

Практическая работа №10

Задание

1. Ознакомиться с практической работой.
2. Создать представление с помощью конструктора для своей предметной области.
3. Создать и изменить 2 представления для своей предметной области с помощью запросов.
4. Оформить отчёт, который должен содержать запросы на создание представлений и результаты их выполнения.
5. Защитить работу, ответив на вопросы по её выполнению.

ПРЕДСТАВЛЕНИЯ

Представления или Views представляют виртуальные таблицы. Но в отличие от обычных стандартных таблиц в базе данных представления содержат запросы, которые динамически извлекают используемые данные.

Представления дают нам ряд преимуществ. Они упрощают комплексные SQL-операции. Они защищают данные, так как представления могут дать доступ к части таблицы, а не ко всей таблице. Представления также позволяют возвращать отформатированные значения из таблиц в нужной и удобной форме.

Для создания представления используется команда CREATE VIEW, которая имеет следующую форму:

```
CREATE VIEW название_представления [(столбец_1, столбец_2, ....)]  
AS выражение_SELECT
```

Например, пусть у нас есть три связанных таблицы:

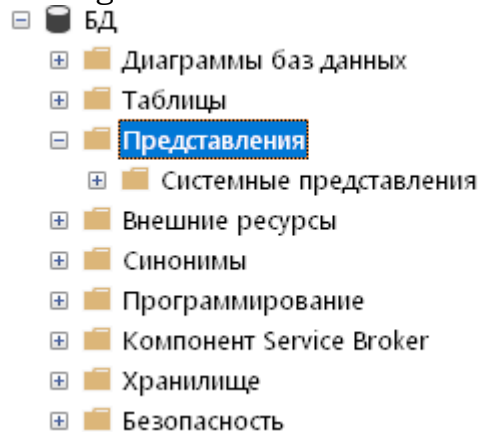
```
CREATE TABLE Products  
(  
    Id INT IDENTITY PRIMARY KEY,  
    ProductName NVARCHAR(30) NOT NULL,  
    Manufacturer NVARCHAR(20) NOT NULL,  
    ProductCount INT DEFAULT 0,  
    Price MONEY NOT NULL  
);  
CREATE TABLE Customers  
(  
    Id INT IDENTITY PRIMARY KEY,  
    FirstName NVARCHAR(30) NOT NULL  
);  
CREATE TABLE Orders  
(  
    Id INT IDENTITY PRIMARY KEY,  
    ProductId INT NOT NULL REFERENCES Products(Id) ON DELETE CASCADE,  
    CustomerId INT NOT NULL REFERENCES Customers(Id) ON DELETE CASCADE,  
    CreatedAt DATE NOT NULL,  
    ProductCount INT DEFAULT 1,  
    Price MONEY NOT NULL  
);
```

Теперь добавим в базу данных, в которой содержатся данные таблицы, следующее представление:

```
CREATE VIEW OrdersProductsCustomers AS  
SELECT Orders.CreatedAt AS OrderDate,  
    Customers.FirstName AS Customer,  
    Products.ProductName As Product
```

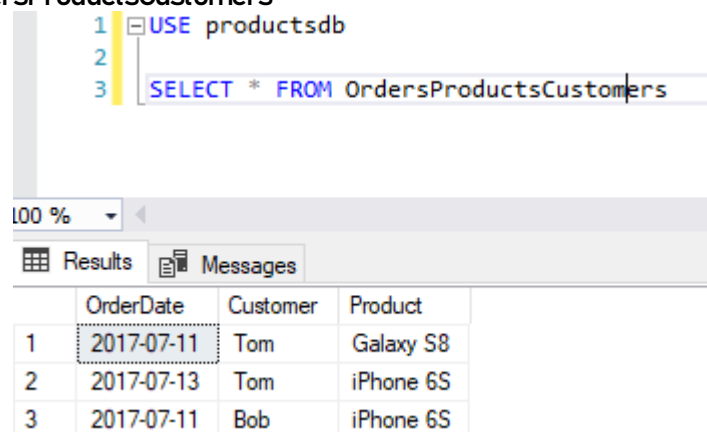
```
FROM Orders INNER JOIN Products ON Orders.ProductId = Products.Id  
INNER JOIN Customers ON Orders.CustomerId = Customers.Id
```

То есть данное представление фактически будет возвращать сводные данные из трех таблиц. И после его создания мы сможем его увидеть в узле Views у выбранной базы данных в SQL Server Management Studio:



Теперь используем созданное выше представление для получения данных:

```
SELECT * FROM OrdersProductsCustomers
```



При создании представлений следует учитывать, что представления, как и таблицы, должны иметь уникальные имена в рамках той же базы данных.

Представления могут иметь не более 1024 столбцов и могут обращаться не более чем к 256 таблицам.

Также можно создавать представления на основе других представлений. Такие представления еще называют вложенными (nested views). Однако уровень вложенности не может быть больше 32-х.

Команда SELECT, используемая в представлении, не может включать выражения INTO или ORDER BY (за исключением тех случаев, когда также применяется выражение TOP или OFFSET). Если же необходима сортировка данных в представлении, то выражение ORDER BY применяется в команде SELECT, которая извлекает данные из представления.

Также при создании представления можно определить набор его столбцов:

```
CREATE VIEW OrdersProductsCustomers2 (OrderDate, Customer, Product)  
AS SELECT Orders.CreatedAt,  
       Customers.FirstName,  
       Products.ProductName  
FROM Orders INNER JOIN Products ON Orders.ProductId = Products.Id  
INNER JOIN Customers ON Orders.CustomerId = Customers.Id
```

Изменение представления

Для изменения представления используется команда ALTER VIEW. Эта команда имеет практически тот же самый синтаксис, что и CREATE VIEW:

```
ALTER VIEW название_представления [(столбец_1, столбец_2, ....)]  
AS выражение_SELECT
```

Например, изменим выше созданное представление OrdersProductsCustomers:

```
ALTER VIEW OrdersProductsCustomers  
AS SELECT Orders.CreatedAt AS OrderDate,  
       Customers.FirstName AS Customer,  
       Products.ProductName AS Product,  
       Products.Manufacturer AS Manufacturer  
FROM Orders INNER JOIN Products ON Orders.ProductId = Products.Id  
INNER JOIN Customers ON Orders.CustomerId = Customers.Id
```

Удаление представления

Для удаления представления вызывается команда DROP VIEW:

```
DROP VIEW OrdersProductsCustomers
```

Также стоит отметить, что при удалении таблиц также следует удалить и представления, которые используют эти таблицы.

Практическая работа № 11

Задание

1. Ознакомиться с практической работой.
2. Создать 3 хранимые процедуры для своей предметной области с помощью запросов.
3. Оформить отчёт, который должен содержать запросы на создание хранимых процедур и результаты их выполнения.
4. Защитить работу, ответив на вопросы по её выполнению.

ХРАНИМЫЕ ПРОЦЕДУРЫ

Нередко операция с данными представляет набор инструкций, которые необходимо выполнить в определенной последовательности. Например, при добавлении данных покупки товара необходимо внести данные в таблицу заказов. Однако перед этим надо проверить, а есть ли покупаемый товар в наличии. Возможно, при этом понадобится проверить еще ряд дополнительных условий. То есть фактически процесс покупки товара охватывает несколько действий, которые должны выполняться в определенной последовательности. И в этом случае более оптимально будет инкапсулировать все эти действия в один объект - хранимую процедуру (stored procedure).

То есть по сути хранимые процедуры представляют набор инструкций, которые выполняются как единое целое. Тем самым хранимые процедуры позволяют упростить комплексные операции и вынести их в единый объект. Изменится процесс покупки товара, соответственно достаточно будет изменить код процедуры. То есть процедура также упрощает управление кодом.

Также хранимые процедуры позволяют ограничить доступ к данным в таблицах и тем самым уменьшить вероятность преднамеренных или неосознанных нежелательных действий в отношении этих данных.

И еще один важный аспект - производительность. Хранимые процедуры обычно выполняются быстрее, чем обычные SQL-инструкции. Все потому что код процедур компилируется один раз при первом ее запуске, а затем сохраняется в скомпилированной форме.

Для создания хранимой процедуры применяется команда CREATE PROCEDURE или CREATE PROC.

Таким образом, хранимая процедура имеет три ключевых особенности: упрощение кода, безопасность и производительность.

Например, пусть в базе данных есть таблица, которая хранит данные о товарах:

```
CREATE TABLE Products
(
    Id INT IDENTITY PRIMARY KEY,
    ProductName NVARCHAR(30) NOT NULL,
    Manufacturer NVARCHAR(20) NOT NULL,
    ProductCount INT DEFAULT 0,
    Price MONEY NOT NULL
);
```

Создадим хранимую процедуру для извлечения данных из этой таблицы:

```
USE productsdb;
GO
CREATE PROCEDURE ProductSummary AS
SELECT ProductName AS Product, Manufacturer, Price
FROM Products
```

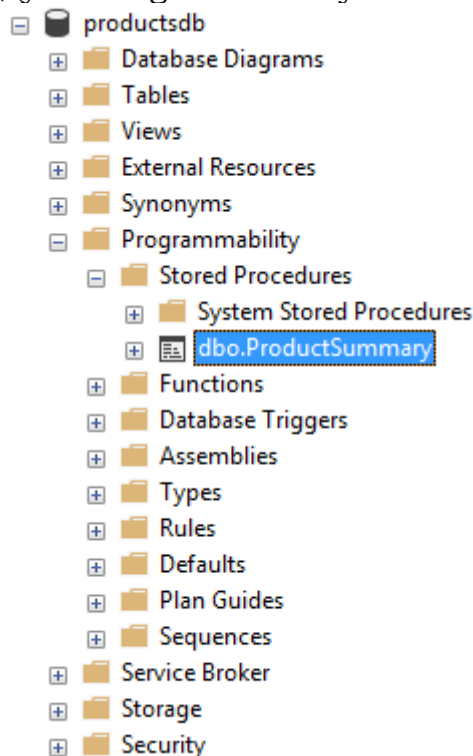
Поскольку команда CREATE PROCEDURE должна вызываться в отдельном пакете, то после команды USE, которая устанавливает текущую базу данных, используется команда GO для определения нового пакета.

После имени процедуры должно идти ключевое слово AS.

Для отделения тела процедуры от остальной части скрипта код процедуры нередко помещается в блок BEGIN...END:

```
USE productsdb;  
GO  
CREATE PROCEDURE ProductSummary AS  
BEGIN  
    SELECT ProductName AS Product, Manufacturer, Price  
    FROM Products  
END;
```

После добавления процедуры мы ее можем увидеть в узле базы данных в SQL Server Management Studio в подузле Programmability -> Stored Procedures:

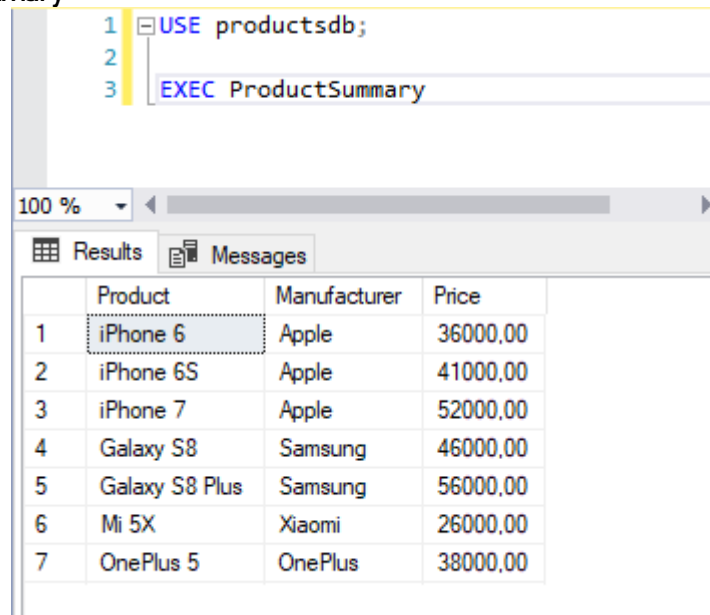


И мы сможем управлять процедурой также и через визуальный интерфейс.

Выполнение процедуры

Для выполнения хранимой процедуры вызывается команда EXEC или EXECUTE:

EXEC ProductSummary



Удаление процедуры

Для удаления процедуры применяется команда DROP PROCEDURE:
DROP PROCEDURE ProductSummary

Параметры в процедурах

Процедуры могут принимать параметры. Параметры бывают входными - с их помощью в процедуру можно передать некоторые значения. И также параметры бывают выходными - они позволяют вернуть из процедуры некоторое значение.

Например, пусть в базе данных будет следующая таблица Products:

```
USE productsdb;  
CREATE TABLE Products  
(  
    Id INT IDENTITY PRIMARY KEY,  
    ProductName NVARCHAR(30) NOT NULL,  
    Manufacturer NVARCHAR(20) NOT NULL,  
    ProductCount INT DEFAULT 0,  
    Price MONEY NOT NULL  
);
```

Определим процедуру, которая будет добавлять данные в эту таблицу:

```
USE productsdb;  
GO  
CREATE PROCEDURE AddProduct  
    @name NVARCHAR(20),  
    @manufacturer NVARCHAR(20),  
    @count INT,  
    @price MONEY  
AS  
INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)  
VALUES(@name, @manufacturer, @count, @price)
```

После названия процедуры идет список входных параметров, которые определяются также как и переменные - название начинается с символа @, а после названия идет тип переменной. И с помощью команды INSERT значения этих параметров будут передаваться в таблицу Products.

Используем эту процедуру:

```
USE productsdb;  
  
DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20);  
DECLARE @prodCount INT, @price MONEY  
SET @prodName = 'Galaxy C7'  
SET @company = 'Samsung'  
SET @price = 22000  
SET @prodCount = 5  
  
EXEC AddProduct @prodName, @company, @prodCount, @price  
  
SELECT * FROM Products
```

Здесь передаваемые в процедуру значения определяются через переменные. При вызове процедуры ей через запятую передаются значения. При этом значения передаются параметрам процедуры по позиции. Так как первым определен параметр @name, то ему будет передаваться первое значение - значение переменной @prodName. Второму параметру - @manufacturer передается второе значение -

```
1 USE productsdb;
2
3 DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20);
4 DECLARE @prodCount INT, @price MONEY
5 SET @prodName = 'Galaxy C7'
6 SET @company = 'Samsung'
7 SET @price = 22000
8 SET @prodCount = 5
9
10 EXEC AddProduct @prodName, @company, @prodCount, @price
11
12 SELECT * FROM Products
```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	2	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00
4	4	Galaxy S8	Samsung	2	46000,00
5	5	Galaxy S8 Plus	Samsung	1	56000,00
6	6	Mi 5X	Xiaomi	2	26000,00
7	7	OnePlus 5	OnePlus	6	38000,00
8	1004	Galaxy C7	Samsung	5	22000,00

EXEC AddProduct 'Galaxy C7', 'Samsung', 5, 22000

USE productsdb;

```
SET @company = 'Huawei'
```

@price = 18000

Необязательные параметры

GO

```
@count INT = 1
```

AS

```
INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)
VALUES(@name, @manufacturer, @count, @price)
```

При этом необязательные параметры лучше помещать в конце списка параметров процедуры.

```
DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20), @price MONEY
SET @prodName = 'Redmi Note 5A'
SET @company = 'Xiaomi'
SET @price = 22000
```

```
EXEC AddProductWithOptionalCount @prodName, @company, @price
```

```
SELECT * FROM Products
```

И в этом случае для параметра @count в процедуру можно не передавать значение.

Выходные параметры и возвращение результата

Выходные параметры позволяют вернуть из процедуры некоторый результат. Выходные параметры определяются с помощью ключевого слова OUTPUT. Например, определим еще одну процедуру:

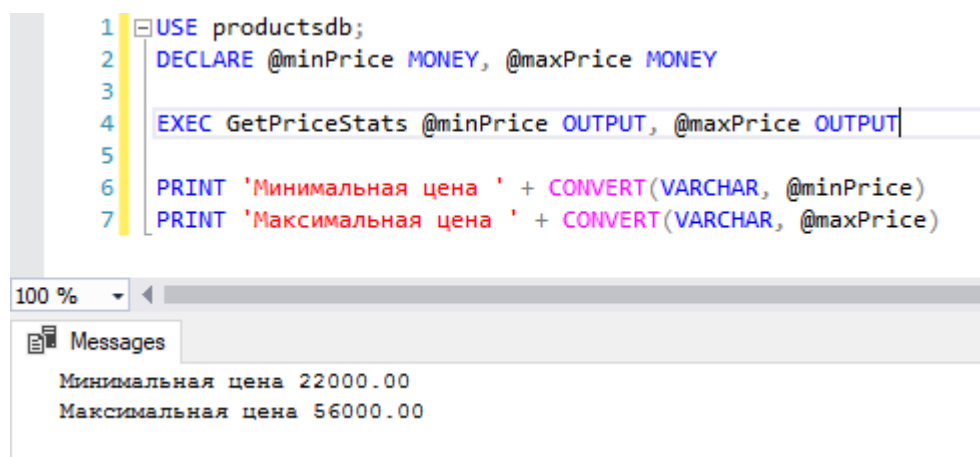
```
USE productsdb;
GO
CREATE PROCEDURE GetPriceStats
    @minPrice MONEY OUTPUT,
    @maxPrice MONEY OUTPUT
AS
SELECT @minPrice = MIN(Price), @maxPrice = MAX(Price)
FROM Products
```

При вызове процедуры для выходных параметров передаются переменные с ключевым словом OUTPUT:

```
USE productsdb;
DECLARE @minPrice MONEY, @maxPrice MONEY

EXEC GetPriceStats @minPrice OUTPUT, @maxPrice OUTPUT

PRINT 'Минимальная цена ' + CONVERT(VARCHAR, @minPrice)
PRINT 'Максимальная цена ' + CONVERT(VARCHAR, @maxPrice)
```



The screenshot shows a SQL query window with the following code:

```
1 USE productsdb;
2 DECLARE @minPrice MONEY, @maxPrice MONEY
3
4 EXEC GetPriceStats @minPrice OUTPUT, @maxPrice OUTPUT
5
6 PRINT 'Минимальная цена ' + CONVERT(VARCHAR, @minPrice)
7 PRINT 'Максимальная цена ' + CONVERT(VARCHAR, @maxPrice)
```

Below the query window, the Messages pane displays the output of the PRINT statements:

```
Минимальная цена 22000.00
Максимальная цена 56000.00
```

Также можно сочетать входные и выходные параметры. Например, определим процедуру, которая добавляет новую строку в таблицу и возвращает ее id:


```
USE productsdb;  
GO
```

```
CREATE PROCEDURE CreateProduct  
    @name NVARCHAR(20),  
    @manufacturer NVARCHAR(20),  
    @count INT,  
    @price MONEY,  
    @id INT OUTPUT  
AS  
    INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)  
    VALUES(@name, @manufacturer, @count, @price)  
    SET @id = @@IDENTITY
```

С помощью глобальной переменной @@IDENTITY можно получить идентификатор добавленной записи.

При вызове этой процедуры ей также по позиции передаются все входные и выходные параметры:

```
USE productsdb;
```

```
DECLARE @id INT
```

```
EXEC CreateProduct 'LG V30', 'LG', 3, 28000, @id OUTPUT
```

```
PRINT @id
```

Возвращение значения

Кроме передачи результата выполнения через выходные параметры хранимая процедура также может возвращать какое-либо значение типа INT с помощью оператора RETURN. Хотя данная возможность во многом нивелирована использованием выходных параметров, через которые можно возвращать результат, тем не менее, если надо вернуть из процедуры одно значение, то вполне можно использовать оператор RETURN.

Например, возвратим среднюю цену на товары:

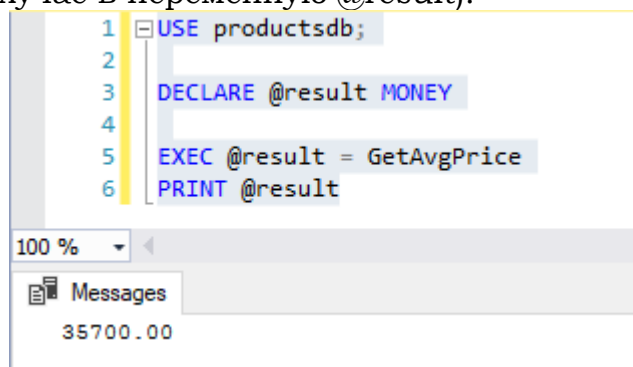
```
USE productsdb;  
GO  
CREATE PROCEDURE GetAvgPrice AS  
DECLARE @avgPrice MONEY  
SELECT @avgPrice = AVG(Price)  
FROM Products  
RETURN @avgPrice;
```

После оператора RETURN указывается возвращаемое значение. В данном случае это значение переменной @avgPrice.

Вызовем данную процедуру:

```
USE productsdb;  
  
DECLARE @result MONEY  
  
EXEC @result = GetAvgPrice  
PRINT @result
```

Для получения результата процедуры ее значение сохраняется в переменную (в данном случае в переменную @result):



Стоит отметить, что RETURN возвращает только целочисленные значения.

Практическая работа № 12

Задание

1. Ознакомиться с практической работой.
2. Создать 3 триггера для своей предметной области с помощью запросов.
3. Оформить отчёт, который должен содержать запросы на создание триггера и результаты их выполнения.
4. Защитить работу, ответив на вопросы по её выполнению.

ТРИГГЕРЫ

Триггеры представляют специальный тип хранимой процедуры, которая вызывается автоматически при выполнении определенного действия над таблицей или представлением, в частности, при добавлении, изменении или удалении данных, то есть при выполнении команд INSERT, UPDATE, DELETE.

Формальное определение триггера:

```
CREATE TRIGGER имя_триггера
ON {имя_таблицы | имя_представления}
{AFTER | INSTEAD OF} [INSERT | UPDATE | DELETE]
AS выражения_sql
```

Для создания триггера применяется выражение CREATE TRIGGER, после которого идет имя триггера. Как правило, имя триггера отражает тип операций и имя таблицы, над которой производится операция.

Каждый триггер ассоциируется с определенной таблицей или представлением, имя которых указывается после слова ON.

Затем устанавливается тип триггера. Мы можем использовать один из двух типов:

AFTER: выполняется после выполнения действия. Определяется только для таблиц.

INSTEAD OF: выполняется вместо действия (то есть по сути действие - добавление, изменение или удаление - вообще не выполняется). Определяется для таблиц и представлений

После типа триггера идет указание операции, для которой определяется триггер: INSERT, UPDATE или DELETE.

Для триггера AFTER можно применять сразу для нескольких действий, например, UPDATE и INSERT. В этом случае операции указываются через запятую. Для триггера INSTEAD OF можно определить только одно действие.

И затем после слова AS идет набор выражений SQL, которые собственно и составляют тело триггера.

Создадим триггер. Допустим, у нас есть база данных productsdb со следующим определением:

```
CREATE DATABASE productdb;
GO

USE productdb;
CREATE TABLE Products
(
    Id INT IDENTITY PRIMARY KEY,
    ProductName NVARCHAR(30) NOT NULL,
    Manufacturer NVARCHAR(20) NOT NULL,
    ProductCount INT DEFAULT 0,
    Price MONEY NOT NULL
);
```

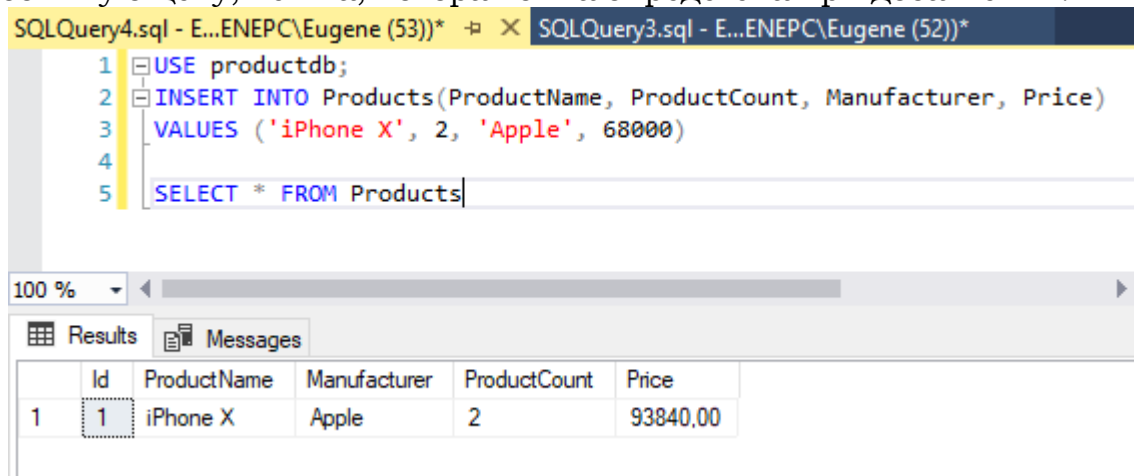
Определим триггер, который будет срабатывать при добавлении и обновлении данных:

```
USE productdb;
GO
CREATE TRIGGER Products_INSERT_UPDATE
ON Products
AFTER INSERT, UPDATE
AS
UPDATE Products
SET Price = Price + Price * 0.38
WHERE Id = (SELECT Id FROM inserted)
```

Допустим, в таблице Products хранятся данные о товарах. Но цена товара нередко содержит различные надбавки типа налога на добавленную стоимость, налога на добавленную коррупцию и так далее. Человек, добавляющий данные, может не знать все эти тонкости с налоговой базой, и он определяет чистую цену. С помощью триггера мы можем поправить цену товара на некоторую величину.

Таким образом, триггер будет срабатывать при любой операции INSERT или UPDATE над таблицей Products. Сам триггер будет изменять цену товара, а для получения того товара, который был добавлен или изменен, находим этот товар по Id. Но какое значение должен иметь Id такой товар? Дело в том, что при добавлении или изменении данные сохраняются в промежуточную таблицу inserted. Она создается автоматически. И из нее мы можем получить данные о добавленных/измененных товарах.

И после добавления товара в таблицу Products в реальности товар будет иметь несколько большую цену, чем та, которая была определена при добавлении:



Удаление триггера

Для удаления триггера необходимо применить команду DROP TRIGGER:

```
DROP TRIGGER Products_INSERT_UPDATE
```

Отключение триггера

Бывает, что мы хотим приостановить действие триггера, но удалять его полностью не хотим. В этом случае его можно временно отключить с помощью команды DISABLE TRIGGER:

```
DISABLE TRIGGER Products_INSERT_UPDATE ON Products
```

А когда триггер понадобится, его можно включить с помощью команды ENABLE TRIGGER:

```
ENABLE TRIGGER Products_INSERT_UPDATE ON Products
```

Триггеры для операций INSERT, UPDATE, DELETE

Для рассмотрения операций с триггерами определим следующую базу данных productsdb:

```
CREATE DATABASE productsdb;
GO
USE productsdb;
CREATE TABLE Products
(
    Id INT IDENTITY PRIMARY KEY,
    ProductName NVARCHAR(30) NOT NULL,
    Manufacturer NVARCHAR(20) NOT NULL,
    ProductCount INT DEFAULT 0,
    Price MONEY NOT NULL
);
CREATE TABLE History
(
    Id INT IDENTITY PRIMARY KEY,
    ProductId INT NOT NULL,
    Operation NVARCHAR(200) NOT NULL,
    CreateAt DATETIME NOT NULL DEFAULT GETDATE(),
);
```

Здесь определены две таблиц: Products - для хранения товаров и History - для хранения истории операций с товарами.

Добавление

При добавлении данных (при выполнении команды INSERT) в триггере мы можем получить добавленные данные из виртуальной таблицы INSERTED.

Определим триггер, который будет срабатывать после добавления:

```
USE productsdb
GO
CREATE TRIGGER Products_INSERT
ON Products
AFTER INSERT
AS
INSERT INTO History (ProductId, Operation)
SELECT Id, 'Добавлен товар ' + ProductName + ' фирма ' + Manufacturer
FROM INSERTED
```

Этот триггер будет добавлять в таблицу History данные о добавлении товара, которые берутся из виртуальной таблицы INSERTED.

Выполним добавление данных в Products и получим данные из таблицы History:

```
USE productsdb;
INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)
VALUES('iPhone X', 'Apple', 2, 79900)
```

```
SELECT * FROM History
```

```

1 USE productsdb;
2 INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)
3 VALUES('iPhone X', 'Apple', 2, 79900)
4
5 SELECT * FROM History

```

100 %

Results Messages

	Id	ProductId	Operation	CreateAt
1	1	2	Добавлен товар iPhone X фирма Apple	2017-11-09 14:05:52.020

Удаление данных

При удалении все удаленные данные помещаются в виртуальную таблицу DELETED:

```

USE productsdb
GO
CREATE TRIGGER Products_DELETE
ON Products
AFTER DELETE
AS
INSERT INTO History (ProductId, Operation)
SELECT Id, 'Удален товар ' + ProductName + ' фирма ' + Manufacturer
FROM DELETED

```

Здесь, как и в случае с предыдущим триггером, помещаем информацию об удаленных товарах в таблицу History.

Выполним команду на удаление:

```

USE productsdb;
DELETE FROM Products
WHERE Id=2

```

SELECT * FROM History

```

1 USE productsdb;
2 DELETE FROM Products
3 WHERE Id=2
4
5 SELECT * FROM History

```

100 %

Results Messages

	Id	ProductId	Operation	CreateAt
1	1	2	Добавлен товар iPhone X фирма Apple	2017-11-09 14:05:52.020
2	2	2	Удален товар iPhone X фирма Apple	2017-11-09 14:44:37.040

Изменение данных

Триггер обновления данных срабатывает при выполнении операции UPDATE. И в таком триггере мы можем использовать две виртуальных таблицы. Таблица INSERTED хранит значения строк после обновления, а таблица DELETED хранит те же строки, но до обновления.

Создадим триггер обновления:

```
USE productsdb
GO
CREATE TRIGGER Products_UPDATE
ON Products
AFTER UPDATE
AS
INSERT INTO History (ProductId, Operation)
SELECT Id, 'Обновлен товар ' + ProductName + ' фирма ' + Manufacturer
FROM INSERTED
```

И при обновлении данных сработает данный триггер:

The screenshot shows a SQL query window with the following code:

```
1 USE productsdb;
2 INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)
3 VALUES('C350', 'Motorola', 10, 2100);
4
5 UPDATE Products SET Manufacturer='Moto'
6 WHERE Manufacturer='Motorola';
7
8 SELECT * FROM History
```

Below the query window, the 'Results' tab is active, displaying a table with the following data:

	Id	ProductId	Operation	CreateAt
1	1	2	Добавлен товар iPhone X фирма Apple	2017-11-09 14:05:52.020
2	2	2	Удален товар iPhone X фирма Apple	2017-11-09 14:44:37.040
3	3	3	Добавлен товар C350 фирма Motorola	2017-11-09 15:00:18.130
4	4	3	Обновлен товар C350 фирма Moto	2017-11-09 15:02:04.733

Триггер INSTEAD OF

Триггер INSTEAD OF срабатывает вместо операции с данными. Он определяется в принципе также, как триггер AFTER, за тем исключением, что он может определяться только для одной операции - INSERT, DELETE или UPDATE. И также он может применяться как для таблиц, так и для представлений (триггер AFTER применяется только для таблиц).

Например, создадим следующие базу данных и таблицу:

```
CREATE DATABASE prods;
GO
USE prods;
CREATE TABLE Products
(
    Id INT IDENTITY PRIMARY KEY,
    ProductName NVARCHAR(30) NOT NULL,
    Manufacturer NVARCHAR(20) NOT NULL,
    Price MONEY NOT NULL,
    IsDeleted BIT NULL
);
```

Здесь таблица содержит столбец IsDeleted, который указывает, удалена ли запись. То есть вместо жесткого удаления полностью из базы данных мы хотим выполнить мягкое удаление, при котором запись остается в базе данных.

Определим триггер для удаления записи:

```
USE prods
```

```
GO
```

```
CREATE TRIGGER products_delete
```

```
ON Products
```

```
INSTEAD OF DELETE
```

```
AS
```

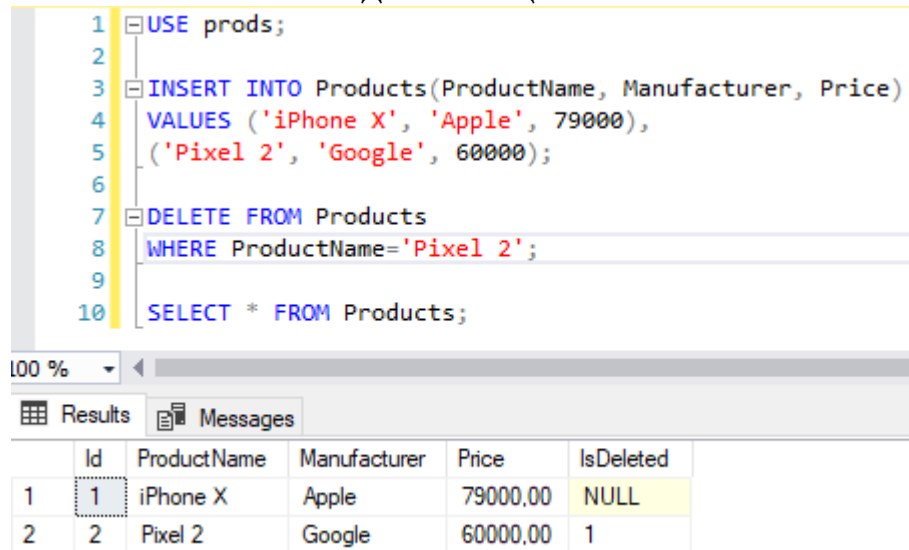
```
UPDATE Products
```

```
SET IsDeleted = 1
```

```
WHERE ID =(SELECT Id FROM deleted)
```

Добавим некоторые данные в таблицу и выполним удаление из нее:

Таким образом, удаляемые записи на самом деле не будут удаляться, просто у них будет устанавливаться значение для столбца IsDeleted:



The screenshot shows a SQL script in the 'Script' tab and its results in the 'Results' tab. The script performs an insert, a delete, and a select. The results table shows two rows: one for 'iPhone X' with IsDeleted as NULL, and one for 'Pixel 2' with IsDeleted as 1.

```
1 USE prods;
2
3 INSERT INTO Products(ProductName, Manufacturer, Price)
4 VALUES ('iPhone X', 'Apple', 79000),
5 ('Pixel 2', 'Google', 60000);
6
7 DELETE FROM Products
8 WHERE ProductName='Pixel 2';
9
10 SELECT * FROM Products;
```

	Id	ProductName	Manufacturer	Price	IsDeleted
1	1	iPhone X	Apple	79000,00	NULL
2	2	Pixel 2	Google	60000,00	1