SQL Check

**CHECK** - устанавливает допустимые значения, которые могут быть вставлены в таблицу.

**SQL CHECK Ограничения при CREATE TABLE**

Следующие SQL создает CHECK ограничитель в колонке "P\_Id", при создании таблицы "Persons" . Данный ограничитель позволяет вставлять в колонку "P\_Id" только числа больше нуля.

|  |  |
| --- | --- |
| 1234567 | [**CREATE**](http://search.mysql.com/search?site=refman-51&q=CREATE&) [**TABLE**](http://search.mysql.com/search?site=refman-51&q=TABLE&) Persons (P\_Id [**int**](http://search.mysql.com/search?site=refman-51&q=INT&) [**NOT**](http://dev.mysql.com/doc/refman/5.1/en/non-typed-operators.html) [**NULL**](http://search.mysql.com/search?site=refman-51&q=NULL&) [**CHECK**](http://search.mysql.com/search?site=refman-51&q=CHECK&) (P\_Id>0),LastName [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255) [**NOT**](http://dev.mysql.com/doc/refman/5.1/en/non-typed-operators.html) [**NULL**](http://search.mysql.com/search?site=refman-51&q=NULL&),FirstName [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255),Address [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255),City [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255)) |

Для создания нескольких CHECK используйте следующий синтаксис SQL:

|  |  |
| --- | --- |
| 12345678 | [**CREATE**](http://search.mysql.com/search?site=refman-51&q=CREATE&) [**TABLE**](http://search.mysql.com/search?site=refman-51&q=TABLE&) Persons (P\_Id [**int**](http://search.mysql.com/search?site=refman-51&q=INT&) [**NOT**](http://dev.mysql.com/doc/refman/5.1/en/non-typed-operators.html) [**NULL**](http://search.mysql.com/search?site=refman-51&q=NULL&),LastName [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255) [**NOT**](http://dev.mysql.com/doc/refman/5.1/en/non-typed-operators.html) [**NULL**](http://search.mysql.com/search?site=refman-51&q=NULL&),FirstName [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255),Address [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255),City [**varchar**](http://search.mysql.com/search?site=refman-51&q=VARCHAR&)(255),[**CONSTRAINT**](http://search.mysql.com/search?site=refman-51&q=CONSTRAINT&) chk\_Person [**CHECK**](http://search.mysql.com/search?site=refman-51&q=CHECK&) (P\_Id>0 [**AND**](http://dev.mysql.com/doc/refman/5.1/en/non-typed-operators.html) City='Sandnes')) |

**SQL CHECK Ограничения при ALTER TABLE**

Следующие SQL создает CHECK ограничитель в колонке "P\_Id", когда таблицы "Persons" уже создана. Данный ограничитель позволяет вставлять в колонку "P\_Id" только числа больше нуля.

|  |  |
| --- | --- |
| 12 | [**ALTER**](http://search.mysql.com/search?site=refman-51&q=ALTER&) [**TABLE**](http://search.mysql.com/search?site=refman-51&q=TABLE&) Persons[**ADD**](http://search.mysql.com/search?site=refman-51&q=ADD&) [**CHECK**](http://search.mysql.com/search?site=refman-51&q=CHECK&) (P\_Id>0) |

Для создания нескольких CHECK используйте следующий синтаксис SQL:

|  |  |
| --- | --- |
| 12 | [**ALTER**](http://search.mysql.com/search?site=refman-51&q=ALTER&) [**TABLE**](http://search.mysql.com/search?site=refman-51&q=TABLE&) Persons[**ADD**](http://search.mysql.com/search?site=refman-51&q=ADD&) [**CONSTRAINT**](http://search.mysql.com/search?site=refman-51&q=CONSTRAINT&) chk\_Person [**CHECK**](http://search.mysql.com/search?site=refman-51&q=CHECK&) (P\_Id>0 [**AND**](http://dev.mysql.com/doc/refman/5.1/en/non-typed-operators.html) City='Sandnes') |

**Удаление CHECK**

Для удаления ограничителя CHECK используйте следующий SQL:

|  |  |
| --- | --- |
| 12 | [**ALTER**](http://search.mysql.com/search?site=refman-51&q=ALTER&) [**TABLE**](http://search.mysql.com/search?site=refman-51&q=TABLE&) Persons[**DROP**](http://search.mysql.com/search?site=refman-51&q=DROP&) [**CONSTRAINT**](http://search.mysql.com/search?site=refman-51&q=CONSTRAINT&) chk\_Person |

***Триггер*** - это механизм, который вызывается, когда в указанной таблице происходит определенное действие. Каждый триггер имеет следующие основные составляющие: имя, действие и исполнение. Действием триггера может быть или инструкция DML (INSERT, UPDATE или DELETE), или инструкция DDL. Таким образом, существует два типа триггеров: триггеры DML и триггеры DDL. Исполнительная составляющая триггера обычно состоит из хранимой процедуры или пакета.

## Создание триггера DML

Триггер создается с помощью инструкции **CREATE TRIGGER**, которая имеет следующий синтаксис:

CREATE TRIGGER [schema\_name.]trigger\_name

 ON {table\_name | view\_name}

 [WITH dml\_trigger\_option [,…]]

 {FOR | AFTER | INSTEAD OF} { [INSERT] [,] [UPDATE] [,] [DELETE]}

 [WITH APPEND]

 {AS sql\_statement | EXTERNAL NAME method\_name}

Предшествующий синтаксис относится только к триггерам DML. Триггеры DDL имеют несколько иную форму синтаксиса.

Здесь в параметре schema\_name указывается имя схемы, к которой принадлежит триггер, а в параметре trigger\_name - имя триггера. В параметре table\_name задается имя таблицы, для которой создается триггер. (Также поддерживаются триггеры для представлений, на что указывает наличие параметра view\_name.)

Также можно задать тип триггера с помощью двух дополнительных параметров: AFTER и INSTEAD OF. (Параметр FOR является синонимом параметра AFTER.) **Триггеры типа AFTER** вызываются после выполнения действия, запускающего триггер, а **триггеры типа INSTEAD OF** выполняются вместо действия, запускающего триггер. Триггеры AFTER можно создавать только для таблиц, а триггеры INSTEAD OF - как для таблиц, так и для представлений.

Параметры INSERT, UPDATE и DELETE задают действие триггера. Под действием триггера имеется в виду инструкция Transact-SQL, которая запускает триггер. Допускается любая комбинация этих трех инструкций. Инструкция DELETE не разрешается, если используется параметр IF UPDATE.

Как можно видеть в синтаксисе инструкции CREATE TRIGGER, действие (или действия) триггера указывается в спецификации AS sql\_statement.

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы и для каждого действия (INSERT, UPDATE и DELETE). По умолчанию определенного порядка исполнения нескольких триггеров для данного модифицирующего действия не имеется.

Только владелец базы данных, администраторы DDL и владелец таблицы, для которой определяется триггер, имеют право создавать триггеры для текущей базы данных. (В отличие от разрешений для других типов инструкции CREATE это разрешение не может передаваться.)

### Изменение структуры триггера

Язык Transact-SQL также поддерживает инструкцию **ALTER TRIGGER**, которая модифицирует структуру триггера. Эта инструкция обычно применяется для изменения тела триггера. Все предложения и параметры инструкции ALTER TRIGGER имеют такое же значение, как и одноименные предложения и параметры инструкции CREATE TRIGGER.

Для удаления триггеров в текущей базе данных применяется инструкция **DROP TRIGGER**.

### Использование виртуальных таблиц deleted и inserted

При создании действия триггера обычно требуется указать, ссылается ли он на значение столбца до или после его изменения действием, запускающим триггер. По этой причине, для тестирования следствия инструкции, запускающей триггер, используются две специально именованные виртуальные таблицы:

* deleted - содержит копии строк, удаленных из таблицы;
* inserted - содержит копии строк, вставленных в таблицу.

Структура этих таблиц эквивалентна структуре таблицы, для которой определен триггер.

Таблица deleted используется в том случае, если в инструкции CREATE TRIGGER указывается предложение DELETE или UPDATE, а если в этой инструкции указывается предложение INSERT или UPDATE, то используется таблица inserted. Это означает, что для каждой инструкции DELETE, выполненной в действии триггера, создается таблица deleted. Подобным образом для каждой инструкции INSERT, выполненной в действии триггера, создается таблица inserted.

Инструкция UPDATE рассматривается, как инструкция DELETE, за которой следует инструкция INSERT. Поэтому для каждой инструкции UPDATE, выполненной в действии триггера, создается как таблица deleted, так и таблица inserted (в указанной последовательности).

Таблицы inserted и deleted реализуются, используя управление версиями строк, которое рассматривалось в предыдущей статье. Когда для таблицы с соответствующими триггерами выполняется инструкция DML (INSERT, UPDATE или DELETE), для всех изменений в этой таблице всегда создаются версии строк. Когда триггеру требуется информация из таблицы deleted, он обращается к данным в хранилище версий строк. В случае таблицы inserted, триггер обращается к самым последним версиям строк.

В качестве хранилища версий строк механизм управления версиями строк использует системную базу данных tempdb. По этой причине, если база данных содержит большое число часто используемых триггеров, следует ожидать значительного увеличения объема этой системной базы данных.

## Области применения DML-триггеров

Такие триггеры применяются для решения разнообразных задач..

### Триггеры AFTER

Как вы уже знаете, триггеры AFTER вызываются после того, как выполняется действие, запускающее триггер. Триггер AFTER задается с помощью ключевого слова AFTER или FOR. Триггеры AFTER можно создавать только для базовых таблиц. Триггеры этого типа можно использовать для выполнения, среди прочих, следующих операций:

* создания журнала логов действий в таблицах базы данных;
* реализации бизнес-правил;
* принудительного обеспечения ссылочной целостности.

#### Создание журнала логов

В SQL Server можно выполнять отслеживание изменения данных, используя систему перехвата изменения данных CDC (change data capture). Эту задачу можно также решить с помощью триггеров DML. В примере ниже показывается, как с помощью триггеров можно создать журнал логов действий в таблицах базы данных:

USE SampleDb;

/\* Таблица AuditBudget используется в качестве

журнала логов действий в таблице Project \*/

GO

CREATE TABLE AuditBudget (

 ProjectNumber CHAR(4) NULL,

 UserName CHAR(16) NULL,

 Date DATETIME NULL,

 BudgetOld FLOAT NULL,

 BudgetNew FLOAT NULL

);

GO

CREATE TRIGGER trigger\_ModifyBudget

 ON Project AFTER UPDATE

 AS IF UPDATE(budget)

BEGIN

 DECLARE @budgetOld FLOAT

 DECLARE @budgetNew FLOAT

 DECLARE @projectNumber CHAR(4)

 SELECT @budgetOld = (SELECT Budget FROM deleted)

 SELECT @budgetNew = (SELECT Budget FROM inserted)

 SELECT @projectNumber = (SELECT Number FROM deleted)

 INSERT INTO AuditBudget VALUES

 (@projectNumber, USER\_NAME(), GETDATE(), @budgetOld, @budgetNew)

END

В этом примере создается таблица AuditBudget, в которой сохраняются все изменения столбца Budget таблицы Project. Изменения этого столбца будут записываться в эту таблицу посредством триггера trigger\_ModifyBudget.

Этот триггер активируется для каждого изменения столбца Budget с помощью инструкции UPDATE. При выполнении этого триггера значения строк таблиц deleted и inserted присваиваются соответствующим переменным @budgetOld, @budgetNew и @projectNumber. Эти присвоенные значения, совместно с именем пользователя и текущей датой, будут затем вставлены в таблицу AuditBudget.

В этом примере предполагается, что за один раз будет обновление только одной строки. Поэтому этот пример является упрощением общего случая, когда триггер обрабатывает многострочные обновления. Если выполнить следующие инструкции Transact-SQL:

USE SampleDb;

UPDATE Project

 SET Budget = 200000

 WHERE Number = 'p2';

то содержимое таблицы AuditBudget будет таким:



#### Реализация бизнес-правил

С помощью триггеров можно создавать бизнес-правила для приложений. Создание такого триггера показано в примере ниже:

USE SampleDb;

-- Триггер trigger\_TotalBudget является примером использования

-- триггера для реализации бизнес-правила

GO

CREATE TRIGGER trigger\_TotalBudget

 ON Project AFTER UPDATE

 AS IF UPDATE (Budget)

BEGIN

 DECLARE @sum\_old1

 FLOAT DECLARE @sum\_old2

 FLOAT DECLARE @sum\_new FLOAT

 SELECT @sum\_new = (SELECT SUM(Budget) FROM inserted)

 SELECT @sum\_old1 = (SELECT SUM(p.Budget)

 FROM project p WHERE p.Number

 NOT IN (SELECT d.Number FROM deleted d))

 SELECT @sum\_old2 = (SELECT SUM(Budget) FROM deleted)

 IF @sum\_new > (@sum\_old1 + @sum\_old2) \* 1.5

 BEGIN

 PRINT 'Бюджет не изменился'

 ROLLBACK TRANSACTION

 END

 ELSE

 PRINT 'Изменение бюджета выполнено'

END

Здесь создается правило для управления модификацией бюджетов проектов. Триггер trigger\_TotalBudget проверяет каждое изменение бюджетов и выполняет только такие инструкции UPDATE, которые увеличивают сумму всех бюджетов не более чем на 50%. В противном случае для инструкции UPDATE выполняется откат посредством инструкции ROLLBACK TRANSACTION.

#### Принудительное обеспечение ограничений целостности

В системах управления базами данных применяются два типа ограничений для обеспечения целостности данных: декларативные ограничения, которые определяются с помощью инструкций языка CREATE TABLE и ALTER TABLE; процедурные ограничения целостности, которые реализуются посредством триггеров.

В обычных ситуациях следует использовать декларативные ограничения для обеспечения целостности, поскольку они поддерживаются системой и не требуют реализации пользователем. Применение триггеров рекомендуется только в тех случаях, для которых декларативные ограничения для обеспечения целостности отсутствуют.

В примере ниже показано принудительное обеспечение ссылочной целостности посредством триггеров для таблиц Employee и Works\_on:

USE SampleDb;

GO

CREATE TRIGGER trigger\_WorksonIntegrity

 ON Works\_on AFTER INSERT, UPDATE

 AS IF UPDATE(EmpId)

 BEGIN

 IF (SELECT Employee.Id

 FROM Employee, inserted

 WHERE Employee.Id = inserted.EmpId) IS NULL

 BEGIN

 ROLLBACK TRANSACTION

 PRINT 'Строка не была вставлена/модифицирована'

 END

 ELSE

 PRINT 'Строка была вставлена/модифицирована'

 END

Триггер trigger\_WorksonIntegrity в этом примере проверяет ссылочную целостность для таблиц Employee и Works\_on. Это означает, что проверяется каждое изменение столбца Id в ссылочной таблице Works\_on, и при любом нарушении этого ограничения выполнение этой операции не допускается. (То же самое относится и к вставке в столбец Id новых значений.) Инструкция ROLLBACK TRANSACTION во втором блоке BEGIN выполняет откат инструкции INSERT или UPDATE в случае нарушения ограничения для обеспечения ссылочной целостности.

В этом примере триггер выполняет проверку на проблемы ссылочной целостности первого и второго случая между таблицами Employee и Works\_on. А в примере ниже показан триггер, который выполняет проверку на проблемы ссылочной целостности третьего и четвертого случая между этими же таблицами (эти случаи обсуждались в статье ["Transact-SQL - создание таблиц"](https://professorweb.ru/my/sql-server/2012/level2/2_6.php)):

USE SampleDb;

GO

CREATE TRIGGER trigger\_RefintWorkson2

 ON Employee AFTER DELETE, UPDATE

 AS IF UPDATE (Id)

 BEGIN

 IF (SELECT COUNT(\*)

 FROM Works\_on, deleted

 WHERE Works\_on.EmpId = deleted.Id) > 0

 BEGIN

 ROLLBACK TRANSACTION

 PRINT 'Строка не была вставлена/модифицирована'

 END

 ELSE

 PRINT 'Строка была вставлена/модифицирована'

 END

## Примеры

### A. Использование триггера DML с предупреждающим сообщением.

Следующий триггер DML отправляет клиенту сообщение, когда кто-то пытается добавить или изменить данные в таблице Customer в базе данных.

CREATE TRIGGER reminder1

ON Sales.Customer

AFTER INSERT, UPDATE

AS RAISERROR ('Notify Customer Relations', 16, 10);

GO

### В. Использование триггера DML AFTER для принудительного применения бизнес-правил между таблицами PurchaseOrderHeader и VendorUsing.

Поскольку ограничение CHECK может содержать ссылки только на столбцы, для которых определены ограничения на уровне столбцов или таблицы, любые межтабличные ограничения (в данном случае бизнес-правила) должны быть заданы в виде триггеров.

В следующем примере создается триггер DML в базе данных AdventureWorks 2012.Этот триггер проверяет оценку кредитоспособности поставщика (не 5) при попытке добавить новый заказ на покупку в таблицу PurchaseOrderHeader. Для получения сведений о кредитоспособности поставщика требуется ссылка на таблицу Vendor. В случае слишком низкой кредитоспособности выводится соответствующее сообщение и вставка не выполняется.

-- This trigger prevents a row from being inserted in the Purchasing.PurchaseOrderHeader

-- table when the credit rating of the specified vendor is set to 5 (below average).

CREATE TRIGGER Purchasing.LowCredit ON Purchasing.PurchaseOrderHeader

AFTER INSERT

AS

IF (@@ROWCOUNT\_BIG = 0)

RETURN;

IF EXISTS (SELECT \*

 FROM Purchasing.PurchaseOrderHeader AS p

 JOIN inserted AS i

 ON p.PurchaseOrderID = i.PurchaseOrderID

 JOIN Purchasing.Vendor AS v

 ON v.BusinessEntityID = p.VendorID

 WHERE v.CreditRating = 5

 )

BEGIN

RAISERROR ('A vendor''s credit rating is too low to accept new

purchase orders.', 16, 1);

ROLLBACK TRANSACTION;

RETURN

END;

GO

-- This statement attempts to insert a row into the PurchaseOrderHeader table

-- for a vendor that has a below average credit rating.

-- The AFTER INSERT trigger is fired and the INSERT transaction is rolled back.

INSERT INTO Purchasing.PurchaseOrderHeader (RevisionNumber, Status, EmployeeID,

VendorID, ShipMethodID, OrderDate, ShipDate, SubTotal, TaxAmt, Freight)

VALUES (

2

,3

,261

,1652

,4

,GETDATE()

,GETDATE()

,44594.55

,3567.564

,1114.8638 );

GO