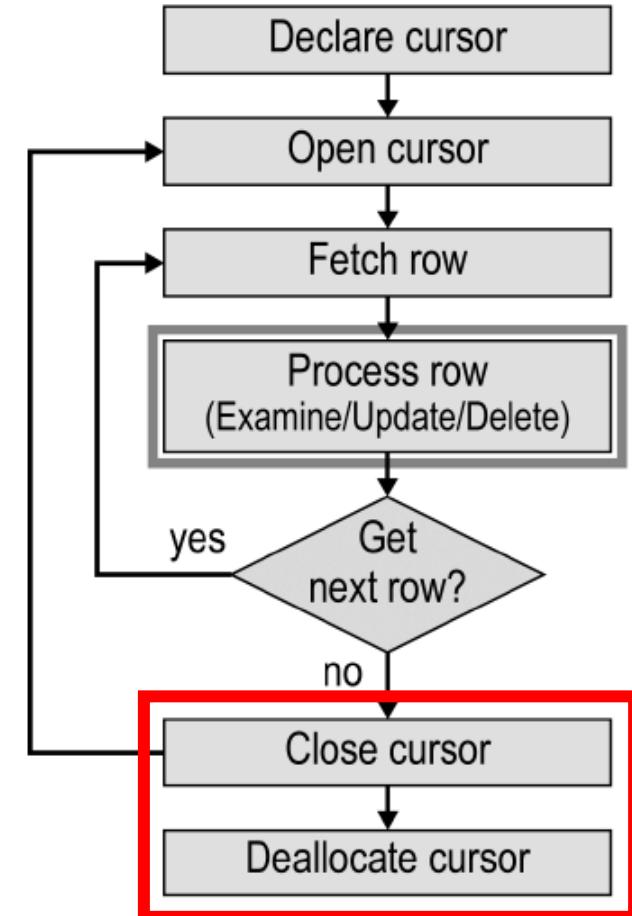
### PASSO 4 E 5: FECHAR E DESALOCAR O CURSOR

Quando você **fecha** um cursor:

- O processamento da query é parado

Quando você **desaloca** um cursor:

- Todo o recurso de memória alocado para o cursor é liberado



# Stored Procedures

## CURSORES - CLOSE E DEALLOCATE

### SINTAXE

```
close cursor_name  
deallocate cursor_name
```

### EXEMPLO

```
declare cur_biz_book cursor  
    for select title, title_id from titles  
        where type = 'business'  
open cur_biz_book  
declare @title char(80), @title_id char(6)  
while 0=0  
begin  
    fetch cur_biz_book into @title, @title_id  
    if @@FETCH_STATUS <> 0  
        break  
    --continua códigos sql  
end  
close cur_biz_book  
deallocate cur_biz_book
```

# Stored Procedures

## EXEMPLOS E CASOS DE USO

### CASOS DE USO PARA UTILIZAÇÃO DE CURSORES

- Execução de tarefas repetitivas (criação de logins em lote, atualização de rotinas administrativas, envio de e-mails, sms, etc.),
- Atualização de registros em lote,
- Geração de relatórios complexos,
- Validação e correção de dados.

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios Stored Procedure 2”**

Utilizando os itens vistos em aula:

**PROGRAMAÇÃO T-SQL, STORED PROCEDURE e CURSORES.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

# Triggers

**TRIGGER ou GATILHO** são um tipo especial de procedimento armazenado executado automaticamente quando um evento ocorre no servidor de banco de dados.

Existem 3 grupos de triggers:

## Escopo de estudo

- **Triggers de DML** – Triggers que reagem a comandos de DML como: INSERT, UPDATE e DELETE.
- **Triggers de DDL** – Triggers que reagem a comandos de DDL como: CREATE, ALTER e DROP.
- **Triggers de Logon** – Triggers que reagem a eventos de logon.

# Triggers

**TRIGGER DE DML** são as triggers mais utilizadas em um banco de dados relacional.

No SQL Server existem 2 tipos principais de triggers de DML:

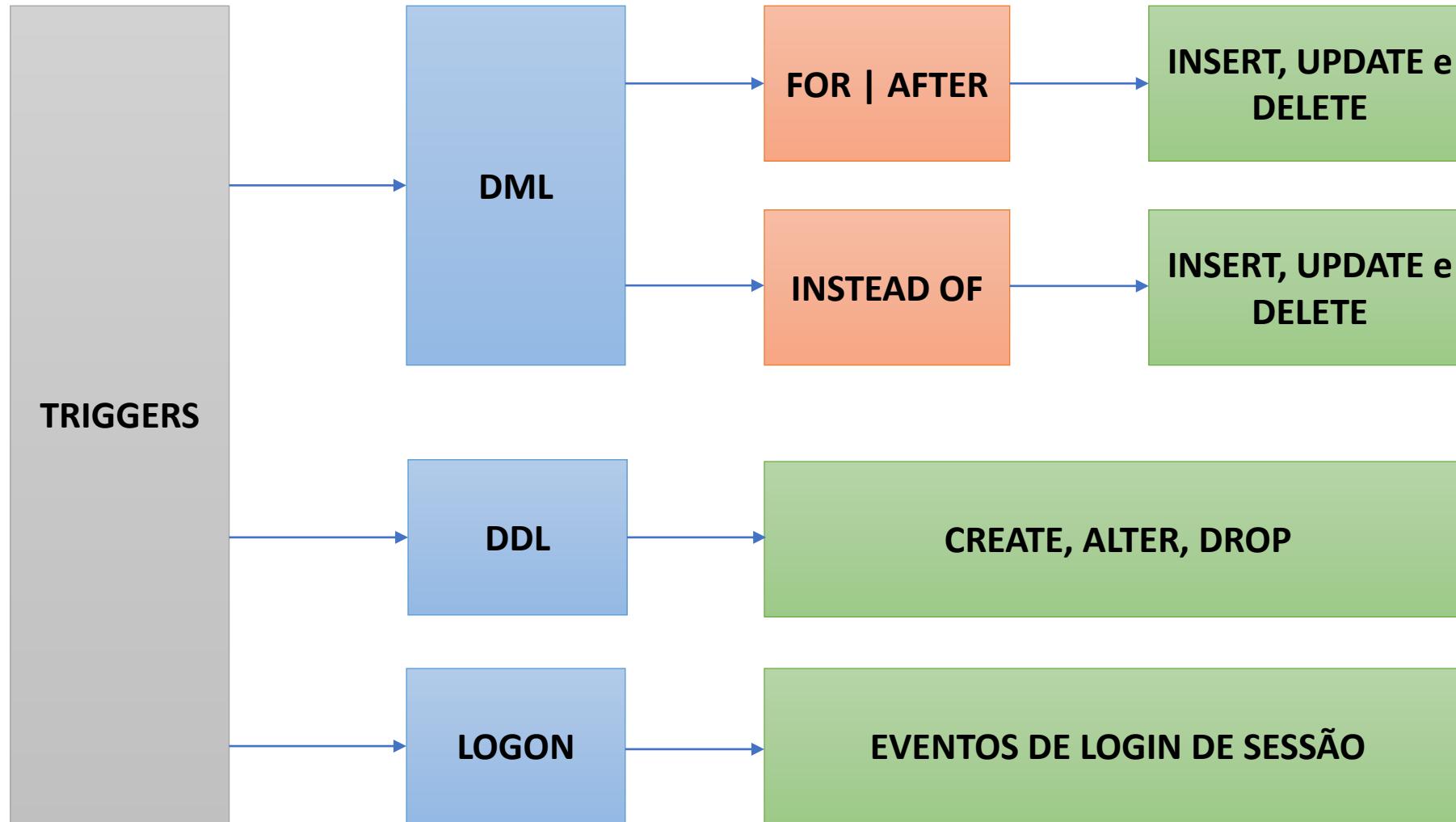
## FOR | AFTER

Especifica que a trigger de DML é disparada apenas quando todas as operações especificadas na instrução SQL de gatilho foram iniciadas com êxito.

## INSTEAD OF

Especifica que a trigger de DML será iniciada *em vez* da instrução SQL de gatilho, substituindo as ações das instruções de gatilho. Não é possível especificar INSTEAD OF para gatilhos DDL ou de logon.

# Triggers



# Triggers

## CRIANDO UMA TRIGGER

### SINTAXE

```
create trigger trigger_name on table_name
for {insert | update | delete} [, {insert | update | delete}]
...
as
sql_statements
```

### EXEMPLO

```
create trigger ti_sales on sales for insert as
begin
    if datename (dw, getdate()) = 'Terça-Feira'
    begin
        raiserror ('Vendas não podem ser feitas na terça.', 16,1)
        rollback tran
        return
    end
end
```

AFTER INSERT  
INSTEAD OF INSERT

# Triggers

## EXCLUINDO UMA TRIGGER

### SINTAXE SIMPLIFICADA

```
DROP TRIGGER trigger_name
```

### EXEMPLO

```
DROP TRIGGER ti_sales
```

### VALIDANDO A EXISTÊNCIA DA TRIGGER ANTES DE EXCLUIR

```
DROP TRIGGER IF EXISTS ti_sales  
go
```

# Triggers

## DESABILITANDO E HABILITANDO UMA TRIGGER

### SINTAXE

```
alter table table_name
{enable | disable} trigger trigger_name
```

OU

```
{enable | disable} trigger trigger_name on table_name
```

### EXEMPLO

```
ALTER TABLE sales DISABLE TRIGGER ti_sales
DISABLE TRIGGER ti_sales ON sales
```

# Triggers

## PROCEDURES DE SISTEMAS PARA TRIGGERS

**sp\_depends {table\_name | trigger\_name}**

- Quando informado uma tabela, ele lista todos os objetos (incluindo triggers) no mesmo banco de dados que fazem referência a tabela.
- Quando informada uma trigger, lista todas as tabelas no mesmo banco de dados referenciadas pela trigger.

**sp\_help trigger\_name**

- Mostra informações sobre uma determinada trigger

**sp\_helptext triggers\_name**

- Mostra o texto (código) usado para criar a trigger

**sp\_rename old\_triggers\_name, new\_trigger\_name**

- Renomeia o nome da stored trigger

# Triggers

## TABELAS INSERTED E DELETED

**INSERTED** e **DELETED** são duas tabelas virtuais automaticamente criadas, sempre que uma trigger é disparada, dentro do escopo de execução da trigger.

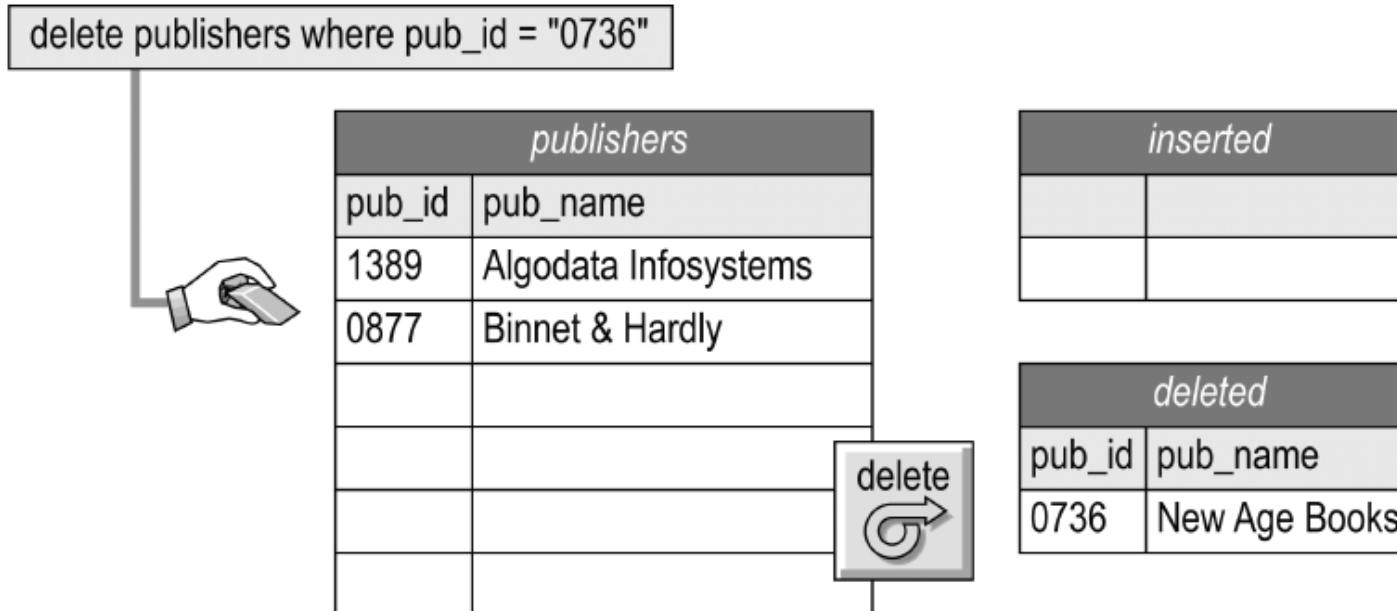
- *inserted* guarda os registros adicionados a tabela.
- *deleted* guarda os registros que foram removidos da tabela.

Evento DML	tabela <b>INSERTED</b> contém	tabela <b>DELETED</b> contém
INSERT	Linhas a serem inseridas	vazio
UPDATE	novas linhas modificadas pelo UPDATE	linhas existentes modificadas pelo UPDATE
DELETE	vazio	Linhas a serem excluídas

# Triggers

## DELETE

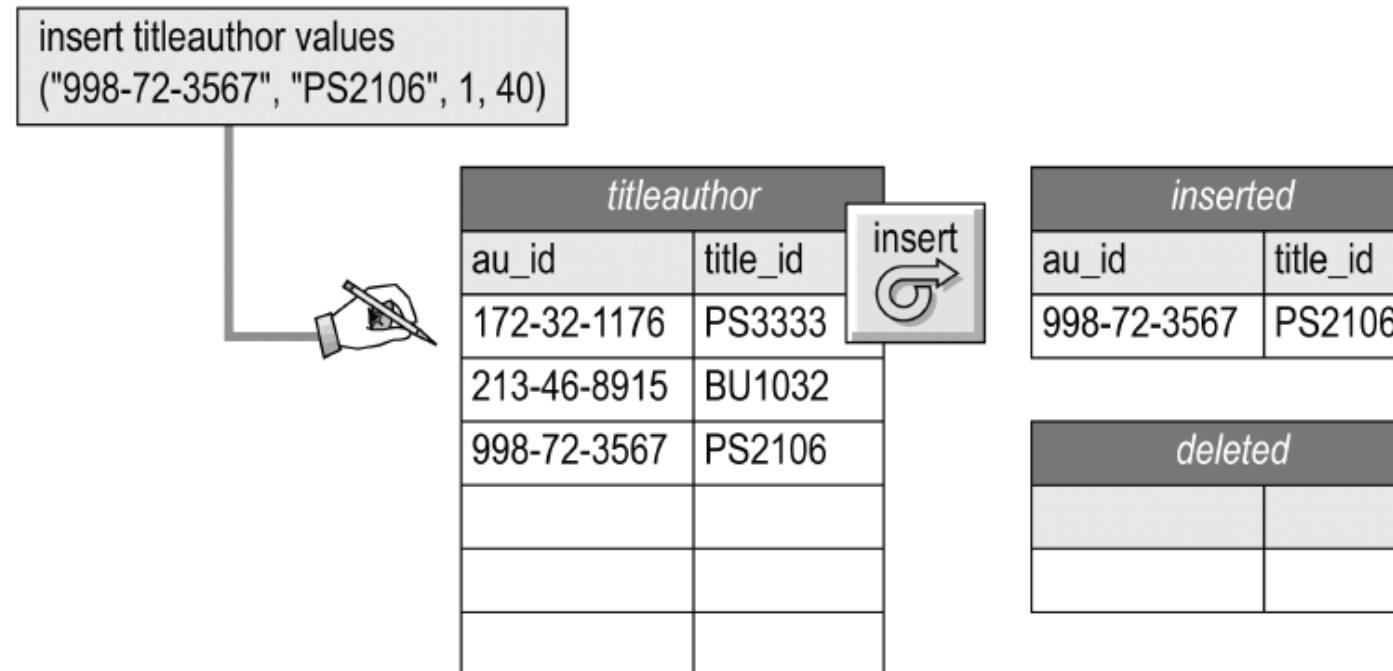
Um comando de *delete* adiciona linhas a tabela *deleted*.



# Triggers

## INSERT

Um comando de *insert* adiciona linhas a tabela *inserted*.



# Triggers

## UPDATE

Um comando de *update* adiciona linhas as tabelas *inserted* e *deleted*.

```
update publishers set pub_id = "9988"  
from publishers where pub_id = "9999"
```



publishers	
pub_id	pub_name
1389	Algodata Infosystems
0877	Binnet & Hardly
9988	Tech Books



inserted	
pub_id	pub_name
9988	Tech Books

deleted	
pub_id	pub_name
9999	Tech Books

# Triggers

## CONSIDERAÇÕES TABELAS INSERTED E DELETED

- Ambas as tabelas *inserted* e *deleted* possuem as mesmas colunas da tabela da trigger.
- Cada trigger pode consultar somente suas tabelas *inserted* e *deleted* (outros processos não podem). Uma trigger não pode consultar as tabelas *inserted* e *deleted* de outras triggers.
- As triggers não podem modificar os dados das tabelas *inserted* e *delete* (as tabelas são somente leitura).

# Triggers

## MELHORES PRÁTICAS

Os itens a seguir devem ser levados em consideração para um eficaz desenvolvimento de triggers:

- *@@ROWCOUNT ou ROWCOUNT\_BIG() no começo do código.*
- IF UPDATE
- RAISERROR ou THROW

# Triggers

## MELHORES PRÁTICAS

```
DROP TRIGGER IF EXISTS Person.reminder;
GO

CREATE TRIGGER reminder ON Person.Address AFTER UPDATE AS
BEGIN
    IF (ROWCOUNT_BIG() = 0)
        RETURN;
    IF ( UPDATE (StateProvinceID) OR UPDATE (PostalCode) )
    BEGIN
        RAISERROR ('Notify Customer Relations', 16, 1);
        --OU
        THROW 50000,N'Notify Customer Relations',1
    END;
END
GO
```

# Triggers

## MELHORES PRÁTICAS - IF UPDATE

- **IF UPDATE** é uma condição que permite uma trigger checar se há uma alteração numa coluna específica (somente INSERT ou UPDATE).
- Ele só pode ser usado com trigger
- Normalmente usado para verificar se os valores em uma coluna de chave primária foram alterados

### SINTAXE

```
if update (column_name) [ {and | or} update (column_name) ] ...
```

# Triggers

## MELHORES PRÁTICAS – RAISERROR E THROW

**RAISERROR** e **THROW** gera uma exceção no código que pode ser capturado pela aplicação para tratamento de erros (transfere a execução para um bloco CATCH de uma construção TRY CATCH).

```
THROW 50000, N'An error occurred', 1;
```

```
Msg 50000, Level 16, State 1, Line 11  
An error occurred
```

```
RAISERROR (N'An error occurred', 16, 1)
```

```
Msg 50000, Level 16, State 1, Line 13  
An error occurred
```

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios gatilhos (triggers)”**

Utilizando os itens vistos em aula:

**PROGRAMAÇÃO T-SQL, STORED PROCEDURE, CURSORES e TRIGGERS.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.

# Functions

**FUNCTIONS ou FUNÇÕES** como funções em linguagens de programação, funções no SQL Server são rotinas que aceitam parâmetros, executam ações (rotinas, cálculos complexos, etc) e retornam um resultado.

Existem 3 tipos de funções:

## Escopo de estudo

- **Escalar:** Retornam um valor único.
- **Table-Valued:** Retornam uma lista de resultados.
- **Sistema:** Não podem ser alteradas, utilizadas para diversas operações (agregação, matemática, string, etc).

# Functions

## DIFERENÇA ENTRE STORED PROCEDURE E FUNCTIONS

- **Funções** podem ser utilizadas em comandos de SELECT, diferentemente de uma **stored procedure**.
- **Stored procedures** aceitam parâmetros de entrada e saída, **funções** somente de entrada.
- **Funções** sempre retornam um valor, **stored procedure** podem ou não retornar um valor.
- Em **stored procedures** é possível trabalhar com controle transacional (begin tran, commit, rollback) em **funções** não é possível.

# Functions

## CRIANDO UMA FUNÇÃO

### SINTAXE

```
create function function_name (@par1 type, @par2 type, ... ) returns int as
begin
    sql_statements

    return @parameter
end
go
```

### EXEMPLO

```
create function fn_idade (@data datetime) returns int as
begin
    declare @idade int

    select @idade = floor(datediff(day, @data, getdate()) / 365.25)

    return @idade
end
go
```

# Functions

## EXCLUINDO UMA FUNÇÃO

### SINTAXE SIMPLIFICADA

```
drop function function_name
```

### EXEMPLO

```
drop function fn_idade
```

### VALIDANDO A EXISTÊNCIA DA FUNCTION ANTES DE EXCLUIR

```
drop function if exists fn_idade  
go
```

# Functions

## CHAMANDO UMA FUNÇÃO

### SINTAXE

```
select schema.function_name(parametros)
[ from table_name ... ]
```

Sempre necessário informar o nome do schema ao chamar uma função escalar.

### EXEMPLO

```
select top 5 cd_paciente,
        nm_paciente,
        dbo.fn_idade(dt_nascimento) as idade
from paciente
```

	cd_paciente	nm_paciente	idade
0		PACIENTE NÃO CADASTRADO	NULL
1		MARILENE	58
2		ALINE	33
3		AMARILDO	58
4		ANA	18

# Exercícios

Resolver a lista de exercícios do arquivo:

**“Exercícios funções (functions)”**

Utilizando os itens vistos em aula:

**PROGRAMAÇÃO T-SQL, CURSORES e FUNCTIONS.**

Procure entender o resultado obtido em cada resposta para melhor entendimento da matéria.