Триггеры DML T-SQL

Для поддержания согласованности и точности данных используются *декларативные* и *процедурные* методы. Триггеры представляют собой особый вид хранимых процедур, привязанных к таблицам и представлениям. Они позволяют реализовать в базе данных сложные **процедурные** методы поддержания целостности данных. События при модификации данных вызывают автоматическое срабатывание триггеров. Как и в случае хранимых процедур, глубина вложенности триггеров достигает 32 уровней, также возможно рекурсивное срабатывание триггеров.
Прежде чем реализовывать триггер, следует выяснить, нельзя ли получить аналогичные результаты с использованием ограничений или правил. Для уникальной идентификации строк табличных данных используют целостность сущностей (ограничения **primary key** и **unique**). Доменная целостность служит для определения значений по умолчанию (определения **default**) и ограничения диапазона значений, разрешенных для ввода в данное поле (ограничения **check** и ссылочные ограничения). Ссылочная целостность используется для реализации логических связей между таблицами (ограничения **foreign key** и **check**). Если значение обязательного поля не задано в операторе INSERT, то оно определяется с помощью определения **default**. Лучше применять ограничения, чем триггеры и правила.

Триггеры применяются в следующих случаях:

- если использование методов декларативной целостности данных не отвечает функциональным потребностям приложения. Например, для изменения числового значения в таблице при удалении записи из этой же таблицы следует создать триггер;
- если необходимо каскадное изменение через связанные таблицы в базе данных. Чтобы обновить или удалить данные в столбцах с ограничением **foreign key**, вместо пользовательского триггера следует применять ограничения каскадной ссылочной целостности;
- если база данных денормализована и требуется способ автоматизированного обновления избыточных данных в нескольких таблицах;
- если необходимо сверить значение в одной таблице с неидентичным значением в другой таблице;
- если требуется вывод пользовательских сообщений и сложная обработка ошибок.

События, вызывающие срабатывание триггеров DML (типы триггеров)

Автоматическое срабатывание триггера вызывают три события: INSERT, UPDATE и DELETE, которые происходят в таблице или представлении. Триггеры нельзя запустить вручную. В синтаксисе триггеров перед фрагментом программы, уникально определяющим выполняемую триггером задачу, всегда определено одно или несколько таких событий. Один триггер разрешается запрограммировать для реакции на несколько событий, поэтому несложно создать процедуру, которая одновременно является триггером на обновление и добавление. Порядок этих событий в определении триггера является несущественным.

Классы триггеров

В SQL Server существуют два класса триггеров DML:

- INSTEAD OF. Триггеры этого класса выполняются в обход действий, вызывавших их срабатывание, заменяя эти действия. Например, обновление таблицы, в которой есть триггер INSTEAD OF, вызовет срабатывание этого триггера. В результате вместо оператора обновления выполняется код триггера. Это позволяет размешать в триггере сложные операторы обработки, которые дополняют действия оператора, модифицирующего таблицу.
- AFTER. Триггеры этого класса исполняются после действия, вызвавшего срабатывание триггера. Они считаются классом триггеров по умолчанию.

Между триггерами этих двух классов существует ряд важных отличий, показанных в таблице.

Характеристика	**INSTEAD OF-триггер**	**AFTER-триггер**
Объект, к которому можно привязать триггер	Таблица или представление. Триггеры, привязанные к представлению, расширяют список типов обновления, которые может поддерживать представление	Таблица. AFTER-триггеры срабатывают при модификации данных в таблице в ответ на модификацию представления
Допустимое число триггеров	В таблице или представлении допускается не больше одного триггера в расчете на одно действие. Можно определять представления для других представлений, каждое	К таблице можно привязать несколько AFTER-триггеров

	со своим собственным INSTEAD OF-триггером	
Порядок исполнения	Поскольку в таблице или представлении допускается не больше одного такого триггера в расчете на одно событие, порядок не имеет смысла	Можно определять триггеры, срабатывающие первым и последним. Для этого служит системная хранимая процедура sp_settriggerorder. Порядок срабатывания других триггеров, привязанных к таблице, случаен

К таблице разрешено привязывать триггеры обоих классов. Если в таблице определены ограничения и триггеры обоих классов, то первым из них срабатывает триггер INSTEAD OF, затем обрабатываются ограничения и последним срабатывает AFTER-триггер. При нарушении ограничения выполняется откат действий INSTEAD OF-триггера. Если нарушаются ограничения или происходят какие-либо другие события, не позволяющие модифицировать таблицу, AFTER-триггер не исполняется.

Создание триггеров DML

CREATE TRIGGER *имя_триггера*
ON *имя_таблицы_или_представления*
[WITH ENCRYPTION]
класс_триггера тип(ы)_триггера
[WITH APPEND]
[NOT FOR REPLICATION]
AS *sql_инструкции*

В квадратных скобках указаны опции триггера, которые в данной теме не рассматриваются.

Пояснения к инструкции CREATE TRIGGER:

1. Триггеры не допускают указания имени базы данных в виде префикса имени объекта. Поэтому перед созданием триггера необходимо выбрать нужную базу данных с помощью конструкции USE *имя_базы_данных* и ключевого слова GO. Ключевое слово GO требуется, поскольку оператор CREATE TRIGGER должен быть первым в пакете. Право на создание триггеров по умолчанию принадлежит владельцу таблицы.
2. Триггер необходимо привязать к таблице или представлению. Любой триггер можно привязать только к одной таблице или представлению. Чтобы привязать к другой таблице триггер, выполняющий ту же самую задачу, следует создать новый триггер с другим именем, но с той же самой функциональностью и привязать его к другой таблице. AFTER-триггеры (этот класс задан по умолчанию) разрешено привязывать только к таблицам, а триггеры класса INSTEAD OF – как к таблицам, так и к представлениям.
3. При создании триггера следует задать тип события, вызывающего его срабатывание: INSERT, UPDATE и DELETE. Один и тот же триггер может сработать на одно, два или все три события. Если необходимо, чтобы он срабатывал на все события, то после конструкций FOR, AFTER или INSTEAD OF следует поместить все три ключевых слова: INSERT, UPDATE и DELETE в любом порядке.
4. Конструкция AS и следующие за ней команды языка T-SQL определяют задачу, которую будет выполнять триггер.

Пример простого триггера AFTER на событие UPDATE

```
USE dbSPJ
GO
IF OBJECT_ID (N'AfterApdateSPJ','TR') IS NOT NULL
DROP TRIGGER AfterApdateSPJ
GO
CREATE TRIGGER AfterApdateSPJ
ON SPJ
AFTER UPDATE
AS
BEGIN
RAISERROR(N'Произошло обновление в таблице поставок',1,1)
END
GO
```

```
USE dbSPJ
GO
UPDATE SPJ
SET StartDate = Null
WHERE Sno = 5
GO
```

```
Произошло обновление в таблице поставок
Msg 50000, Level 1, State 1

(10 row(s) affected)
```

Пример упрощен, чтобы более ясно продемонстрировать создание задачи в триггере.

Управление триггерами

Для управления триггерами предназначен ряд команд и инструментов баз данных. Триггеры разрешается:

- модифицировать с помощью оператора ALTER TRIGGER;
- переименовать средствами системной хранимой процедуры sp_rename;
- просмотреть путем запроса системных таблиц или с использованием системных хранимых процедур sp_helptrigger или sp_helptext;
- удалить с помощью оператора DROP TRIGGER;
- включить или выключить при помощи конструкций DISABLE TRIGGER и ENABLE TRIGGER оператора ALTER TABLE.

Программирование триггеров

1. **Псевдотаблицы Inserted и Deleted**

При срабатывании триггера на события INSERT, UPDATE или DELETE создается одна или несколько псевдотаблиц (также известных как логические таблицы). Можно рассматривать логические таблицы как журналы транзакций для события. Существует два типа логических таблиц: **Inserted** и **Deleted**. **Inserted** создается в результате события добавления или обновления данных. В ней находится набор добавленных или измененных записей. UPDATE-триггер создает также логическую таблицу **Deleted**. В ней находится исходный набор записей в том состоянии, в каком он был до операции обновления. Следующий пример создает триггер, который выводит содержимое **Inserted** и **Deleted** после события UPDATE в таблице S:

```
USE dbSPJ
GO
IF OBJECT_ID ('dbo.UpdateTables','TR') IS NOT NULL
DROP TRIGGER dbo.UpdateTables
GO
CREATE TRIGGER dbo.UpdateTables
ON S
AFTER UPDATE
AS
SELECT * FROM inserted
SELECT * FROM deleted
GO
```

После исполнения простой инструкции UPDATE, изменяющей имя поставщика с `'Алмаз'` на `'Графит'`

```
USE dbSPJ
GO
UPDATE S set Sname='Графит'
WHERE Sname='Алмаз'
GO
```

срабатывает триггер, который выводит следующие результаты:

```
1 Графит 20 Смоленск
1 Алмаз 20 Смоленск
```

После исполнения триггера обновления в таблице S содержится обновленная запись. При срабатывании триггера возможен откат модификаций таблицы S, если в триггере запрограммированы соответствующие действия. Откат транзакции также предусмотрен для триггеров INSERT и DELETE. При срабатывании триггера на удаление набор удаленных записей помещается в логическую таблицу **Deleted**. Таблица **Inserted** не участвует в событии DELETE.

2. Конструкции UPDATE (имя-столбца) и COLUMNS_UPDATED()

Важными компонентами операторов CREATE TRIGGER и ALTER TRIGGER являются две конструкции – UPDATE(*имя-столбца*) и (COLUMNS_UPDATED()). Эти конструкции разрешается включать в UPDATE- и INSERT-триггеры, и они могут располагаться в любом месте оператора CREATE TRIGGER или ALTER TRIGGER.

Конструкция IF UPDATE (имя-столбца) определяет, произошло ли в столбце имя-столбца событие INSERT или UPDATE. Если нужно задать несколько столбцов, следует разделить их конструкциями UPDATE (*имя-столбца*). Например, следующий фрагмент кода проверяет, выполнено ли добавление или обновление в столбцах First_Name и Last_Name, и выполняет некоторые действия над этими столбцами в результате событий INSERT или UPDATE:

```
USE Northwind
GO
CREATE TABLE Orders_Audit (
OrderID INT ,
CustomerID NCHAR (5) NULL ,
EmployeeID INT NULL ,
OrderDate DATETIME NULL ,
RequiredDate DATETIME NULL ,
ShippedDate DATETIME NULL ,
ShipVia INT NULL ,
Freight MONEY NULL ,
ShipName NVARCHAR (40) NULL ,
ShipAddress NVARCHAR (60) NULL ,
ShipCity NVARCHAR (15) NULL ,
ShipRegion NVARCHAR (15) NULL ,
ShipPostalCode NVARCHAR (10) NULL ,
ShipCountry NVARCHAR (15) NULL ,
DateAdded DATETIME NOT NULL DEFAULT GETDATE()
)

CREATE TRIGGER tr_iud_Orders
ON Orders
AFTER INSERT,UPDATE,DELETE
AS
BEGIN
-- If either of these two dates altered, then create the audit record
-- If either are not altered, then no audit record
IF UPDATE( ShippedDate ) OR UPDATE( RequiredDate ) THEN
INSERT INTO Orders_Audit ( OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate,
ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode,
ShipCountry )
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight,
ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
FROM deleted
END
```

Если подставить значение вместо *имя_столбца*, конструкция UPDATE вернет **true**. Конструкция COLUMNS_UPDATED() также проверяет, произошло ли обновление столбцов. Вместо t**rue** или **false** конструкция COLUMNS_UPDATED() возвращает битовую маску типа **varbinary**, описывающую столбцы, в которых выполнено добавление или обновление.

Написать конструкцию (COLUMNS_UPDATED()) сложнее, чем UPDATE (*имя-столбца*), зато она позволяет точно определить, в каких из проверенных столбцов добавлены или обновлены данные. Подлежащие проверке столбцы задают с помощью маски, представляющей номер (порядковый) каждого столбца в таблице. В расположенной далее таблице показаны первые восемь столбцов и назначенные им маски.

Столбец 1 2 3 4 5 6 7 8
Битовая маска 1 2 4 8 16 32 64 128

Следующий код позволят проверить, были ли добавлены или обновлены данные в столбцах 4 или 6:

```
IF (COIUMNS_UPDATED() & 40) > 0
```

Значение 40 – это результат суммирования маски 8 для столбца 4 и маски 32 для столбца 6. Выражение проверяет, действительно ли значение COLUMNS_UPDATED() больше 0. Другими словами, условие выполняется, если хотя бы один из двух или оба столбца обновлены. Если задать условие

```
IF (COLUMNS_UPDATED() & 40) = 40
```

то проверка обновления выполняется в обоих столбцах. Если обновление произошло только в одном из столбцов, условие не выполнится.

Чтобы проверить девять и больше столбцов, следует использовать функцию SUBSTRING, которая позволяет указать триггеру маску, подлежащую проверке. Например, следующий код позволяет проверить, обновлен ли девятый столбец:

```
IF ((SUBSTRING(COLUMNS_UPDATED(),2,1)=1))
```

Функция SUBSTRING заставляет конструкцию COLUMNS_UPDATED() перейти ко второму октету столбцов и проверить, обновлен ли первый столбец второго октета (его реальный порядковый номер равен 9). Для этого столбца возвращается значение типа varbinary, равное 1. В показанной далее таблице иллюстрируется принцип действия функции SUBSTRING, необходимой для проверки столбцов с 9 по 16.

```
IF ((SUBSTRING(COLUMNS_UPDATED(),2,y)=z))
```

Столбец 9 10 11 12 13 14 15 16
Значения y и z 1 2 4 8 16 32 64 128

Чтобы проверить несколько столбцов на предмет модификации, следует просто сложить значения битовой маски для каждого из них. Например, чтобы проверить столбцы 14 и 16, нужно задать для z выражение 160 (32 + 128).

Функции и системные команды

Для реализации бизнес-логики в триггерах предназначены различные функции и системные команды.

1. В триггерах часто используется функция @@ROWCOUNT. Она возвращает число строк, на которое повлияло исполнение предыдущего оператора T-SQL.
2. Триггер срабатывает на событие INSERT, UPDATE или DELETE, даже если при этом не изменяется ни одна строка. Поэтому для выхода из триггера при отсутствии модификаций таблицы используется системная команда RETURN.
3. В случае возникновения ошибки иногда требуется вывести сообщение с описанием ее причины. Для вывода сообщений об ошибках используется системная команда RAISERROR.
4. Пользовательские сообщения об ошибках создают с помощью системной хранимой процедуры sp_addmessage или выводят встроенные сообщения при вызове системной команды RAISERROR. За дополнительной информацией о системной хранимой процедуре sp_addmessage обращайтесь в SQL Server Books Online.
5. В триггерах, написанных на языке Transact-SQL, также иногда применяют системную команду ROLLBACK TRANSACTION. Она вызывает откат всего пакета триггера. При фатальной ошибке также происходит откат, но в неявном виде. Если задачей триггера является завершение транзакции в .любом случае (кроме тех, когда во время транзакции возникает фатальная ошибка), то в код триггера можно не включать системную команду ROLLBACK TRANSACTION.

Использование триггера для поддержки простой ссылочной целостности

Следует заметить, что использовать в этих целях триггеры — не всегда лучшее решение. Предположим, что в демонстрационной базе данных Northwind не действуют никакие декларативные правила по поддержке ссылочной целостности (DRI). Для этого необходимо сформировать копию базы данных Northwind, в которой будут отсутствовать DRI. Пусть это будет база данных NorthwindTriggers. В этом случае появляется пространство для экспериментов. Займемся отношениями, которые в оригинальной базе данных Northwind обеспечивались при помощи DRI. Восстановим отношение между таблицами Customers и Orders. При помощи данного отношения грантируется, что нльзя добавить заказ в таблицу Orders, если он ссылается на CustomerID, отсутствующий в таблице Customers. Чтобы реализовать это ограничение, создадим следующий триггер:

```
CREATE TRIGGER OrderHasCustomer
ON Orders
FOR INSERT, UPDATE
AS
IF EXISTS
(
SELECT 'True'
FROM Inserted i LEFT JOIN Customers c ON i.CustomerID = c.CustomerID
WHERE c.CustomerID IS NULL
)
BEGIN
RAISERROR('Счет должен иметь валидный CustomerID',16,1)
ROLLBACK TRAN
END
```

Теперь, если попытаться добавить в таблицу `Orders` данные, не удовлетворяющие условиям, заданным в триггере:

```
INSERT Orders
DEFAULT VALUES
```

то будет получено сообщение об ошибке:

```
Сообщение 50000, уровень 16, состояние 1, процедура OrderHasCustomer, строка 15
Счет должен иметь валидный CustomerID
Сообщение 3609, уровень 16, состояние 1, строка 1
Транзакция завершилась в триггере. Выполнение пакета прервано.
```

Данный триггер можно усовершенствовать, чтобы повысить его информативность.

Практика программирования триггеров DML T-SQL

Пример 1. Создание и использование триггера AFTER

```
USE Northwind;
GO

-- Создаем таблицу, выполняющую роль журнала аудита изменений в таблицах
CREATE TABLE dbo.DmlActionLog
(
EntryNum int IDENTITY(1, 1) PRIMARY KEY NOT NULL,
SchemaName sysname NOT NULL,
TableName sysname NOT NULL,
ActionType nvarchar(10) NOT NULL,
ActionXml xml NOT NULL,
UserName nvarchar(256) NOT NULL,
Spid int NOT NULL,
ActionDateTime datetime NOT NULL DEFAULT (GETDATE())
);
GO

-- Создаем триггер AFTER INSERT, UPDATE, DELETE для таблицы Customers,
-- который регистрирует любые изменения в таблице Customers
```

```
CREATE TRIGGER CustomersChangeAudit
ON Customers
AFTER INSERT, UPDATE, DELETE
NOT FOR REPLICATION
AS
BEGIN
-- Получить число обработанных строк
DECLARE @Count int;
SET @Count = @@ROWCOUNT

-- Убедиться в том, что хотя бы одна строка на самом деле обработана
IF (@Count > 0)
BEGIN
-- Отключить сообщения вида "rows affected"
SET NOCOUNT ON;
DECLARE @ActionType nvarchar(10);
DECLARE @ActionXml xml;

-- Получить количество вставляемых строк
DECLARE @inserted_count int;
SET @inserted_count = (SELECT COUNT(*) FROM inserted);

-- Получить количество удаляемых строк
DECLARE @deleted_count int;
SET @deleted_count = (SELECT COUNT(*) FROM deleted);

-- Определить тип действия DML, которое привело к срабатыванию триггера
SELECT @ActionType = CASE
WHEN (@inserted_count > 0) AND (@deleted_count = 0)
THEN N'insert'
WHEN (@inserted_count = 0) AND (@deleted_count > 0)
THEN N'delete'
ELSE N'update'
END;

-- Использовать FOR XML AUTO для получения снимков до и после изменения
       -- данных в формате XML
SELECT @ActionXml = COALESCE
(
(
SELECT *
FROM deleted
FOR XML AUTO
), N'<deleted/>'
) + COALESCE
(
(
SELECT *
FROM inserted
FOR XML AUTO
), N'<inserted/>'
);

-- Вставить строки для протоколирования действий в журнал аудита таблицы
INSERT INTO dbo.DmlActionLog
(
SchemaName,
TableName,
ActionType,
ActionXml,
UserName,
Spid,
```

```
ActionDateTime
)
SELECT
OBJECT_SCHEMA_NAME(@@PROCID, DB_ID()),
OBJECT_NAME(t.parent_id, DB_ID()),
@ActionType,
@ActionXml,
USER_NAME(),
@@SPID,
GETDATE()
FROM sys.triggers t
WHERE t.object_id = @@PROCID;
END;
END;
GO
```

Тестирование триггера на инструкциях UPDATE, INSERT и DELETE

```
USE Northwind;
GO
UPDATE Customers
SET CompanyName = N'Lotus Software'
WHERE CustomerID = 'WOLZA';
GO
INSERT INTO Customers
(
CustomerID,
CompanyName
)
VALUES
(
N'ZXCVB',
N'Mandriva'
);
GO
DELETE
FROM Customers
WHERE CompanyName = N'Mandriva';
GO
SELECT
EntryNum,
SchemaName,
TableName,
ActionType,
ActionXml,
UserName,
Spid,
ActionDateTime
FROM dbo.DmlActionLog;
GO
```

Результаты срабатывания триггера

```
EntryNum SchemaName TableName ActionType ActionXml UserName Spid ActionDateTime
-------- ---------- --------- ---------- --------- -------- ---- -----------------------
1 dbo Customers update <deleted dbo 53 2011-03-15 03:50:17.450
2 dbo Customers insert <deleted dbo 53 2011-03-15 03:50:17.530
3 dbo Customers delete <deleted dbo 53 2011-03-15 03:50:17.560

<deleted CustomerID="WOLZA" CompanyName="Wolski Zajazd" ContactName="Zbyszek Piestrzeniewicz"
ContactTitle="Owner" Address="ul. Filtrowa 68" City="Warszawa" PostalCode="01-012"
Country="Poland" Phone="(26) 642-7012" Fax="(26) 642-7012" />
```

```
<inserted CustomerID="WOLZA" CompanyName="Lotus Software" ContactName="Zbyszek Piestrzeniewicz"
ContactTitle="Owner"    Address="ul.    Filtrowa    68"    City="Warszawa"    PostalCode="01-012"
Country="Poland" Phone="(26) 642-7012" Fax="(26) 642-7012" />
<deleted />
<inserted CustomerID="ZXCVB" CompanyName="Mandriva" />
<deleted CustomerID="ZXCVB" CompanyName="Mandriva" />
<inserted />
```

Пример 2. Создание и использование триггера INSTEAD OF

```
-- Создаем тестовую базу данных специально для примера
CREATE DATABASE OurInsteadOfTest;
GO

USE OurInsteadOfTest;
GO

-- Создаем тестовые таблицы
CREATE TABLE dbo.Customers
(
CustomerID varchar(5) NOT NULL PRIMARY KEY ,
Name varchar(40) NOT NULL
);

CREATE TABLE dbo.Orders
(
OrderID int IDENTITY NOT NULL PRIMARY KEY,
CustomerID varchar(5) NOT NULL
REFERENCES Customers(CustomerID),
OrderDate datetime NOT NULL
);

CREATE TABLE dbo.Products
(
ProductID int IDENTITY NOT NULL PRIMARY KEY,
Name varchar(40) NOT NULL,
UnitPrice money NOT NULL
);

CREATE TABLE dbo.OrderItems
(
OrderID int NOT NULL
REFERENCES dbo.Orders(OrderID),
ProductID int NOT NULL
REFERENCES dbo.Products(ProductID),
UnitPrice money NOT NULL,
Quantity int NOT NULL
CONSTRAINT PKOrderItem PRIMARY KEY CLUSTERED
(OrderID, ProductID)
);
select * FROM dbo.OrderItems

-- Наполняем тестовые таблицы небольшим количеством данных
INSERT dbo.Customers
VALUES ('ABCDE', 'Bob''s Pretty Good Garage');

INSERT dbo.Orders
VALUES ('ABCDE', CURRENT_TIMESTAMP);

INSERT dbo.Products
VALUES ('Widget', 5.55),
```

```
('Thingamajig', 8.88)

INSERT dbo.OrderItems
VALUES (1, 1, 5.55, 3);

USE OurInsteadOfTest;
GO

-- Создаем представление с параметром WITH SCHEMABINDING
CREATE VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT o.OrderID, o.OrderDate, od.ProductID, p.Name, od.Quantity, od.UnitPrice
FROM dbo.Orders AS o JOIN dbo.OrderItems AS od ON o.OrderID = od.OrderID
JOIN dbo.Products AS p ON od.ProductID = p.ProductID;
GO

-- Делаем попытку добавить в представление одну запись
INSERT INTO CustomerOrders_vw
(
OrderID,
OrderDate,
ProductID,
Quantity,
UnitPrice
)
VALUES
(
1,
'1998-04-06',
2,
10,
6.00
);

-- Попытка завершается неудачей
Msg 4405, Level 16, State 1, Line 1
View or function 'CustomerOrders_vw' is not updatable because the modification affects multiple
base tables.

-- Создаем INSTEAD OF триггер для представления
CREATE TRIGGER trCustomerOrderInsert ON CustomerOrders_vw
INSTEAD OF INSERT
AS
BEGIN
-- Проверям, на самом ли деле INSERT пытается добавить хотя бы одну строку.
-- (A WHERE clause might have filtered everything out)
IF (SELECT COUNT(*) FROM Inserted) > 0
BEGIN
INSERT INTO dbo.OrderItems
SELECT i.OrderID,
i.ProductID,
i.UnitPrice,
i.Quantity
FROM Inserted AS i
JOIN Orders AS o
ON i.OrderID = o.OrderID;
-- Если есть записи в псевдотаблице Inserted,
-- но нет соответствия с таблицей Orders,
-- то операция вставки в таблицу OrderItems не может быть выполнена.
IF @@ROWCOUNT = 0
RAISERROR('No matching Orders. Cannot perform insert',10,1);
```

```
END
END;
GO

-- Делаем еще одну попытку добавить в представление одну запись
INSERT INTO CustomerOrders_vw
(
OrderID,
OrderDate,
ProductID,
Quantity,
UnitPrice
)
VALUES
(
1,
'1998-04-06',
2,
10,
6.00
);

-- Попытка завершается удачей
```