
T-SQL - Triggers

Triggers

- *Triggers* são procedimentos que são executados automaticamente, como resultado de um evento gerado no SGBD
- Os *Triggers* não têm parâmetros de entrada nem saída, nem podem ser executados explicitamente
- No entanto, são semelhantes a procedimentos armazenados, na forma como são tratados pelo SQL Server, pelo que passam pelas mesmas fases
 - Análise sintáctica, resolução e optimização
- O SQL Server suporta *triggers* que disparam em resultado de
 - Instruções DML (*Data Manipulation language*)
 - Instruções DDL (*Data Definition Language*)

Triggers (cont.)

- DML triggers disparam quando são executadas instruções de INSERT, UPDATE, ou DELETE sobre tabelas ou vistas
- Podem ser definidos para dispararem apenas quando ocorre uma das acções (INSERT ou DELETE ou UPDATE) ou com uma qualquer combinação das três
- Note-se que estes *triggers* não disparam em resultado de operações que não são escritas em log (e.g. BULK INSERT)
- Não podem ser definidos sobre tabelas temporárias
- SQL Server não suporta SELECT *triggers*, nem *triggers* ao nível do tuplo

Triggers (cont.)

- Se uma restrição de integridade de uma tabela falhar, qualquer *trigger* definido sobre essa tabela não será executado
- Se existirem múltiplos *triggers* definidos sobre uma tabela, estes executam-se sequencialmente
- No entanto, é possível definir (parcialmente) a ordem de execução, através do procedimento `sp_settriggerorder`. Este tópico será abordado mais adiante

Triggers (cont.)

- Os *triggers* podem ser de dois tipos:
- **AFTER**
 - São executados após a execução da instrução que os despoletou, não por cada tuplo manipulado
 - Podem ser utilizados apenas sobre tabelas
- **INSTEAD OF**
 - São executados em vez da instrução que os despoletou
 - Podem ser utilizados sobre tabelas e vistas
- Independentemente do tipo do *trigger*, a instrução que os despoletou só é realmente concluída, após a execução do *trigger*
- Isto porque os *triggers* fazem parte da transacção que os originou

AFTER DML Triggers

Nome da tabela a que o *trigger* se destina

Nome da esquema (opcional) e o nome do *trigger*

Opções relativas a contexto de execução e encriptação

```
CREATE TRIGGER [ schema_name . ]trigger_name
ON table
[ WITH <dml_trigger_option> [ ...,n ] ]
AFTER
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ NOT FOR REPLICATION ]
AS { sql_statement [ ...n ] }
```

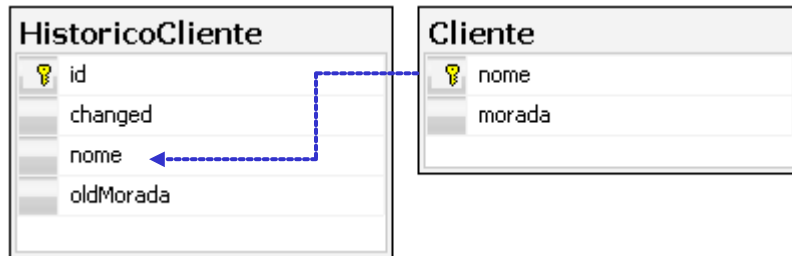
Acção (ou acções) que despoletam o *trigger*. Pelo menos uma acção tem de ser especificada

Código T-SQL que é executado quando o *trigger* dispara

Quando especificado, indica que o *trigger* não deve correr quando acção resulta de uma replicação

AFTER DML Triggers - Exemplo

Pretende-se guardar em histórico todas as alterações nos dados de cliente



Como aceder aos valores depois de alterados?

Através da tabela especial
DELETED

```
CREATE TRIGGER trgHistoricoCli
on Cliente
AFTER UPDATE,DELETE
as
INSERT INTO HistoricoCliente([Nome],[oldMorada])
SELECT * FROM DELETED
```

Tabelas INSERTED e DELETED

- Nos *triggers* tem-se acesso a duas versões dos dados, **antes** e **depois** da alteração, através de duas tabelas especiais
 - INSERTED
 - DELETED
- Para *triggers* AFTER, a figura seguinte resume conteúdo de cada uma, após a execução de cada comando

Operação	INSERTED contém	DELETED contém
INSERT	Os tuplos inseridos	
UPDATE	Os tuplos actualizados (já com os novos valores)	Os tuplos que foram actualizados mas com os valores antes da actualização
DELETE		Os tuplos removidos

Tabelas INSERTED e DELETED (cont.)

- Estas tabelas têm o mesmo esquema (estrutura) da tabela onde o *trigger* foi definido
- No entanto, não são indexadas
- Por isso, deve ser tida em consideração as operações que são feitas sobre estas tabelas
- Estas tabelas residem na base de dado *tempdb* do SQL Server
- Utilizam a tecnologia de *row versioning*

AFTER DML *Triggers* - Exemplo II

Pretende-se adicionar funcionalidade ao *trigger* anterior. Além de guardar em histórico todas as alterações sobre a tabela *cliente*, deve negar qualquer tipo de alteração à chave da tabela

Como detectar que foi feita um alteração sobre uma coluna?



Funções **UPDATE** e **COLUMNS_UPDATED**

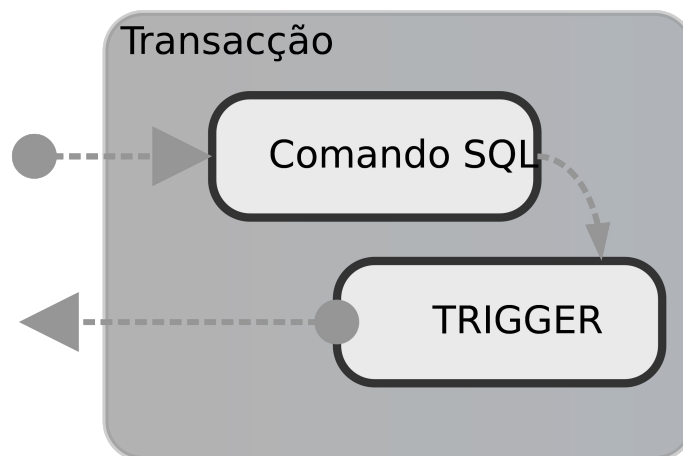
Como desfazer as acções do *trigger*



Manipular a transacção dentro da qual o *trigger* corre

Triggers e transacções

- Quando um *trigger* corre, a execução é sempre feita dentro de uma transacção (implícita ou explícita)
- Essa transacção é a mesma que envolve o comando que fez despoletar o *trigger*



- Assim, o sucesso do comando depende do sucesso da execução do *trigger*

AFTER DML Triggers – Exemplo II (cont.)

```
CREATE TRIGGER trgHistoricoCli
on Cliente
AFTER UPDATE,DELETE
as
IF UPDATE(Nome)
BEGIN
    RAISERROR('Não é possível alterar a coluna Nome.
    Consulte o administrador para mais
    esclarecimentos.',16,1)
    ROLLBACK
    RETURN
END
INSERT INTO HistoricoCliente([Nome],[oldMorada]) SELECT
    * FROM DELETED
```

GO

- Esta função retorna um booleano que indica se foi feito um INSERT ou UPDATE sobre a coluna de uma tabela ou vista
- É retornado TRUE, independentemente do INSERT ou UPDATE suceder
- Alterações sobre várias colunas podem ser verificadas utilizando a função COLUMNS_UPDATE. Esta retorna na forma de uma máscara de bits as colunas alteradas.

AFTER DML Triggers – Exemplo II (cont.)

```
CREATE TRIGGER trgHistoricoCli  
on Cliente  
AFTER UPDATE,DELETE  
as  
IF UPDATE(Nome)  
BEGIN
```

```
    RAISERROR('Não é possível alterar a coluna Nome.  
    Consulte o administrador para mais  
    esclarecimentos.',16,1)
```

```
    ROLLBACK  
    RETURN
```

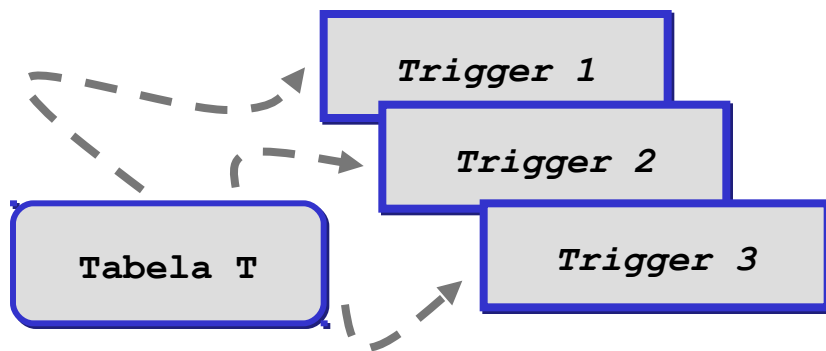
```
END  
INSERT INTO HistoricoCliente([Nome],[oldMorada]) SELECT  
    * FROM DELETED
```

GO

- Evocar o ROLLBACK termina a transacção corrente
- São desfeitas as acções do comando o despoletou o *trigger*, assim como o trabalho efectuado pelo *trigger* até esse momento

Controlar a ordem de execução do trigger

- O SQL Server permite definir qual o *trigger* que corre em primeiro lugar e em último, através do procedimento `sp_settriggerorder`
- A ordem dos *triggers* intermédios é indefinida



```
CREATE TRIGGER trg_execT1
```

```
CREATE TRIGGER trg_execT2
```

```
CREATE TRIGGER trg_execT3  
ON T3 AFTER INSERT,UPDATE,DELETE  
AS
```

```
declare @type as nvarchar(10)
```

```
...//Detectar a acção
```

```
PRINT 'T1 TRIGGER FOR ' + @TYPE
```

```
DELETE FROM dbo.T3
```

```
T1 TRIGGER FOR DELETE  
T2 TRIGGER FOR DELETE  
T3 TRIGGER FOR DELETE
```

Controlar a ordem de execução do trigger (cont.)

```
sp_settriggerorder
    [ @triggername = ] '[ triggerschema. ]triggername'
, [ @order = ] 'value'
, [ @stmttype = ] 'statement_type'
[ , [ @namespace = ] { 'DATABASE' | 'SERVER' } ]
```

- **@triggername** – Indica o nome do *trigger* a ordenar. *Triggerschema* apenas é utilizado para triggers DML.
- **@order** – Indica a nova ordem de execução do *trigger*. Pode tomar os valores FIRST, LAST ou NONE. Neste último caso a ordem é aleatória.
- **@stmttype** – Indica para que tipo de comando a ordenação está a ser definida. Pode tomar qualquer dos valores possíveis na definição dos *triggers* (DML e DDL)
- **@namespace** – Para triggers DDL apenas.

Controlar a ordem de execução do trigger (cont.)

```
exec sp_settriggerorder @triggername = 'trg_execT3',  
                        @order = 'FIRST', @stmttype = 'DELETE'  
exec sp_settriggerorder @triggername = 'trg_execT1',  
                        @order = 'LAST', @stmttype = 'DELETE'
```

Update dbo.T3 SET datacol=2

**T1 TRIGGER FOR UPDATE
T2 TRIGGER FOR UPDATE
T3 TRIGGER FOR UPDATE**

DELETE FROM dbo.T3

**T3 TRIGGER FOR DELETE
T2 TRIGGER FOR DELETE
T1 TRIGGER FOR DELETE**

```
exec sp_settriggerorder @triggername = 'trg_execT3',  
                        @order = 'NONE', @stmttype = 'DELETE'  
exec sp_settriggerorder @triggername = 'trg_execT1',  
                        @order = 'NONE', @stmttype = 'DELETE'
```

DELETE FROM dbo.T3

**T1 TRIGGER FOR DELETE
T2 TRIGGER FOR DELETE
T3 TRIGGER FOR DELETE**

Controlo de Aninhamento e recursão

- O SQL Server suporta aninhamento e *triggers* recursivos
- O aninhamento resulta da execução de um comando dentro de um *trigger*, que por sua vez, provoca a execução de outro *trigger*, e por aí em diante
- SQL limita o aninhamento a 32 níveis
- É possível controlar se existe aninhamento, através da opção de configuração do servidor *nested_triggers*
- Esta opção encontra-se activada por omissão

```
USE master
GO
EXEC sp_configure 'nested triggers', 0 /* 0 -Disable, 1-Enable*/
RECONFIGURE WITH OVERRIDE
GO
```

Controlo de Aninhamento e recursão (cont.)

- A recursividade resulta quando os comandos de um trigger provocam outra execução dele próprio
- O controlo da recursão é feito através da opção das base de dados `RECURSIVE_TRIGGERS`

```
ALTER DATABASE umaDB  
SET RECURSIVE_TRIGGERS ON
```

Activar / Desactivar *Triggers*

- É possível activar ou desactivar um determinado *trigger*

```
ENABLE TRIGGER [ schema_name . ]  
    trigger_name  
ON { object_name | DATABASE | SERVER  
    }
```

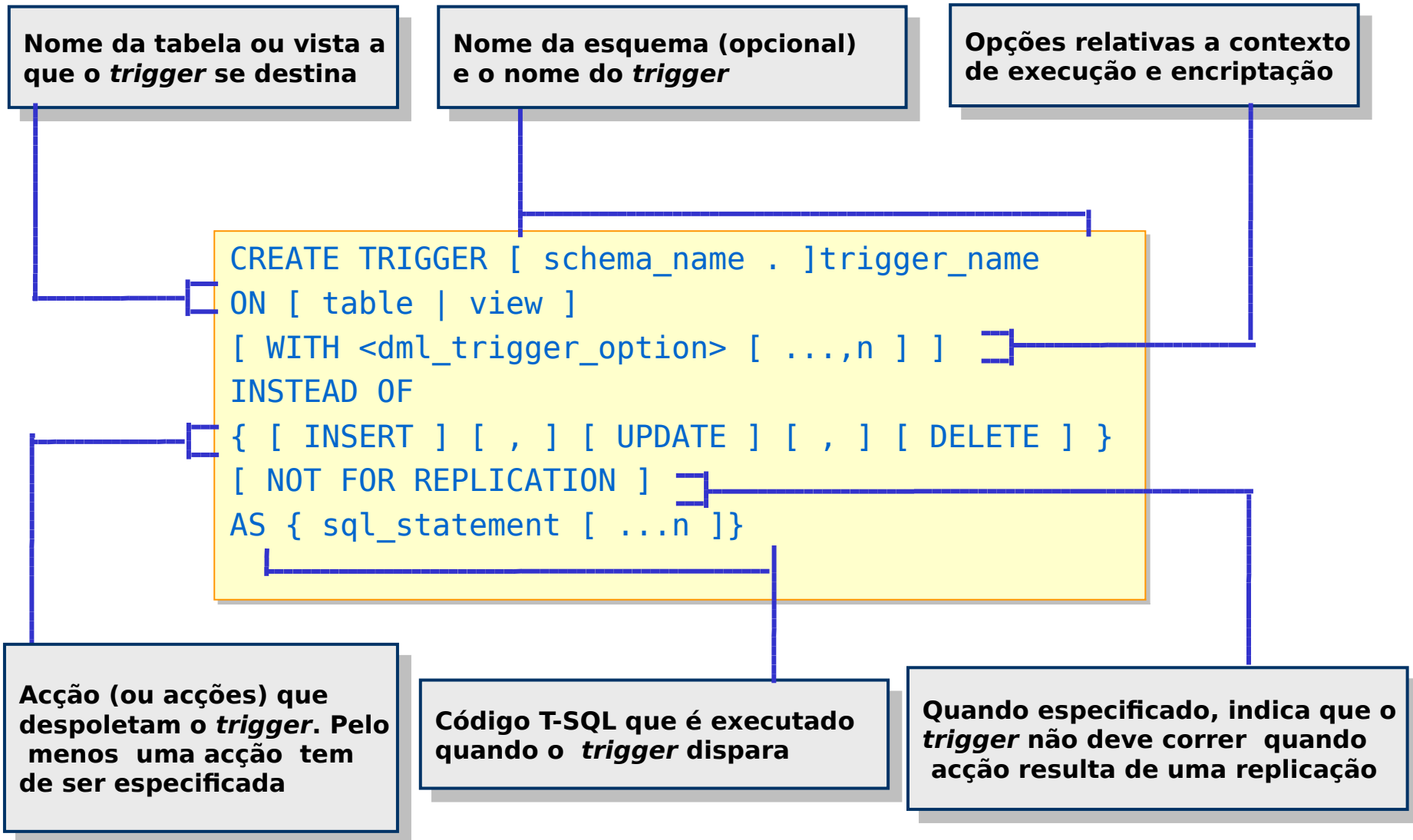
```
DISABLE TRIGGER [ schema . ]  
    trigger_name  
ON { object_name | DATABASE | SERVER  
    }
```

- No entanto, não é possível fazê-lo, directamente, por tipo de comando
- Essa acção é possível, recorrendo a código e uma tabela auxiliar, que controla o tipo de acções a efectuar

Identificar o número de tuplos afectados

- Os *triggers* disparam por comando, não por tuplo afectado
 - Podem disparar quando zero, um ou mais tuplos são afectados
- Sabendo o número de tuplos afectados, é possível desenvolver código por forma a ter o máximo desempenho em cada uma das situações
- A função de @@rowcount determinar esse número, uma vez que retorna o número de tuplos afectados pela última instrução SQL
- ```
CREATE TRIGGER ...
AS
DECLARE @rc AS INT;
SET @rc = @@rowcount;
...
```
- Note-se que a afectação de uma variável local como primeira instrução é imperativa, uma vez que os comandos subsequentes podem alterar o valor retornado por esta função

# INSTEAD OF *Triggers*



---

## INSTEAD OF *Triggers* (cont.)

---

- Este tipo de *triggers* corre em vez do comando que os fez disparar
- Uma vez que as alterações não chegam a ser efectuadas, se não forem efectuados os comandos SQL necessários para efectuar as acções pretendidas, as alterações não serão efectuadas

```
CREATE TRIGGER JOKE
ON table1
INSTEAD OF INSERT, UPDATE, DELETE
AS
DECLARE @dummy AS INT;
```

O *trigger* anterior evita que sejam feitas inserções, alteração e remoções , na tabela!

---

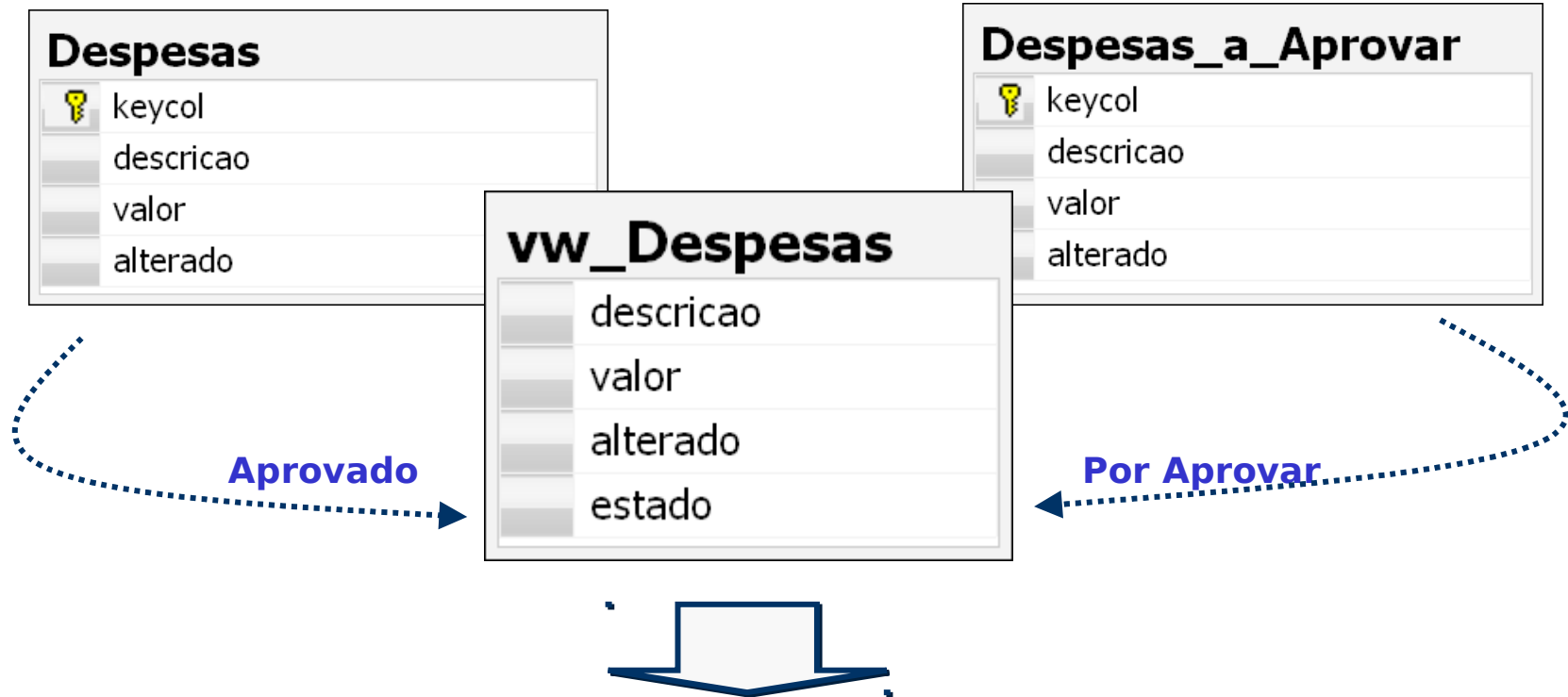
## INSTEAD OF *Triggers* (cont.)

---

- OS triggers INSTEAD OF podem ser criados sobre tabelas, mas também sobre vistas
- É sobre as vistas que são normalmente utilizados, por forma a permitir modificações em vistas não actualizáveis
- Por objecto apenas pode existir um *trigger* INSTEAD OF, por cada comando
- Este tipo de *triggers* não disparam recursivamente
- Ao contrário dos *triggers* AFTER, estes disparam antes das restrições de integridade serem verificadas
- As tabelas INSERTED e DELETED contêm os tuplos a serem alterados

# INSTEAD OF *Triggers* – Exemplo I

Pretende-se permitir a inserção sobre uma vista, que apresenta as despesas previstas, aprovadas ou não.




Esta vista não é actualizável, pelo que a solução passará por criar um *trigger* *INSTEAD OF* para inserções




# INSTEAD OF *Triggers* - Exemplo I (cont.)

```
CREATE TRIGGER trg_update_Despesas
ON vw_Despesas
INSTEAD OF INSERT
AS
SET NOCOUNT ON
INSERT INTO dbo.Despesas_a_Aprovar(descricao,valor)
 SELECT descricao,valor FROM inserted
 WHERE descricao NOT IN (SELECT descricao FROM dbo.Despesas)
```



```
INSERT INTO vw_Despesas(descricao,valor) VALUES('Toner', €75);
INSERT INTO vw_Despesas(descricao,valor)
 SELECT 'Toner', €80
UNION
 SELECT 'Livro sobre .NET', €40
```



```
SELECT * FROM vw_Despesas
```



|                         |              |                                |                    |
|-------------------------|--------------|--------------------------------|--------------------|
| <b>Toner</b>            | <b>75,00</b> | <b>2007-09-05 12:20:17.573</b> | <b>Por aprovar</b> |
| <b>Livro sobre .NET</b> | <b>40,00</b> | <b>2007-09-05 12:20:17.613</b> | <b>Por aprovar</b> |

---

## INSTEAD OF *Triggers* – Exemplo II

---

Pretende-se inserir vários tuplos numa tabela, sobre a qual já existe um *trigger*, mas que só suporta actualizações sobre um único tuplo. Não é possível alterar o código desse *trigger*

- Uma possível solução passa por criar um *trigger* INSTEAD OF, que transforma as alterações sobre vários tuplos numa alteração tuplo a tuplo
- Note-se que aqui não estão em jogo questões de desempenho, mas a expansibilidade de uma solução onde, por vezes, é impossível alterar o código já feito
- Por exemplo, se os *triggers* existentes estiverem cifrados

# INSTEAD OF *Triggers* – Exemplo II (cont.)

```
CREATE TRIGGER trg_T3 ON T3 AFTER INSERT
```

```
AS
```

```
DECLARE @msg AS VARCHAR(100);
```

```
SET @msg = 'Key: '
```

```
+ CAST((SELECT keycol FROM inserted) AS VARCHAR(10)) + ' inserted.';
```

```
PRINT @msg;
```

```
GO
```

**T3**



keycol

datacol

**Simulação de um *trigger* que só funciona quando um tuplo é alterado. Admite-se que este é o *trigger* que não podia ser modificado**

```
CREATE TRIGGER trg_T3_perrow ON dbo.T3 INSTEAD OF INSERT
```

```
AS
```

```
DECLARE @rc AS INT;
```

```
SET @rc = @@rowcount;
```

```
IF @rc = 0 RETURN;
```

**Determina-se o número de tuplos a afectar. Se são zero, não se toma nenhuma acção**

```
SET NOCOUNT ON
```

```
IF @rc = 1
```

```
INSERT INTO dbo.T3 SELECT * FROM inserted;
```

```
ELSE
```

```
...
```

**Se é apenas um, colocasse a informação que está da tabela *inserted* na tabela T3**

# INSTEAD OF *Triggers* – Exemplo II (cont.)

```
...
BEGIN
 DECLARE @keycol AS INT, @datacol AS INT;

 DECLARE Cinserted CURSOR FAST_FORWARD FOR
 SELECT keycol, datacol FROM inserted;

 OPEN Cinserted;

 FETCH NEXT FROM Cinserted INTO @keycol, @datacol;
 WHILE @@fetch_status = 0
 BEGIN
 INSERT INTO dbo.T3(keycol, datacol)
 VALUES(@keycol, @datacol);

 FETCH NEXT FROM Cinserted INTO @keycol, @datacol;
 END

 CLOSE Cinserted;
 DEALLOCATE Cinserted;
END
GO
```

**T3**

|                                                                                     |         |
|-------------------------------------------------------------------------------------|---------|
|  | keycol  |
|                                                                                     | datacol |

**Caso sejam vários os tuplos, utiliza-se um cursor para iterar sobre a tabela inserted**

**Para cada tuplo, efectua-se a inserção**

# Criação de colunas do tipo XML

-- Tabela com coluna do tipo XML

```
CREATE TABLE dbo.Book (
 BookID int IDENTITY(1,1) PRIMARY KEY,
 ISBNBR char(10) NOT NULL,
 BookNM varchar(250) NOT NULL,
 AuthorID int NOT NULL,
 ChapterDESC XML NULL
)
```

-- Variável do tipo XML

```
DECLARE @Book XML
```

**A tabela pode ter várias colunas XML.**

**Limitações:**

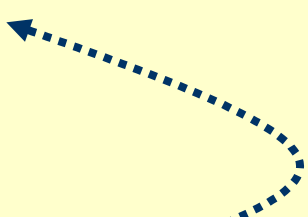
- PRIMARY KEY, FOREIGN KEY, UNIQUE, COLLATE

**Permite:**

- NULL | NOT NULL, DEFAULT, CHECK

# Inserção de dados numa coluna do tipo XML

```
INSERT dbo.Book (ISBNNBR, BookNM, AuthorID, ChapterDESC)
VALUES ('570X000000', 'SQL Server 2005 T-SQL Recipes', 55,
CAST (
 '<Book name="SQL Server 2005 T-SQL Recipes">
 <Chapters>
 <Chapter id="1"> SELECT </Chapter>
 <Chapter id="2"> INSERT,UPDATE,DELETE </Chapter>
 <Chapter id="3"> Transactions, Locking, Blocking, and
Deadlocking </Chapter>
 <Chapter id="4"> Tables </Chapter>
 <Chapter id="5"> Indexes </Chapter>
 <Chapter id="6"> Full-text search </Chapter>
 </Chapters>
 </Book>' AS XML
))
```



**O CAST faz a validação sintáctica do XML**

# Limitações do tipo de dados XML

- O tipo de dados XML não é comparável
- Não pode ser usado nas cláusulas `GROUP BY` e `ORDER BY`
- Risco na conversão para para *strings*
- Solução:
  - Métodos XQuery

| Método                                    | Descrição                                                                                                     |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>exist(xquery_text)</code>           | Retorna “1” ou “0” função da existência de um ou mais nós que satisfaçam a expressão <code>xquery_text</code> |
| <code>value(xquery_text, sql_Type)</code> | Retorna um único valor do tipo de dados <code>sql_Type</code> baseado na expressão <code>xquery_text</code>   |
| <code>query(xquery_text)</code>           | Retorna um resultado do tipo XML baseado na expressão <code>xquery_text</code>                                |
| <code>nodes(xquery_text)</code>           | Converte em dados relacionais                                                                                 |

---

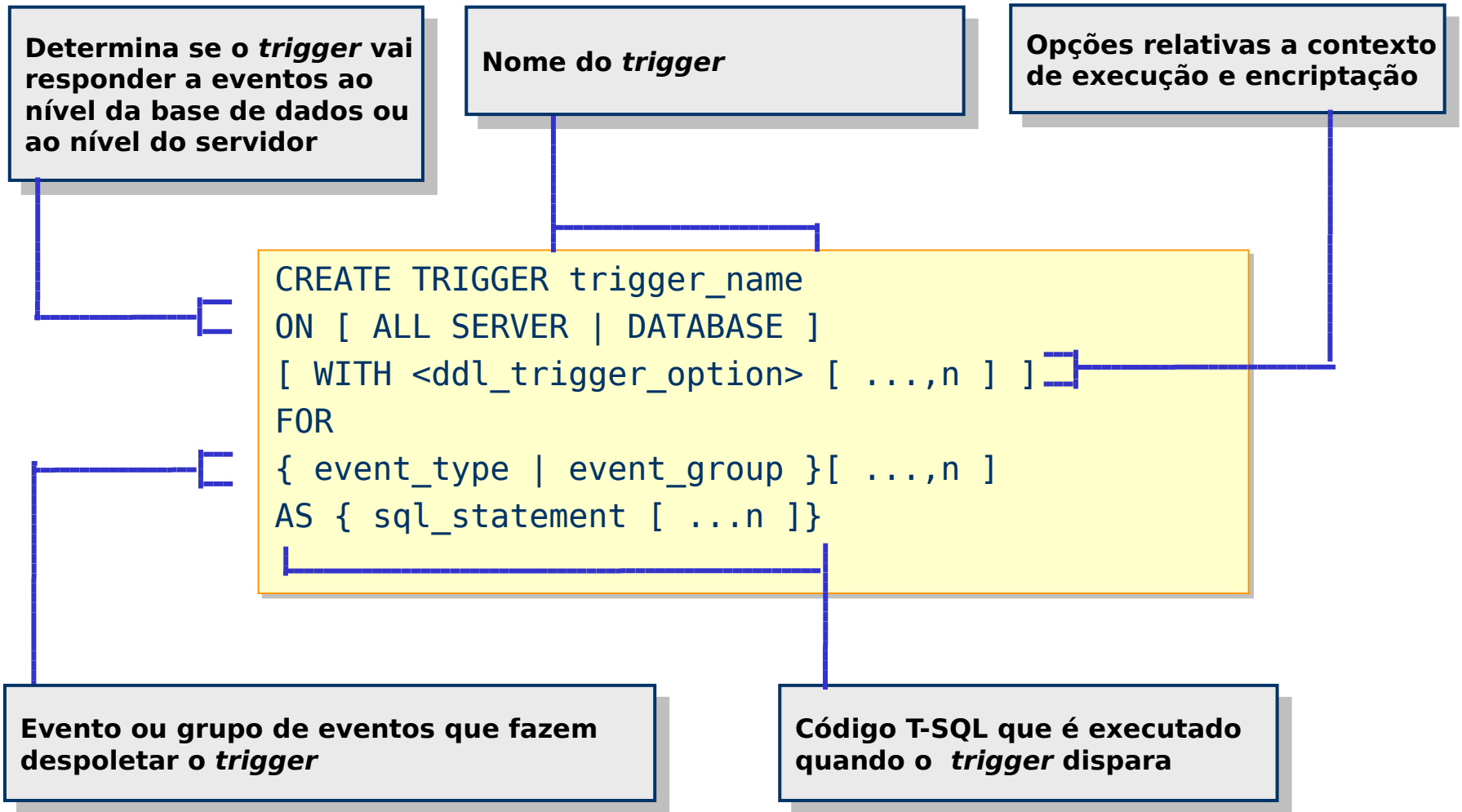
# DDL Triggers

---

- O SQL Server suporta *triggers* que respondem a comandos DDL (*Data Definition Language*)
- A reacção pode ser a comandos específicos, e.g. CREATE, ou a grupos de eventos, e.g. DDL\_TYPE\_EVENTS (que engloba CREATE e DROP)
- São apenas suportados *triggers* AFTER
- Podem ser definidos ao nível do servidor
  - podem responder a criação de bases de dados ou *logins*, entre outros
  - Os triggers são guardados como objectos da base de dados master
- Podem ser definidos ao nível da base de dados
  - Respondem apenas a eventos específicos da base de dados em questão
  - São guardados como objectos da base de dados



# DDL Triggers (cont.)



---

## DDL *Triggers* - *eventdata*

---

- Dentro do *trigger* é possível obter informações sobre o evento que o despoletou
- Essa informação é obtida através da função `eventdata`
- O valor retornado é XML
- Para a sua manipulação é necessário utilizar *XQuery*
- Para extrair um determinado atributo do valor XML, utiliza-se a seguinte sintaxe:
  - `xml_value.query('data(//attribute_name)')`
- Onde `xml_value` é a variável onde se guardou o retorno da função `eventdata` e `attribute_name` é o nome do atributo a extrair

# DDL Triggers - eventdata (cont.)

```
<EVENT_INSTANCE>
 <EventType>CREATE_TABLE</EventType>
 <PostTime>2007-09-05T14:51:53.333</PostTime>
 <SPID>53</SPID>
 <ServerName>OPHIS</ServerName>
 <LoginName>OPHIS\datia</LoginName>
 <UserName>dbo</UserName>
 <DatabaseName>TRIGGERSDB</DatabaseName>
 <SchemaName>dbo</SchemaName>
 <ObjectName>T3</ObjectName>
 <ObjectType>TABLE</ObjectType>
 <TSQLCommand>
 <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON"
ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON" ENCRYPTED="FALSE"/>
 <CommandText>
 CREATE TABLE T3(keycol int not null primary key,datacol int)
 </CommandText>
 </TSQLCommand>
</EVENT_INSTANCE>
```

**T3**



keycol

datacol

# DDL Triggers – Exemplo I

Pretende-se garantir todas as tabelas criadas têm uma chave primária definida

```
CREATE TRIGGER trg_enforcePK
ON DATABASE
FOR CREATE_TABLE
AS
```

**Obter o esquema e o nome da tabela, utilizando XQuery**

```
DECLARE @eventdata AS XML, @tablename as NVARCHAR(128), @msg NVARCHAR(256);
```

```
SET @eventdata = eventdata();
```

```
SET @tablename =
```

```
QUOTENAME(CAST(@eventdata.query('data(//SchemaName)') as sysname))
+ N'.' +
QUOTENAME(CAST(@eventdata.query('data(//ObjectName)') as sysname));
```

```
IF(COALESCE(OBJECTPROPERTY(OBJECT_ID(@tablename), 'TableHasPrimaryKey'),0) = 0)
BEGIN
```

```
 SET @msg = N'A tabela ' + @tablename + ' não têm uma chave primária definida.'
```

```
 + CHAR(10) + N'A tabela não foi criada';
```

```
 RAISERROR(@msg, 16, 1);
```

```
 ROLLBACK;
```

```
 RETURN;
```

```
END
```

# DDL Triggers – Exemplo II

Pretende-se que todas as alterações a tabelas e vistas sejam registadas, com a informação do utilizador que as efectuou

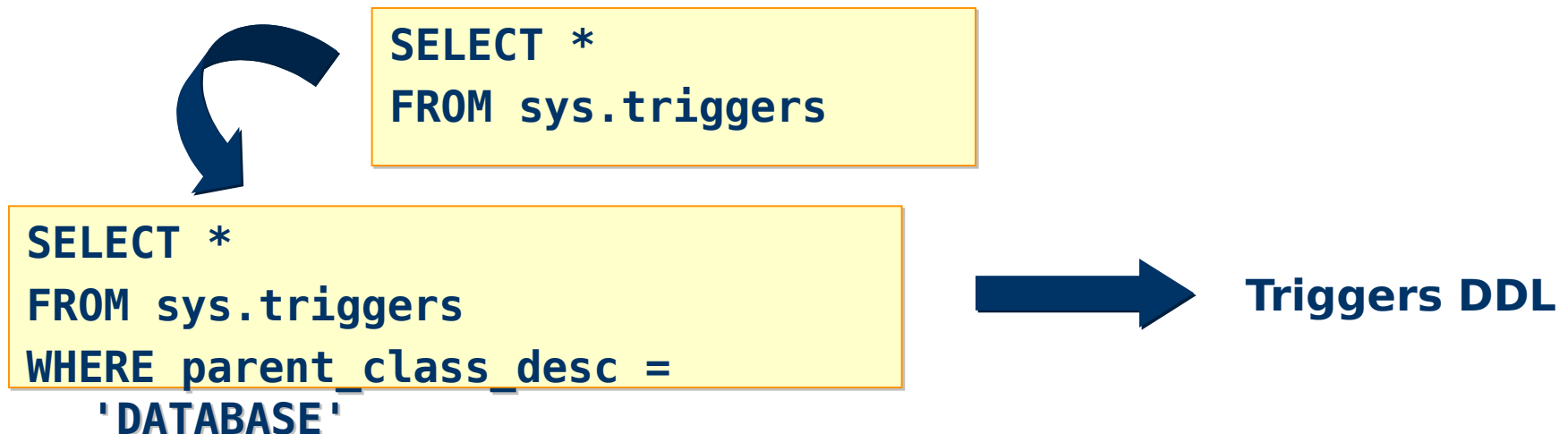
```
CREATE TABLE dbo.ChangeLog
(
 EventData xml NOT NULL,
 AttemptDate datetime NOT NULL DEFAULT GETDATE(),
 DBUser char(50) NOT NULL
)
```

**Obter o esquema e o nome da tabela, utilizando XQuery**

```
CREATE TRIGGER trg_audit
ON DATABASE
FOR DDL_TABLE_VIEW_EVENTS
AS
 SET NOCOUNT ON
 INSERT dbo.ChangeLog
 (EventData, DBUser)
 VALUES (EVENTDATA(), USER)
```

# Trigger Metadata

- A informação relacionadas com os *triggers* não está concentrada apenas num local, mas dispersa, consoante a natureza do *trigger*
- Para *triggers* ao nível da base de dados, a meta-informação está localizada na tabela de sistema `sys.triggers`. Para o caso de serem triggers DML, essa informação também pode ser obtida através da tabela `sys.objects`.



## Triggers Metadata (cont.)


- Para triggers DDL ao nível do servidor, essa informação esta presente na tabela sys.server\_triggers e sys.server\_triggers\_events
  - A primeira tem a informação referente ao *trigger*
  - A segunda, a informação sobre os eventos que levam o *trigger* a disparar e a sua ordem

```
CREATE TRIGGER trg_audit_ddl_logins ON ALL SERVER
FOR DDL_LOGIN_EVENTS
AS
Print 'DDL_LOGIN_EVENTS'
```

Trigger que corre nas acções de  
**DROP\_LOGIN, CREATE\_LOGIN,  
ALTER\_LOGIN**

## Triggers Metadata (cont.)

```
SELECT name, s.type_desc SQL_or_CLR,
is_disabled, e.type_desc FiringEvents, e.is_First, e.is_Last
FROM sys.server_triggers s
INNER JOIN sys.server_trigger_events e ON
s.object_id = e.object_id
```



|                      |             |   |              |   |   |
|----------------------|-------------|---|--------------|---|---|
| trg_audit_ddl_logins | SQL_TRIGGER | 0 | CREATE_LOGIN | 0 | 0 |
| trg_audit_ddl_logins | SQL_TRIGGER | 0 | ALTER_LOGIN  | 0 | 0 |
| trg_audit_ddl_logins | SQL_TRIGGER | 0 | DROP_LOGIN   | 0 | 0 |



DDL\_LOGIN\_EVENTS



# Triggers Metadata (cont.)

```
SELECT t.name, m.Definition
FROM sys.server_triggers AS t
INNER JOIN sys.server_sql_modules m ON
t.object_id = m.object_id
```

Código SQL. Quando é NULL indica que o código está cifrado.

trg\_audit\_ddl\_logins

CREATE TRIGGER trg\_audit\_ddl\_logins ON ...

sys.server\_triggers  
sys.server\_sql\_modules

Contexto: *Server* !

sys.triggers  
sys.sql\_modules

Contexto: *Database* !

---

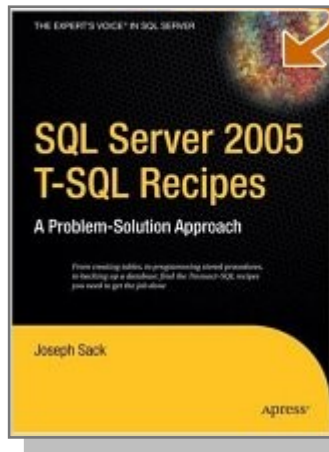
# Bibliografia

---



## **Itzik Ben-Gan**

“Inside SQL Server 2005: T-SQL Programming ” ,  
Microsoft Press, 2006



## **Joseph Sack**

“SQL Server 2005 T-SQL Recipes: A Problem Solution  
Approach”, Apress, 2005