**Тема: Язык SQL. Хранимые процедуры и триггеры.**

**Цели урока:**

1. Познакомить учащихся с основами программирования на SQL
2. Познакомить учащихся с особенностями создания хранимых процедур и триггеров на языке SQL**.**
3. Формировать у учащихся культуру слушания и выполнения действий.

**Тип урока:** Урок сообщения новых знаний.

**Метод обучения:** Проблемное изложение нового материала.

**Оборудование:** проектор, белая доска.

**Источники:**

1. Пакеты. Команда GO. URL: https://metanit.com/sql/sqlserver/3.7.php.

2. Переменные и управляющие конструкции. URL: https://metanit.com/sql/sqlserver/9.1.php.

3. Хранимые процедуры. URL: https://metanit.com/sql/sqlserver/11.1.php.

4. Триггеры. URL: https://metanit.com/sql/sqlserver/12.1.php.

**ПЛАН УРОКА.**

1. Организационный момент (приветствие учащихся, проверить готовность учащихся к уроку).
2. Объяснение нового материала (лекция).
3. Ответы на вопросы учащихся по теме урока.
4. Подведение итогов урока.
5. Дать задание на дом, назначить дежурных.

**ХОД УРОКА.**

1. Лекция по новой теме:

**Пакеты. Команда GO**

Ранее сначала создавалась БД, а затем в эту БД добавлялась таблица с помощью отдельных команд SQL. Но можно сразу совместить в одном скрипте несколько команд. В этом случае отдельные наборы команд называются пакетами.

Каждый пакет состоит из одного или нескольких SQL-выражений, которые выполняются как оно целое. В качестве сигнала завершения пакета и выполнения его выражений служит команда **GO**.

Смысл разделения SQL-выражений на пакеты состоит в том, что одни выражения должны успешно выполниться до запуска других выражений. Например, при добавлении таблиц мы должны быть уверены, что была создана база данных, в которой мы собираемся создать таблицы.

Например, определим следующий скрипт:

|  |
| --- |
| CREATE DATABASE internetstore;  GO  USE internetstore;  CREATE TABLE Customers (      Id INT PRIMARY KEY IDENTITY,      Age INT DEFAULT 18,      FirstName NVARCHAR(20) NOT NULL,      LastName NVARCHAR(20) NOT NULL,      Email VARCHAR(30) UNIQUE,      Phone VARCHAR(20) UNIQUE  );  CREATE TABLE Orders (      Id INT PRIMARY KEY IDENTITY,      CustomerId INT,      CreatedAt DATE,      FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE CASCADE  ); |

Вначале создается бд internetstore. Затем идет команда **GO**, которая сигнализирует, что можно выполнять следующий пакет выражений. И далее выполняется второй пакет, который добавляет в нее две таблицы - Customers и Orders.

**Переменные** **в T-SQL**

Переменная представляет именованный объект, который хранит некоторое значение. Для определения переменных применяется выражение **DECLARE**, после которого указывается название и тип переменной. При этом название локальной переменной должно начинаться с символа **@**:

|  |
| --- |
| DECLARE @название\_переменной тип\_данных |

Например, определим переменную **name**, которая будет иметь тип **NVARCHAR**:

|  |
| --- |
| DECLARE @name NVARCHAR(20) |

Также можно определить через запятую сразу несколько переменных:

|  |
| --- |
| DECLARE @name NVARCHAR(20), @age INT |

С помощью выражения **SET** можно присвоить переменной некоторое значение:

|  |
| --- |
| DECLARE @name NVARCHAR(20), @age INT;  SET @name='Tom';  SET @age = 18; |

Так как @name предоставляет тип NVARCHAR, то есть строку, то этой переменной соответственно и присваивается строка. А переменной @age присваивается число, так как она представляет тип INT.

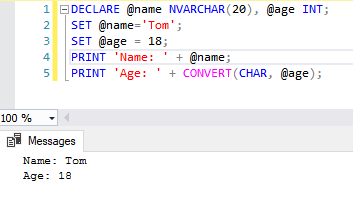
Выражение **PRINT** возвращает сообщение клиенту. Например:

|  |
| --- |
| PRINT 'Hello World' |

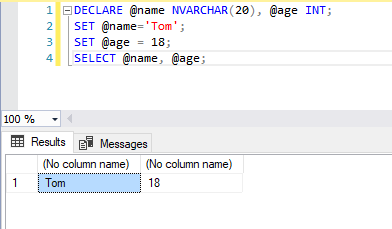
И с его помощью мы можем вывести значение переменной:

|  |
| --- |
| DECLARE @name NVARCHAR(20), @age INT;  SET @name='Tom';  SET @age = 18;  PRINT 'Name: ' + @name;  PRINT 'Age: ' + CONVERT(CHAR, @age); |

При выполнении скрипта внизу SQL Server Management Studio отобразится значение переменных:



Также можно использовать для получения значения команду **SELECT**:



**Переменные в запросах**

Через переменные мы можем передавать данные в запросы. И также мы можем получать данные, которые являются результатом запросов, в переменные. Например, при выборке из таблиц с помощью команды **SELECT** мы можем извлекать данные в переменную с помощью следующего синтаксиса:

|  |
| --- |
| SELECT @переменная\_1 = спецификация\_столбца\_1,         @переменная\_2 = спецификация\_столбца\_2,          ......................................         @переменная\_N = спецификация\_столбца\_N |

Кроме того, в выражении **SET** значение, присваиваемое переменной, также может быть результатом команды **SELECT**.

Например, пусть у нас будут следующие таблицы в базе данных **productsdb**:

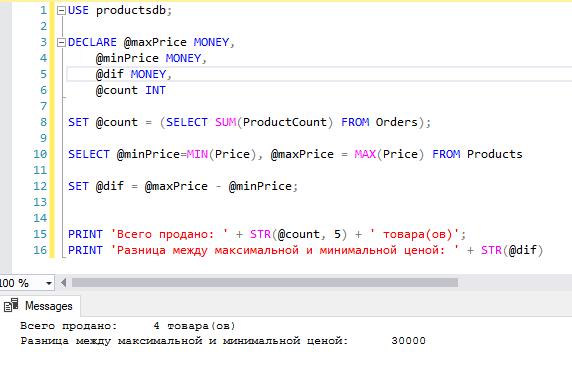
|  |
| --- |
| CREATE DATABASE productsdb;  GO  USE productsdb;  CREATE TABLE Products (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price MONEY NOT NULL  );  CREATE TABLE Customers (      Id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

Используем переменные при извлечении данных:

|  |
| --- |
| DECLARE @maxPrice MONEY,      @minPrice MONEY,      @dif MONEY,      @count INT  SET @count = (SELECT SUM(ProductCount) FROM Orders);  SELECT @minPrice=MIN(Price), @maxPrice = MAX(Price) FROM Products  SET @dif = @maxPrice - @minPrice;  PRINT 'Всего продано: ' + STR(@count, 5) + ' товарa(ов)';  PRINT 'Разница между максимальной и минимальной ценой: ' + STR(@dif) |

В данном случае переменная **@count** будет содержать сумму всех значений из столбца **ProductCount** таблицы **Orders**, то есть общее количество проданных товаров.

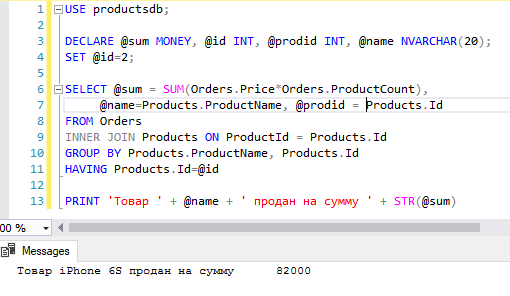
Переменные **@min** и **@max** хранят соответственно минимальное и максимальное значения столбца **Price** из таблицы **Products**, а переменная **@dif** - разницу между этими значениями. И подобно простым значениям, переменные также могут участвовать в операциях.



Другой пример:

|  |
| --- |
| DECLARE @sum MONEY, @id INT, @prodid INT, @name NVARCHAR(20);  SET @id=2;  SELECT @sum = SUM(Orders.Price\*Orders.ProductCount),       @name=Products.ProductName, @prodid = Products.Id  FROM Orders INNER JOIN Products ON ProductId = Products.Id  GROUP BY Products.ProductName, Products.Id  HAVING Products.Id=@id  PRINT 'Товар ' + @name + ' продан на сумму ' + STR(@sum) |

Здесь извлекаемые данные из двух таблиц **Products** и **Orders** группируются по столбцам **Id** и **ProductName** из таблицы **Products**. Затем данные фильтруются по столбцу **Id** из **Products**. А извлеченные данные попадают в переменные **@sum**, **@name**, **@prodid**.

****

**Управляющие конструкции в T-SQL**

**Условные выражения**

Для выполнения действий по условию используется выражение **IF ... ELSE**. SQL Server вычисляет выражение после ключевого слова **IF** и, если оно истинно, то выполняются инструкции после ключевого слова **IF**. Если условие ложно, то выполняются инструкции после ключевого слова **ELSE**.

IF условие

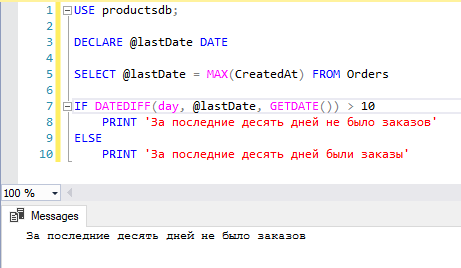
    инструкция

ELSE

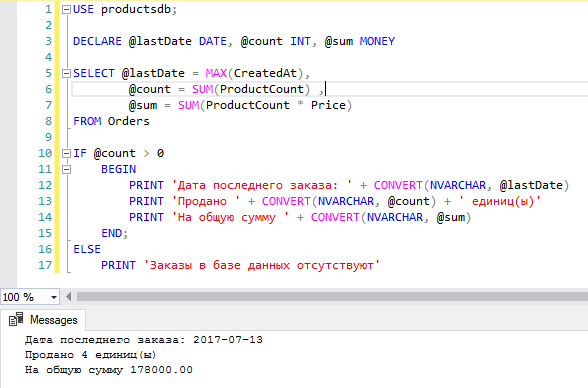
    инструкция

Выражение **ELSE** является необязательным, и его можно опускать.

Рассмотрим ранее созданную в базе данных **productsdb** таблицу **Orders**, которая представляет заказы. В ней есть столбец **CreatedAt**, содержащий дату заказов. Узнаем, были ли заказы за последние 10 дней:



Если после **IF** или **ELSE** идут две и более инструкций, то они заключаются в блок **BEGIN...END**:



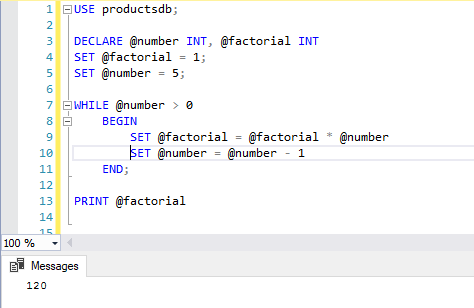
## Циклы

Для выполнения повторяющихся операций в T-SQL применяются циклы. В частности, есть цикл **WHILE**. Этот цикл выполняет определенные действия, пока некоторое условие истинно.

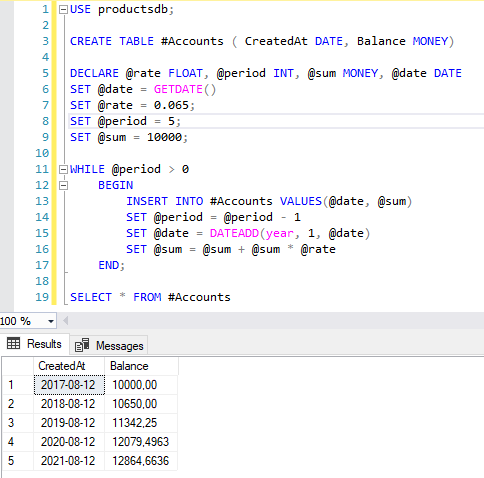
|  |
| --- |
| WHILE условие      инструкция |

Если в блоке **WHILE** необходимо разместить несколько инструкций, то все они помещаются в блок **BEGIN...END**.

Например, вычислим факториал числа. В данном случае, пока переменная **@number** не будет равна 0, будет продолжаться цикл **WHILE**. Так как **@number** равна 5, то цикл сделает пять проходов. Каждый проход цикла называется итерацией. В каждой итерации будет переустанавливаться значение переменных **@factorial** и **@number**.



Другой пример – рассчитаем баланс счета через несколько лет с учетом процентной ставки. Здесь создается временная таблица **#Accounts**, в которую добавляется в цикле пять строк с данными.

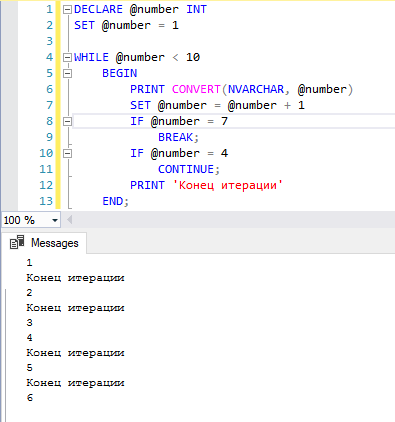


## Операторы BREAK и CONTINUE

Оператор **BREAK** позволяет завершить цикл, а оператор **CONTINUE** – перейти к новой итерации.

В следующем примере, когда переменная **@number** станет равна 4, то с помощью оператора **CONTINUE** произойдет переход к новой итерации, поэтому следующая строка **PRINT 'Конец итерации'** не будет выполняться, хотя цикл продолжится.

Когда переменная **@number** станет равна 7, то оператор **BREAK** произведет выход из цикла, и он завершится.



**Обработка ошибок**

Для обработки ошибок в T-SQL применяется конструкция **TRY...CATCH**. Она имеет следующий формальный синтаксис:

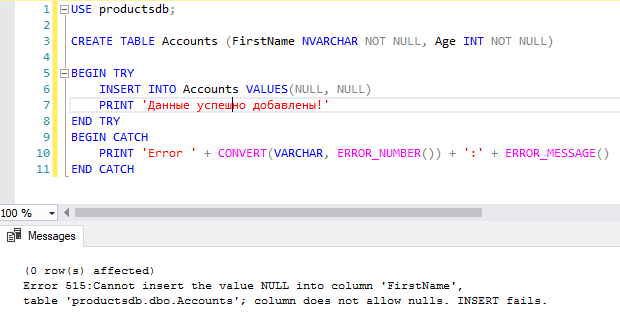
|  |
| --- |
| BEGIN TRY      инструкции  END TRY  BEGIN CATCH      инструкции  END CATCH |

Между выражениями **BEGIN TRY** и **END TRY** помещаются инструкции, которые потенциально могут вызвать ошибку, например, какой-нибудь запрос. И если в этом блоке **TRY** возникнет ошибка, то управление передается в блок **CATCH**, где можно обработать ошибку.

В блоке **CATCH** для обработки ошибки можно использовать ряд функций:

* **ERROR\_NUMBER()**: возвращает номер ошибки.
* **ERROR\_MESSAGE()**: возвращает сообщение об ошибке.
* **ERROR\_SEVERITY()**: возвращает степень серьезности ошибки. Степень серьезности представляет числовое значение. И если оно равно 10 и меньше, то такая ошибка рассматривается как предупреждение и не обрабатывается конструкцией TRY...CATCH. Если же это значение равно 20 и выше, то такая ошибка приводит к закрытию подключения к базе данных, если она не обрабатывается конструкцией TRY...CATCH.
* **ERROR\_STATE()**: возвращает состояние ошибки.

Например, добавим в таблицу данные, которые не соответствуют ограничениям столбцов: в данном случае для столбцов таблицы вставляются недопустимые данные - значения NULL, поэтому обработка программы перейдет к блоку CATCH:



**Хранимые процедуры**

## Создание процедур

Нередко операция с данными представляет набор инструкций, которые необходимо выполнить в определенной последовательности. Например, при добавлении информации о покупке 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 представляет 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.

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

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

Например, создадим хранимую процедуру для извлечения данных из таблицы **Products**, которая хранит данные о товарах:

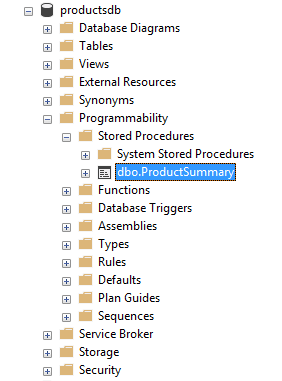
|  |
| --- |
| USE productsdb;  GO  CREATE PROCEDURE ProductSummary AS  BEGIN  SELECT ProductName AS Product, Manufacturer, Price  FROM Products  END; |

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**.

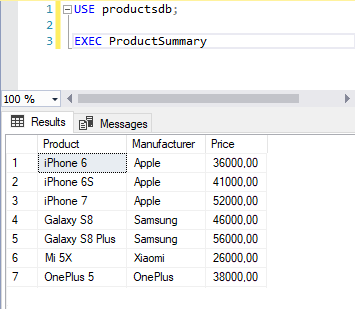
После первого выполнения процедуры ее можно увидеть в SQL Server Management Studio в Обозревателе объектов в узле базы данных в подузле Programmability -> Stored Procedures (Программирование->Хранимые процедуры):



И мы сможем управлять процедурой также и через визуальный интерфейс.

## Выполнение процедуры

Для выполнения хранимой процедуры вызывается команда **EXEC** или **EXECUTE**:

****

**Удаление процедуры**

Для удаления процедуры применяется команда **DROP PROCEDURE**:

|  |
| --- |
| DROP PROCEDURE ProductSummary |

**Параметры в процедурах**

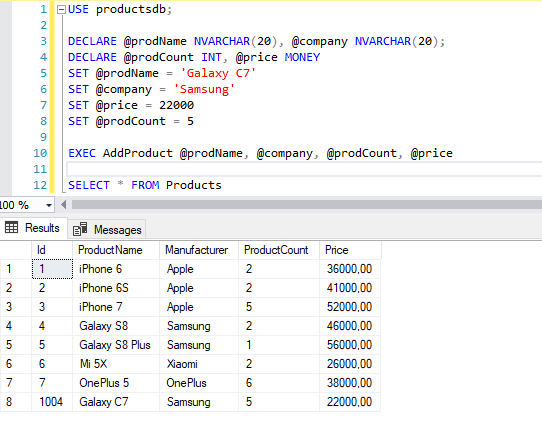
Процедуры могут принимать параметры. Параметры бывают входными – с их помощью в процедуру можно передать некоторые значения. И также параметры бывают выходными – они позволяют возвратить из процедуры некоторое значение.

Например, определим процедуру, которая будет добавлять данные в таблицу **Products**:

|  |
| --- |
| 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:



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 - значение переменной **@company** и так далее. Главное, чтобы между передаваемыми значениями и параметрами процедуры было соответствие по типу данных.

Также можно передавать непосредственно значения:

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

Также значения параметрам процедуры можно передавать по имени:

|  |
| --- |
| DECLARE @prodName NVARCHAR(20), @company NVARCHAR(20);  SET @prodName = 'Honor 9'  SET @company = 'Huawei'  EXEC AddProduct @name = @prodName,                  @manufacturer=@company,                  @count = 3,                  @price = 18000 |

При передаче параметров по имени параметру процедуры присваивается некоторое значение.

**Необязательные параметры**

Параметры можно отмечать как необязательные, присваивая им некоторое значение по умолчанию. Например, мы можем автоматически устанавливать для количества товара значение 1, если соответствующее значение не передано в процедуру:

|  |
| --- |
| CREATE PROCEDURE AddProductWithOptionalCount      @name NVARCHAR(20),      @manufacturer NVARCHAR(20),      @price MONEY,      @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 |

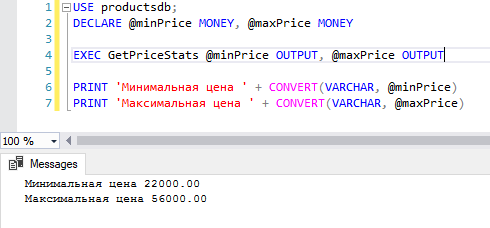
В этом случае для параметра **@count** в процедуру можно не передавать значение.

## Выходные параметры и возвращение результата

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

|  |
| --- |
| 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**:



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

|  |
| --- |
| 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:

|  |
| --- |
| DECLARE @id INT  EXEC CreateProduct 'LG V30', 'LG', 3, 28000, @id OUTPUT  PRINT @id |

## Возвращение значения

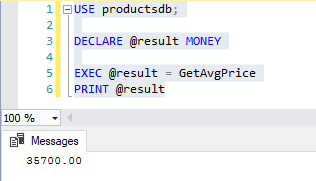
Кроме передачи результата выполнения через выходные параметры хранимая процедура также может возвращать какое-либо значение типа **INT** с помощью оператора **RETURN**. Хотя данная возможность во многом нивелирована использованием выходных параметров, через которые можно возвращать результат, тем не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

Например, возвратим среднюю цену на товары:

|  |
| --- |
| 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: для получения результата процедуры ее значение сохраняется в переменную (в данном случае в переменную **@result**):



## 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. Можно 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**.

Триггер типа **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.

После слова **AS** идет набор выражений SQL, которые составляют тело триггера.

Создадим триггер, который будет срабатывать при добавлении и обновлении данных в таблице **Products**:

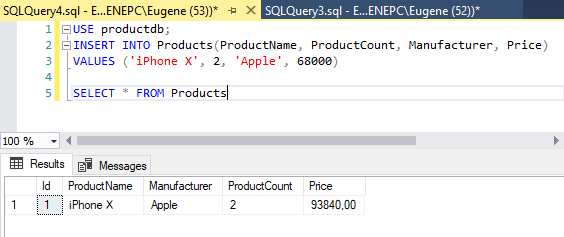
|  |
| --- |
| CREATE TRIGGER Products\_INSERT\_UPDATE  ON Products  AFTER INSERT, UPDATE  AS  UPDATE Products  SET Price = Price + Price \* 0.38  WHERE Id = (SELECT Id FROM inserted) |

Допустим, в таблице **Products** хранятся данные о товарах. Но цена товара нередко содержит различные надбавки типа налога на добавленную стоимость. 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 в таблице **inserted** по **Id**.

При добавлении или изменении данные сохраняются в промежуточную таблицу **inserted**. Она создается автоматически. И из нее можно 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

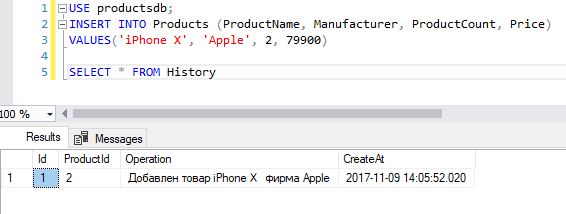
При добавлении данных (при выполнении команды **INSERT**) в триггере можно получить добавленные данные из виртуальной таблицы **INSERTED**.

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**:



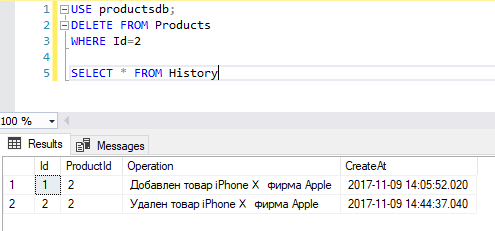
## Удаление данных

При удалении все удаленные данные помещаются в виртуальную таблицу **DELETED**:

|  |
| --- |
| 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:



## Изменение данных

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

Создадим триггер обновления:

|  |
| --- |
| 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:

## 

## Триггер INSTEAD OF

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

Например, создадим следующие базу данных и таблицу:

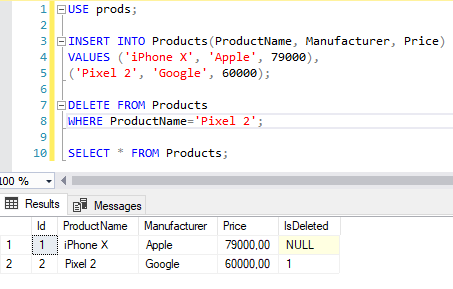
|  |
| --- |
| CREATE DATABASE prods;  GO  USE prods;  CREATE TABLE Products (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      Price MONEY NOT NULL,      IsDeleted BIT NULL  ); |

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:

|  |
| --- |
| 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, при этом удаляемые записи на самом деле не будут удаляться, просто у них будет устанавливаться значение для столбца IsDeleted:



1. Ответы на вопросы учащихся по теме урока.
2. Дать домашнее задание.
3. Д/з: 1. Лекция.