**Лабораторная работа.**

**Разработка пользовательских функций, хранимых процедур, триггеров, курсоров, инструкций управления транзакциями.**

**1. Цель работы:** изучение возможностей основных программных модулей (функций, хранимых процедур) СУБД MS SQL Server и управления процессами изменения данных.

**2.** **Краткие теоретические сведения.**

**2.1. Переменные и управляющие конструкции**

**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.

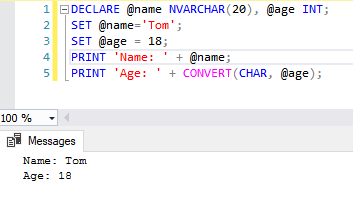
Выражение PRINT возвращает сообщение клиенту. Например:

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

И с его помощью мы можем вывести значение переменной:

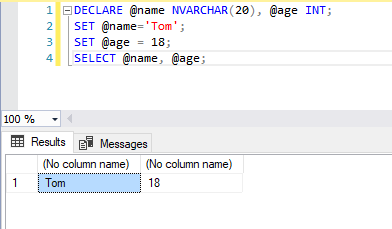
|  |
| --- |
| DECLARE @name NVARCHAR(20), @age INT;  SET @name='Tom';  SET @age = 18;  PRINT 'Name: ' + @name;  PRINT 'Age: ' + CONVERT(CHAR, @age); |

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



Также можно использовать для получения значения команду SELECT:

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



**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:

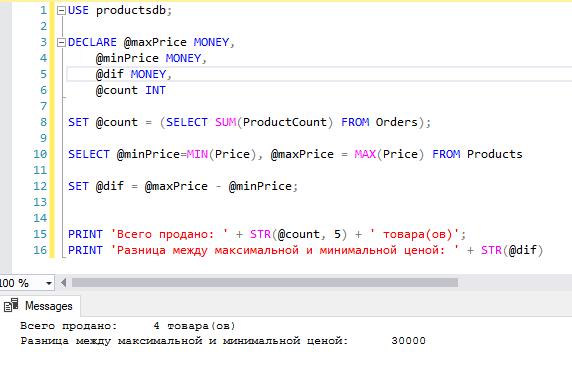
|  |
| --- |
| CREATE TABLE Products  (      Id INT IDENTITY PRIMARY KEY,      ProductName NVARCHAR(30) NOT NULL,      Manufacturer NVARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price MONEY NOT NULL  );  CREATE TABLE Customers  (      Id INT IDENTITY PRIMARY KEY,      FirstName NVARCHAR(30)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

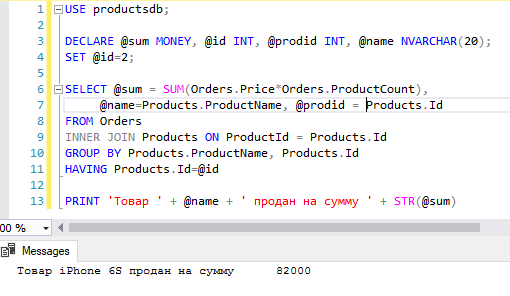
Переменные @min и @max хранят соответственно минимальное и максимальное значения столбца Price из таблицы Products, а переменная @dif - разницу между этими значениями. И подобно простым значениям, переменные также могут участвовать в операциях.



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.



## Циклы

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

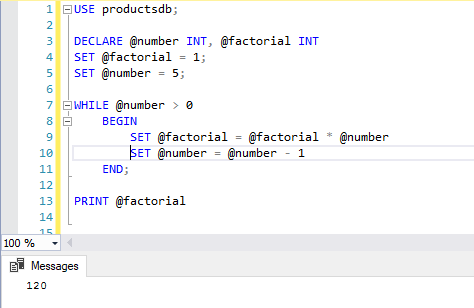
|  |
| --- |
| WHILE условие      {инструкция|BEGIN...END} |

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

Например, вычислим факториал числа:

|  |
| --- |
| DECLARE @number INT, @factorial INT  SET @factorial = 1;  SET @number = 5;    WHILE @number > 0      BEGIN          SET @factorial = @factorial \* @number          SET @number = @number - 1      END;    PRINT @factorial |

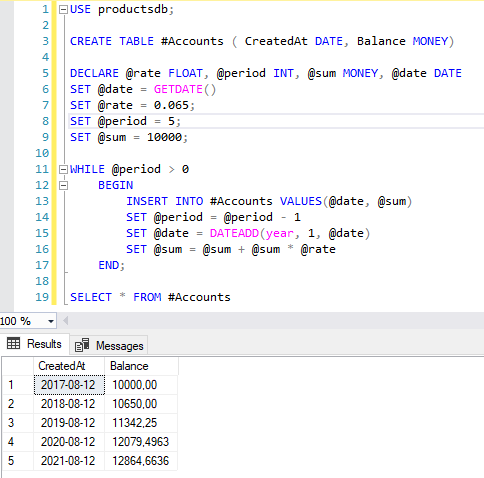
То есть в данном случае пока переменная @number не будет равна 0, будет продолжаться цикл WHILE. Так как @number равна 5, то цикл сделает пять проходов. Каждый проход цикла называется итерацией. В каждой итерации будет переустанавливаться значение переменных @factorial и @number.



Другой пример - рассчитаем баланс счета через несколько лет с учетом процентной ставки:

|  |
| --- |
| USE productsdb;    CREATE TABLE #Accounts ( CreatedAt DATE, Balance MONEY)    DECLARE @rate FLOAT, @period INT, @sum MONEY, @date DATE  SET @date = GETDATE()  SET @rate = 0.065;  SET @period = 5;  SET @sum = 10000;    WHILE @period > 0      BEGIN          INSERT INTO #Accounts VALUES(@date, @sum)          SET @period = @period - 1          SET @date = DATEADD(year, 1, @date)          SET @sum = @sum + @sum \* @rate      END;    SELECT \* FROM #Accounts |

Здесь создается временная таблица #Accounts, в которую добавляется в цикле пять строк с данными.



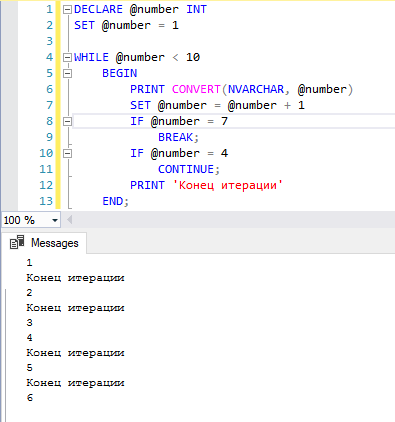
### Операторы BREAK и CONTINUE

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

|  |
| --- |
| DECLARE @number INT  SET @number = 1    WHILE @number < 10      BEGIN          PRINT CONVERT(NVARCHAR, @number)          SET @number = @number + 1          IF @number = 7              BREAK;          IF @number = 4              CONTINUE;          PRINT 'Конец итерации'      END; |

Когда переменная @number станет равна 4, то с помощью оператора CONTINUE произойдет переход к новой итерации, поэтому последующая 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.



**2.2. Функции пользователя**

При реализации на языке SQL сложных алгоритмов, которые могут потребоваться более одного раза, сразу встает вопрос о сохранении разработанного кода для дальнейшего применения. Эту задачу можно было бы реализовать с помощью хранимых процедур, однако их архитектура не позволяет использовать процедуры непосредственно в выражениях, т.к. они требуют промежуточного присвоения возвращенного значения переменной, которая затем и указывается в выражении. Естественно, подобный метод применения программного кода не слишком удобен. Многие разработчики уже давно хотели иметь возможность вызова разработанных алгоритмов непосредственно в выражениях.

**Функции пользователя** представляют собой самостоятельные объекты базы данных, такие, например, как хранимые процедуры или триггеры. Функция пользователя располагается в определенной базе данных и доступна только в ее контексте.

В SQL Server имеются следующие классы функций пользователя:

* **Scalar** – функции возвращают обычное скалярное значение, каждая может включать множество команд, объединяемых в один блок с помощью конструкции BEGIN...END;
* **Inline** – функции содержат всего одну команду SELECT и возвращают пользователю набор данных в виде значения типа данных TABLE;
* **Multi-statement** – функции также возвращают пользователю значение типа данных TABLE, содержащее набор данных, однако в теле функции находится множество команд SQL (INSERT, UPDATE и т.д.). Именно с их помощью и формируется набор данных, который должен быть возвращен после выполнения функции.

Пользовательские функции сходны с хранимыми процедурами, но, в отличие от них, могут применяться в запросах так же, как и системные встроенные функции. Пользовательские функции, возвращающие таблицы, могут стать альтернативой представлениям. Представления ограничены одним выражением SELECT, а пользовательские функции способны включать дополнительные выражения, что позволяет создавать более сложные и мощные конструкции.

**Функции Scalar**

Создание и изменение функции данного типа выполняется с помощью команды:

<определение\_скаляр\_функции>::=

{CREATE | ALTER } **FUNCTION** [владелец.]

имя\_функции

( [ { @имя\_параметра скаляр\_тип\_данных

[=default]}[,...n]])

**RETURNS** скаляр\_тип\_данных

[WITH {ENCRYPTION | SCHEMABINDING}

[,...n] ]

[AS]

**BEGIN**

<тело\_функции>

**RETURN** скаляр\_выражение

**END**

Рассмотрим назначение параметров команды.

Функция может содержать один или несколько входных параметров либо не содержать ни одного. Каждый параметр должен иметь уникальное в пределах создаваемой функции имя и начинаться с символа "@". После имени указывается тип данных параметра. Дополнительно можно указать значение, которое будет автоматически присваиваться параметру (DEFAULT), если пользователь явно не указал значение соответствующего параметра при вызове функции.

С помощью конструкции RETURNS скаляр\_тип\_данных указывается, какой тип данных будет иметь возвращаемое функцией значение.

Дополнительные параметры, с которыми должна быть создана функция, могут быть указаны посредством ключевого слова WITH. Благодаря ключевому слову ENCRYPTION код команды, используемый для создания функции, будет зашифрован, и никто не сможет просмотреть его. Эта возможность позволяет скрыть логику работы функции. Кроме того, в теле функции может выполняться обращение к различным объектам базы данных, а потому изменение или удаление соответствующих объектов может привести к нарушению работы функции. Чтобы избежать этого, требуется запретить внесение изменений, указав при создании этой функции ключевое слово SCHEMABINDING.

Между ключевыми словами BEGIN...END указывается набор команд, они и будут являться телом функции.

Когда в ходе выполнения кода функции встречается ключевое слово RETURN, выполнение функции завершается и как результат ее вычисления возвращается значение, указанное непосредственно после слова RETURN. Отметим, что в теле функции разрешается использование множества команд RETURN, которые могут возвращать различные значения. В качестве возвращаемого значения допускаются как обычные константы, так и сложные выражения. Единственное условие – тип данных возвращаемого значения должен совпадать с типом данных, указанным после ключевого слова RETURNS.

**Пример**. Создать и применить функцию скалярного типа для вычисления суммарного количества товара, поступившего за определенную дату. Владелец функции – пользователь с именем user1.

CREATE FUNCTION

user1.sales(@data DATETIME)

RETURNS INT

AS

BEGIN

DECLARE @c INT

SET @c=(SELECT SUM(количество)

FROM Сделка

WHERE дата=@data)

RETURN (@c)

END

В качестве входного параметра используется дата. Функция возвращает значение целого типа, полученное из оператора SELECT путем суммирования количества товара из таблицы Сделка. Условием отбора записей для суммирования является равенство даты сделки значению входного параметра функции.

Проиллюстрируем обращение к функции пользователя: определим количество товара, поступившего за 02.11.01:

DECLARE @kol INT

SET @kol=user1.sales ('02.11.01')

SELECT @kol

**Функции Inline**

Создание и изменение функции этого типа выполняется с помощью команды:

<определение\_табл\_функции>::=

{CREATE | ALTER } **FUNCTION** [владелец.]

**имя\_функции**

( [ { @имя\_параметра скаляр\_тип\_данных

[=default]}[,...n]])

**RETURNS TABLE**

[ WITH {ENCRYPTION | SCHEMABINDING}

[,...n] ]

[AS]

**RETURN** [(] SELECT\_оператор [)]

Основная часть параметров, используемых при создании табличных функций, аналогична параметрам скалярной функции. Тем не менее, создание табличных функций имеет свою специфику.

После ключевого слова RETURNS всегда должно указываться ключевое слово TABLE. Таким образом, функция данного типа должна строго возвращать значение типа данных TABLE. Структура возвращаемого значения типа TABLE не указывается явно при описании собственно типа данных. Вместо этого сервер будет автоматически использовать для возвращаемого значения TABLE структуру, возвращаемую запросом SELECT, который является единственной командой функции.

Особенность функции данного типа заключается в том, что структура значения TABLE создается автоматически в ходе выполнения запроса, а не указывается явно при определении типа после ключевого слова RETURNS.

Возвращаемое функцией значение типа TABLE может быть использовано непосредственно в запросе, т.е. в разделе FROM.

**Пример**. Создать и применить функцию табличного типа для определения двух наименований товара с наибольшим остатком.

CREATE FUNCTION user1.itog()

RETURNS TABLE

AS

RETURN (SELECT TOP 2 Товар.Название

FROM Товар INNER JOIN Склад

ON Товар.КодТовара=Склад.КодТовара

ORDER BY Склад.Остаток DESC)

Использовать функцию для получения двух наименований товара с наибольшим остатком можно следующим образом:

SELECT Название

FROM user1.itog()

**Функции Multi-statement**

Создание и изменение функций типа Multi-statement выполняется с помощью следующей команды:

<определение\_мульти\_функции>::=

{CREATE | ALTER }**FUNCTION** [владелец.]

**имя\_функции**

( [ { @имя\_параметра скаляр\_тип\_данных

[=default]}[,...n]])

**RETURNS @имя\_параметра TABLE**

**<определение\_таблицы>**

[WITH {ENCRYPTION | SCHEMABINDING}

[,...n] ]

[AS]

**BEGIN**

<тело\_функции>

**RETURN**

**END**

Использование большей части параметров рассматривалось при описании предыдущих функций.

Отметим, что функции данного типа, как и табличные, возвращают значение типа TABLE. Однако, в отличие от табличных функций, при создании функций Multi-statement необходимо явно задать структуру возвращаемого значения. Она указывается непосредственно после ключевого слова TABLE и, таким образом, является частью определения возвращаемого типа данных. Синтаксис конструкции <определение\_таблицы> полностью соответствует одноименным структурам, используемым при создании обычных таблиц с помощью команды CREATE TABLE.

Набор возвращаемых данных должен формироваться с помощью команд INSERT, выполняемых в теле функции. Кроме того, в теле функции допускается использование различных конструкций языка SQL, которые могут контролировать значения, размещаемые в выходном наборе строк. При работе с командой INSERT требуется явно указать имя того объекта, куда необходимо вставить строки. Поэтому в функциях типа Multi-statement, в отличие от табличных, необходимо присвоить какое-то имя объекту с типом данных TABLE – оно и указывается как возвращаемое значение.

Завершение работы функции происходит в двух случаях: если возникают ошибки выполнения и если появляется ключевое слово RETURN. В отличие от функций скалярного типа, при использовании команды RETURN не нужно указывать возвращаемое значение. Сервер автоматически возвратит набор данных типа TABLE, имя и структура которого была указана после ключевого слова RETURNS. В теле функции может быть указано более одной команды RETURN.

Необходимо отметить, что работа функции завершается только при наличии команды RETURN. Это утверждение верно и в том случае, когда речь идет о достижении конца тела функции – самой последней командой должна быть команда RETURN.

**Пример**. Создать и применить функцию (типа multi-statement), которая для некоторого сотрудника выводит список всех его подчиненных (подчиненных как непосредственно ему, так и опосредствованно через других сотрудников).

Список сотрудников с указанием каждого руководителя представлен в таблице emp\_mgr со следующей структурой:

CREATE TABLE emp\_mgr

(emp CHAR(2) PRIMARY KEY,-- сотрудник

mgr CHAR(2)) -- руководитель

Пример данных в таблице emp\_mgr показан ниже. Для упрощения иллюстрации имена сотрудников и их начальников представлены буквами латинского алфавита. У директора организации начальника нет ( NULL ).

emp mgr

---------

a NULL

b a

c a

d a

e f

f b

g b

i c

k d

CREATE FUNCTION fn\_findReports(@id\_emp CHAR(2))

RETURNS @report TABLE(empid CHAR(2)PRIMARY KEY, mgrid CHAR(2))

AS

BEGIN

DECLARE @r INT

DECLARE @t TABLE(empid CHAR(2)PRIMARY KEY, mgrid CHAR(2), pr INT DEFAULT 0)

INSERT @t SELECT emp,mgr,0

FROM emp\_mgr

WHERE emp=@id\_emp

SET @r=@@ROWCOUNT

WHILE @r>0

BEGIN

UPDATE @t SET pr=1 WHERE pr=0

INSERT @t SELECT e.emp, e.mgr,0

FROM emp\_mgr e, @t t

WHERE e.mgr=t.empid

AND t.pr=1

SET @r=@@ROWCOUNT

UPDATE @t SET pr=2 WHERE pr=1

END

INSERT @report SELECT empid, mgrid

FROM @t

RETURN

END

Применим созданную функцию для определения списка подчиненных сотрудника ‘b’:

SELECT \* FROM fn\_findReports('b')

Оператор возвращает следующие значения:

emp mgr

-----------

b a

e f

f b

g b

Список подчиненных сотрудника ‘a’ создается с помощью оператора

SELECT \* FROM fn\_findReports('a')

emp mgr

---------

a NULL

b a

c a

d a

e f

f b

g b

i c

k d

Другой оператор формирует список подчиненных сотрудника ‘e’:

SELECT \* FROM fn\_findReports('e')

emp mgr

--------

e f

Список подчиненных сотрудника ‘c’ создает следующий оператор:

SELECT \* FROM fn\_findReports('c')

emp mgr

--------

c a

i c

Удаление любой функции осуществляется командой:

DROP FUNCTION {[ владелец.] имя\_функции }

[,...n]

**2.3. Хранимые процедуры**

**Хранимые процедуры** представляют собой группы связанных между собой операторов SQL, применение которых делает работу программиста более легкой и гибкой, поскольку выполнить хранимую процедуру часто оказывается гораздо проще, чем последовательность отдельных операторов SQL. Хранимые процедуры представляют собой набор команд, состоящий из одного или нескольких операторов SQL или функций и сохраняемый в базе данных в откомпилированном виде. Выполнение в базе данных хранимых процедур вместо отдельных операторов SQL дает пользователю следующие преимущества:

* необходимые операторы уже содержатся в базе данных;
* все они прошли этап синтаксического анализа и находятся в исполняемом формате; перед выполнением хранимой процедуры SQL Server генерирует для нее план исполнения, выполняет ее оптимизацию и компиляцию;
* хранимые процедуры поддерживают модульное программирование, так как позволяют разбивать большие задачи на самостоятельные, более мелкие и удобные в управлении части;
* хранимые процедуры могут вызывать другие хранимые процедуры и функции;
* хранимые процедуры могут быть вызваны из прикладных программ других типов;
* как правило, хранимые процедуры выполняются быстрее, чем последовательность отдельных операторов;
* хранимые процедуры проще использовать: они могут состоять из десятков и сотен команд, но для их запуска достаточно указать всего лишь имя нужной хранимой процедуры. Это позволяет уменьшить размер запроса, посылаемого от клиента на сервер, а значит, и нагрузку на сеть.

**Типы хранимых процедур**

* **Системные хранимые процедуры** предназначены для выполнения различных административных действий. Практически все действия по администрированию сервера выполняются с их помощью. Можно сказать, что системные хранимые процедуры являются интерфейсом, обеспечивающим работу с системными таблицами, которая, в конечном счете, сводится к изменению, добавлению, удалению и выборке данных из системных таблиц как пользовательских, так и системных баз данных. Системные хранимые процедуры имеют префикс sp\_, хранятся в системной базе данных и могут быть вызваны в контексте любой другой базы данных.
* **Пользовательские хранимые процедуры** реализуют те или иные действия. Хранимые процедуры – полноценный объект базы данных. Вследствие этого каждая хранимая процедура располагается в конкретной базе данных, где и выполняется.
* **Временные хранимые процедуры** существуют лишь некоторое время, после чего автоматически уничтожаются сервером. Они делятся на локальные и глобальные.

**Локальные временные хранимые процедуры** могут быть вызваны только из того соединения, в котором созданы. При создании такой процедуры ей необходимо дать имя, начинающееся с одного символа #. Как и все временные объекты, хранимые процедуры этого типа автоматически удаляются при отключении пользователя, перезапуске или остановке сервера.

**Глобальные временные хранимые процедуры** доступны для любых соединений сервера, на котором имеется такая же процедура. Для ее определения достаточно дать ей имя, начинающееся с символов ##. Удаляются эти процедуры при перезапуске или остановке сервера, а также при закрытии соединения, в контексте которого они были созданы.

**Создание, изменение и удаление хранимых процедур**

Создание хранимой процедуры предполагает решение следующих задач:

* определение типа создаваемой хранимой процедуры: временная или пользовательская. Кроме этого, можно создать свою собственную системную хранимую процедуру, назначив ей имя с префиксом sp\_ и поместив ее в системную базу данных. Такая процедура будет доступна в контексте любой базы данных локального сервера;
* планирование прав доступа. При создании хранимой процедуры следует учитывать, что она будет иметь те же права доступа к объектам базы данных, что и создавший ее пользователь;
* определение параметров хранимой процедуры. Подобно процедурам, входящим в состав большинства языков программирования, хранимые процедуры могут обладать входными и выходными параметрами ;
* разработка кода хранимой процедуры. Код процедуры может содержать последовательность любых команд SQL, включая вызов других хранимых процедур.

Создание новой и изменение имеющейся хранимой процедуры осуществляется с помощью следующей команды:

<определение\_процедуры>::=

{CREATE | ALTER } [PROCEDURE] имя\_процедуры

[;номер]

[{@имя\_параметра тип\_данных } [VARYING ]

[=default][OUTPUT] ][,...n]

[WITH { RECOMPILE | ENCRYPTION | RECOMPILE,

ENCRYPTION }]

[FOR REPLICATION]

AS

sql\_оператор [...n]

Рассмотрим параметры данной команды.

Используя префиксы sp\_, #, ##, создаваемую процедуру можно определить в качестве системной или временной. Как видно из синтаксиса команды, не допускается указывать имя владельца, которому будет принадлежать создаваемая процедура, а также имя базы данных, где она должна быть размещена. Таким образом, чтобы разместить создаваемую хранимую процедуру в конкретной базе данных, необходимо выполнить команду CREATE PROCEDURE в контексте этой базы данных. При обращении из тела хранимой процедуры к объектам той же базы данных можно использовать укороченные имена, т. е. без указания имени базы данных. Когда же требуется обратиться к объектам, расположенным в других базах данных, указание имени базы данных обязательно.

Номер в имени – это идентификационный номер хранимой процедуры, однозначно определяющий ее в группе процедур. Для удобства управления процедурами логически однотипные хранимые процедуры можно группировать, присваивая им одинаковые имена, но разные идентификационные номера.

Для передачи входных и выходных данных в создаваемой хранимой процедуре могут использоваться параметры, имена которых, как и имена локальных переменных, должны начинаться с символа @. В одной хранимой процедуре можно задать множество параметров, разделенных запятыми. В теле процедуры не должны применяться локальные переменные, чьи имена совпадают с именами параметров этой процедуры.

Для определения типа данных, который будет иметь соответствующий параметр хранимой процедуры, годятся любые типы данных SQL, включая определенные пользователем. Однако тип данных CURSOR может быть использован только как выходной параметр хранимой процедуры, т.е. с указанием ключевого слова OUTPUT.

Наличие ключевого слова OUTPUT означает, что соответствующий параметр предназначен для возвращения данных из хранимой процедуры. Однако это вовсе не означает, что параметр не подходит для передачи значений в хранимую процедуру. Указание ключевого слова OUTPUT предписывает серверу при выходе из хранимой процедуры присвоить текущее значение параметра локальной переменной, которая была указана при вызове процедуры в качестве значения параметра. Отметим, что при указании ключевого слова OUTPUT значение соответствующего параметра при вызове процедуры может быть задано только с помощью локальной переменной. Не разрешается использование любых выражений или констант, допустимое для обычных параметров.

Ключевое слово VARYING применяется совместно с параметром OUTPUT, имеющим тип CURSOR. Оно определяет, что выходным параметром будет результирующее множество.

Ключевое слово DEFAULT представляет собой значение, которое будет принимать соответствующий параметр по умолчанию. Таким образом, при вызове процедуры можно не указывать явно значение соответствующего параметра.

Так как сервер кэширует план исполнения запроса и компилированный код, при последующем вызове процедуры будут использоваться уже готовые значения. Однако в некоторых случаях все же требуется выполнять перекомпиляцию кода процедуры. Указание ключевого слова RECOMPILE предписывает системе создавать план выполнения хранимой процедуры при каждом ее вызове.

Параметр FOR REPLICATION востребован при репликации данных и включении создаваемой хранимой процедуры в качестве статьи в публикацию.

Ключевое слово ENCRYPTION предписывает серверу выполнить шифрование кода хранимой процедуры, что может обеспечить защиту от использования авторских алгоритмов, реализующих работу хранимой процедуры.

Ключевое слово AS размещается в начале собственно тела хранимой процедуры, т. е. набора команд SQL, с помощью которых и будет реализовываться то или иное действие. В теле процедуры могут применяться практически все команды SQL, объявляться транзакции, устанавливаться блокировки и вызываться другие хранимые процедуры. Выход из хранимой процедуры можно осуществить посредством команды RETURN.

Удаление хранимой процедуры осуществляется командой:

DROP PROCEDURE {имя\_процедуры} [,...n]

**Выполнение хранимой процедуры**

Для выполнения хранимой процедуры используется команда:

[[ EXEC [ UTE] имя\_процедуры [;номер]

[[@имя\_параметра=]{значение | @имя\_переменной}

[OUTPUT ]|[DEFAULT ]][,...n]

Если вызов хранимой процедуры не является единственной командой в пакете, то присутствие команды EXECUTE обязательно. Более того, эта команда требуется для вызова процедуры из тела другой процедуры или триггера.

Использование ключевого слова OUTPUT при вызове процедуры разрешается только для параметров, которые были объявлены при создании процедуры с ключевым словом OUTPUT.

Когда же при вызове процедуры для параметра указывается ключевое слово DEFAULT, то будет использовано значение по умолчанию. Естественно, указанное слово DEFAULT разрешается только для тех параметров, для которых определено значение по умолчанию.

Из синтаксиса команды EXECUTE видно, что имена параметров могут быть опущены при вызове процедуры. Однако в этом случае пользователь должен указывать значения для параметров в том же порядке, в каком они перечислялись при создании процедуры. Присвоить параметру значение по умолчанию, просто пропустив его при перечислении нельзя. Если же требуется опустить параметры, для которых определено значение по умолчанию, достаточно явного указания имен параметров при вызове хранимой процедуры. Более того, таким способом можно перечислять параметры и их значения в произвольном порядке.

Отметим, что при вызове процедуры указываются либо имена параметров со значениями, либо только значения без имени параметра. Их комбинирование не допускается.

**Пример**. Процедура без параметров. Разработать процедуру для получения названий и стоимости товаров, приобретенных Ивановым.

CREATE PROC my\_proc1

AS

SELECT Товар.Название,

Товар.Цена\*Сделка.Количество

AS Стоимость, Клиент.Фамилия

FROM Клиент INNER JOIN

(Товар INNER JOIN Сделка

ON Товар.КодТовара=Сделка.КодТовара)

ON Клиент.КодКлиента=Сделка.КодКлиента

WHERE Клиент.Фамилия=’Иванов’

Для обращения к процедуре можно использовать команды:

EXEC my\_proc1 или my\_proc1

Процедура возвращает набор данных.

**Пример**. Процедура без параметров. Создать процедуру для уменьшения цены товара первого сорта на 10%.

CREATE PROC my\_proc2

AS

UPDATE Товар SET Цена=Цена\*0.9 WHERE Сорт=’первый’

Для обращения к процедуре можно использовать команды:

EXEC my\_proc2 или my\_proc2

Процедура не возвращает никаких данных.

**Пример**. Процедура с входным параметром. Создать процедуру для получения названий и стоимости товаров, которые приобрел заданный клиент.

CREATE PROC my\_proc3

@k VARCHAR(20)

AS

SELECT Товар.Название, Товар.Цена\*Сделка.Количество AS Стоимость, Клиент.Фамилия

FROM Клиент INNER JOIN

(Товар INNER JOIN Сделка

ON Товар.КодТовара=Сделка.КодТовара)

ON Клиент.КодКлиента=Сделка.КодКлиента

WHERE Клиент.Фамилия=@k

Для обращения к процедуре можно использовать команды:

EXEC my\_proc3 'Иванов' или

my\_proc3 @k='Иванов'

**Пример**. Процедура с входными параметрами. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

CREATE PROC my\_proc4

@t VARCHAR(20), @p FLOAT

AS

UPDATE Товар SET Цена=Цена\*(1-@p)

WHERE Тип=@t

Для обращения к процедуре можно использовать команды:

EXEC my\_proc4 'Вафли',0.05 или

EXEC my\_proc4 @t='Вафли', @p=0.05

**Пример**. Процедура с входными параметрами и значениями по умолчанию. Создать процедуру для уменьшения цены товара заданного типа в соответствии с указанным %.

CREATE PROC my\_proc5

@t VARCHAR(20)=’Конфеты',

@p FLOAT=0.1

AS

UPDATE Товар SET Цена=Цена\*(1-@p)

WHERE Тип=@t

Для обращения к процедуре можно использовать команды:

EXEC my\_proc5 'Вафли',0.05 или

EXEC my\_proc5 @t='Вафли', @p=0.05 или

EXEC my\_proc5 @p=0.05

В этом случае уменьшается цена конфет (значение типа не указано при вызове процедуры и берется по умолчанию).

EXEC my\_proc5

В последнем случае оба параметра (и тип, и проценты) не указаны при вызове процедуры, их значения берутся по умолчанию.

**Пример**. Процедура с входными и выходными параметрами. Создать процедуру для определения общей стоимости товаров, проданных за конкретный месяц.

CREATE PROC my\_proc6

@m INT,

@s FLOAT OUTPUT

AS

SELECT @s=Sum(Товар.Цена\*Сделка.Количество)

FROM Товар INNER JOIN Сделка ON Товар.КодТовара=Сделка.КодТовара

GROUP BY Month(Сделка.Дата)

HAVING Month(Сделка.Дата)=@m

Для обращения к процедуре можно использовать команды:

DECLARE @st FLOAT

EXEC my\_proc6 1, @st OUTPUT

SELECT @st

Этот блок команд позволяет определить стоимость товаров, проданных в январе (входной параметр месяц указан равным 1).

**2.4. Триггеры**

Триггеры являются одной из разновидностей хранимых процедур. Их исполнение происходит при выполнении для таблицы какого-либо оператора языка манипулирования данными (DML). Триггеры используются для проверки целостности данных, а также для отката транзакций.

**Триггер** – это откомпилированная SQL-процедура, исполнение которой обусловлено наступлением определенных событий внутри реляционной базы данных. Применение триггеров большей частью весьма удобно для пользователей базы данных. И все же их использование часто связано с дополнительными затратами ресурсов на операции ввода/вывода. В том случае, когда тех же результатов (с гораздо меньшими непроизводительными затратами ресурсов) можно добиться с помощью хранимых процедур или прикладных программ, применение триггеров нецелесообразно.

Триггеры – особый инструмент SQL-сервера, используемый для поддержания целостности данных в базе данных. С помощью ограничений целостности, правил и значений по умолчанию не всегда можно добиться нужного уровня функциональности. Часто требуется реализовать сложные алгоритмы проверки данных, гарантирующие их достоверность и реальность. Кроме того, иногда необходимо отслеживать изменения значений таблицы, чтобы нужным образом изменить связанные данные. Триггеры можно рассматривать как своего рода фильтры, вступающие в действие после выполнения всех операций в соответствии с правилами, стандартными значениями и т.д.

Триггер представляет собой специальный тип хранимых процедур, запускаемых сервером автоматически при попытке изменения данных в таблицах, с которыми триггеры связаны. Каждый триггер привязывается к конкретной таблице. Все производимые им модификации данных рассматриваются как одна транзакция. В случае обнаружения ошибки или нарушения целостности данных происходит откат этой транзакции. Тем самым внесение изменений запрещается. Отменяются также все изменения, уже сделанные триггером.

Создает триггер только владелец базы данных. Это ограничение позволяет избежать случайного изменения структуры таблиц, способов связи с ними других объектов и т.п.

Триггер представляет собой весьма полезное и в то же время опасное средство. Так, при неправильной логике его работы можно легко уничтожить целую базу данных, поэтому триггеры необходимо очень тщательно отлаживать.

В отличие от обычной подпрограммы, триггер выполняется неявно в каждом случае возникновения триггерного события, к тому же он не имеет аргументов. Приведение его в действие иногда называют запуском триггера. С помощью триггеров достигаются следующие цели:

* проверка корректности введенных данных и выполнение сложных ограничений целостности данных, которые трудно, если вообще возможно, поддерживать с помощью ограничений целостности, установленных для таблицы;
* выдача предупреждений, напоминающих о необходимости выполнения некоторых действий при обновлении таблицы, реализованном определенным образом;
* накопление аудиторской информации посредством фиксации сведений о внесенных изменениях и тех лицах, которые их выполнили;
* поддержка репликации.

В реализации СУБД MS SQL Server используется следующий оператор создания или изменения триггера:

<Определение\_триггера>::=

{CREATE | ALTER} TRIGGER имя\_триггера

ON {имя\_таблицы | имя\_представления }

[WITH ENCRYPTION ]

{

{ { FOR | AFTER | INSTEAD OF }

{ [ DELETE] [,] [ INSERT] [,] [ UPDATE] }

[ WITH APPEND ]

[ NOT FOR REPLICATION ]

AS

sql\_оператор[...n]

} |

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

[UPDATE] }

[ WITH APPEND]

[ NOT FOR REPLICATION]

AS

{ IF UPDATE(имя\_столбца)

[ {AND | OR} UPDATE(имя\_столбца)] [...n]

|

IF (COLUMNS\_UPDATES(){оператор\_бит\_обработки}

бит\_маска\_изменения)

{оператор\_бит\_сравнения }бит\_маска [...n]}

sql\_оператор [...n]

}

}

Триггер может быть создан только в текущей базе данных, но допускается обращение внутри триггера к другим базам данных, в том числе и расположенным на удаленном сервере.

Рассмотрим назначение аргументов из команды CREATE | ALTER TRIGGER.

Имя триггера должно быть уникальным в пределах базы данных. Дополнительно можно указать имя владельца.

При указании аргумента WITH ENCRYPTION сервер выполняет шифрование кода триггера, чтобы никто, включая администратора, не мог получить к нему доступ и прочитать его. Шифрование часто используется для скрытия авторских алгоритмов обработки данных, являющихся интеллектуальной собственностью программиста или коммерческой тайной.

**Типы триггеров**

В SQL Server существует два параметра, определяющих поведение триггеров:

* **AFTER**. Триггер выполняется после успешного выполнения вызвавших его команд. Если же команды по какой-либо причине не могут быть успешно завершены, триггер не выполняется. Следует отметить, что изменения данных в результате выполнения запроса пользователя и выполнение триггера осуществляется в теле одной транзакции: если произойдет откат триггера, то будут отклонены и пользовательские изменения. Можно определить несколько AFTER -триггеров для каждой операции ( INSERT, UPDATE, DELETE ). Если для таблицы предусмотрено выполнение нескольких AFTER -триггеров, то с помощью системной хранимой процедуры sp\_settriggerorder можно указать, какой из них будет выполняться первым, а какой последним. По умолчанию в SQL Server все триггеры являются AFTER -триггерами.
* **INSTEAD OF**. Триггер вызывается вместо выполнения команд. В отличие от AFTER -триггера INSTEAD OF -триггер может быть определен как для таблицы, так и для представления. Для каждой операции INSERT, UPDATE, DELETE можно определить только один INSTEAD OF -триггер.

Триггеры различают по типу команд, на которые они реагируют.

Существует три типа триггеров:

* **INSERT TRIGGER** – запускаются при попытке вставки данных с помощью команды INSERT.
* **UPDATE TRIGGER** – запускаются при попытке изменения данных с помощью команды UPDATE.
* **DELETE TRIGGER** – запускаются при попытке удаления данных с помощью команды DELETE.

Конструкции [ DELETE] [,] [ INSERT] [,] [ UPDATE] и FOR | AFTER | INSTEAD OF } { [INSERT] [,] [UPDATE] определяют, на какую команду будет реагировать триггер. При его создании должна быть указана хотя бы одна команда. Допускается создание триггера, реагирующего на две или на все три команды.

Аргумент WITH APPEND позволяет создавать несколько триггеров каждого типа.

При создании триггера с аргументом NOT FOR REPLICATION запрещается его запуск во время выполнения модификации таблиц механизмами репликации.

Конструкция AS sql\_оператор[...n] определяет набор SQL- операторов и команд, которые будут выполнены при запуске триггера.

Отметим, что внутри триггера не допускается выполнение ряда операций, таких, например, как:

* создание, изменение и удаление базы данных;
* восстановление резервной копии базы данных или журнала транзакций.

Выполнение этих команд не разрешено, так как они не могут быть отменены в случае отката транзакции, в которой выполняется триггер. Этот запрет вряд ли может каким-то образом сказаться на функциональности создаваемых триггеров. Трудно найти такую ситуацию, когда, например, после изменения строки таблицы потребуется выполнить восстановление резервной копии журнала транзакций.

**Программирование триггера**

При выполнении команд добавления, изменения и удаления записей сервер создает две специальные таблицы: inserted и deleted. В них содержатся списки строк, которые будут вставлены или удалены по завершении транзакции. Структура таблиц inserted и deleted идентична структуре таблиц, для которой определяется триггер. Для каждого триггера создается свой комплект таблиц inserted и deleted, поэтому никакой другой триггер не сможет получить к ним доступ. В зависимости от типа операции, вызвавшей выполнение триггера, содержимое таблиц inserted и deleted может быть разным:

* команда INSERT – в таблице inserted содержатся все строки, которые пользователь пытается вставить в таблицу; в таблице deleted не будет ни одной строки; после завершения триггера все строки из таблицы inserted переместятся в исходную таблицу;
* команда DELETE – в таблице deleted будут содержаться все строки, которые пользователь попытается удалить; триггер может проверить каждую строку и определить, разрешено ли ее удаление; в таблице inserted не окажется ни одной строки;
* команда UPDATE – при ее выполнении в таблице deleted находятся старые значения строк, которые будут удалены при успешном завершении триггера. Новые значения строк содержатся в таблице inserted. Эти строки добавятся в исходную таблицу после успешного выполнения триггера.

Для получения информации о количестве строк, которое будет изменено при успешном завершении триггера, можно использовать функцию @@ROWCOUNT; она возвращает количество строк, обработанных последней командой. Следует подчеркнуть, что триггер запускается не при попытке изменить конкретную строку, а в момент выполнения команды изменения. Одна такая команда воздействует на множество строк, поэтому триггер должен обрабатывать все эти строки.

Если триггер обнаружил, что из 100 вставляемых, изменяемых или удаляемых строк только одна не удовлетворяет тем или иным условиям, то никакая строка не будет вставлена, изменена или удалена. Такое поведение обусловлено требованиями транзакции – должны быть выполнены либо все модификации, либо ни одной.

Триггер выполняется как неявно определенная транзакция, поэтому внутри триггера допускается применение команд управления транзакциями. В частности, при обнаружении нарушения ограничений целостности для прерывания выполнения триггера и отмены всех изменений, которые пытался выполнить пользователь, необходимо использовать команду ROLLBACK TRANSACTION.

Для получения списка столбцов, измененных при выполнении команд INSERT или UPDATE, вызвавших выполнение триггера, можно использовать функцию COLUMNS\_UPDATED(). Она возвращает двоичное число, каждый бит которого, начиная с младшего, соответствует одному столбцу таблицы (в порядке следования столбцов при создании таблицы). Если бит установлен в значение "1", то соответствующий столбец был изменен. Кроме того, факт изменения столбца определяет и функция UPDATE (имя\_столбца).

Для удаления триггера используется команда

DROP TRIGGER {имя\_триггера} [,...n]

Приведем примеры использования триггеров.

**Пример 1**. Использование триггера для реализации ограничений на значение. В добавляемой в таблицу Сделка записи количество проданного товара должно быть не больше, чем его остаток из таблицы Склад.

Команда вставки записи в таблицу Сделка может быть, например, такой:

INSERT INTO Сделка VALUES (3,1,-299,'01/08/2002')

Создаваемый триггер должен отреагировать на ее выполнение следующим образом: необходимо отменить команду, если в таблице Склад величина остатка товара оказалась меньше продаваемого количества товара с введенным кодом (в примере код товара=3 ). Во вставляемой записи количество товара указывается со знаком "+", если товар поставляется, и со знаком "-", если он продается. Представленный триггер настроен на обработку только одной добавляемой записи.

CREATE TRIGGER Триггер\_ins

ON Сделка FOR INSERT

AS

IF @@ROWCOUNT=1

BEGIN

IF NOT EXISTS(SELECT \*

FROM inserted

WHERE -inserted.количество<=ALL(SELECT

Склад.Остаток

FROM Склад,Сделка

WHERE Склад.КодТовара=

Сделка.КодТовара))

BEGIN

ROLLBACK TRAN

PRINT

'Отмена поставки: товара на складе нет'

END

END

**Пример 2**. Использование триггера для сбора статистических данных.

Создать триггер для обработки операции вставки записи в таблицу Сделка, например, такой команды:

INSERT INTO Сделка VALUES (3,1,200,'01/08/2002')

поставляется товар с кодом 3 от клиента с кодом 1 в количестве 200 единиц.

При продаже или получении товара необходимо соответствующим образом изменить количество его складского запаса. Если товара на складе еще нет, необходимо добавить соответствующую запись в таблицу Склад. Триггер обрабатывает только одну добавляемую строку.

ALTER TRIGGER Триггер\_ins

ON Сделка FOR INSERT

AS

DECLARE @x INT, @y INT

IF @@ROWCOUNT=1

--в таблицу Сделка добавляется запись

--о поставке товара

BEGIN

--количество проданного товара должно быть не

--меньше, чем его остаток из таблицы Склад

IF NOT EXISTS(SELECT \* FROM inserted

WHERE -inserted.количество< =ALL(SELECT Склад.Остаток

FROM Склад,Сделка

WHERE Склад.КодТовара=

Сделка.КодТовара))

BEGIN

ROLLBACK TRAN

PRINT 'откат товара нет '

END

--если записи о поставленном товаре еще нет,

--добавляется соответствующая запись

--в таблицу Склад

IF NOT EXISTS ( SELECT \* FROM Склад С, inserted i

WHERE С.КодТовара=i.КодТовара )

INSERT INTO Склад (КодТовара,Остаток)

ELSE

--если запись о товаре уже была в таблице

--Склад, то определяется код и количество

--товара из добавленной в таблицу Сделка записи

BEGIN

SELECT @y=i.КодТовара, @x=i.Количество

FROM Сделка С, inserted i

WHERE С.КодТовара=i.КодТовара

--и производится изменения количества товара в

--таблице Склад

UPDATE Склад

SET Остаток=остаток+@x

WHERE КодТовара=@y

END

END

**Пример 3**. Создать триггер для обработки операции удаления записи из таблицы Сделка, например, такой команды:

DELETE FROM Сделка WHERE КодСделки=4

Для товара, код которого указан при удалении записи, необходимо откорректировать его остаток на складе. Триггер обрабатывает только одну удаляемую запись.

CREATE TRIGGER Триггер\_del

ON Сделка FOR DELETE

AS

IF @@ROWCOUNT=1 -- удалена одна запись

BEGIN

DECLARE @y INT,@x INT

--определяется код и количество товара из

--удаленной из таблицы Склад записи

SELECT @y=КодТовара, @x=Количество

FROM deleted

--в таблице Склад корректируется количество

--товара

UPDATE Склад

SET Остаток=Остаток-@x

WHERE КодТовара=@y

END

**Пример 4**. Создать триггер для обработки операции изменения записи в таблице Сделка, например, такой командой:

UPDATE Сделка SET количество=количество-10

WHERE КодТовара=3

во всех сделках с товаром, имеющим код, равный 3, уменьшить количество товара на 10 единиц.

Указанная команда может привести к изменению сразу нескольких записей в таблице Сделка. Поэтому покажем, как создать триггер, обрабатывающий не одну запись. Для каждой измененной записи необходимо для старого (до изменения) кода товара уменьшить остаток товара на складе на величину старого (до изменения) количества товара и для нового (после изменения) кода товара увеличить его остаток на складе на величину нового (после изменения) значения. Чтобы обработать все измененные записи, введем курсоры, в которых сохраним все старые (из таблицы deleted ) и все новые значения (из таблицы inserted ).

CREATE TRIGGER Триггер\_upd

ON Сделка FOR UPDATE

AS

DECLARE @x INT, @x\_old INT, @y INT, @y\_old INT

-- курсор с новыми значениями

DECLARE CUR1 CURSOR FOR

SELECT КодТовара,Количество

FROM inserted

-- курсор со старыми значениями

DECLARE CUR2 CURSOR FOR

SELECT КодТовара, Количество

FROM deleted

OPEN CUR1

OPEN CUR2

-- перемещаемся параллельно по обоим курсорам

FETCH NEXT FROM CUR1 INTO @x, @y

FETCH NEXT FROM CUR2 INTO @x\_old, @y\_old

WHILE @@FETCH\_STATUS=0

BEGIN

--для старого кода товара уменьшается его

--количество на складе

UPDATE Склад

SET Остаток=Остаток-@y\_old

WHERE КодТовара=@x\_old

--для нового кода товара, если такого товара

--еще нет на складе, вводится новая запись

IF NOT EXISTS (SELECT \* FROM Склад

WHERE КодТовара=@x)

INSERT INTO Склад(КодТовара,Остаток)

VALUES (@x,@y)

ELSE

--иначе для нового кода товара увеличивается

--его количество на складе

UPDATE Склад

SET Остаток=Остаток+@y

WHERE КодТовара=@x

FETCH NEXT FROM CUR1 INTO @x, @y

FETCH NEXT FROM CUR2 INTO @x\_old, @y\_old

END

CLOSE CUR1

CLOSE CUR2

DEALLOCATE CUR1

DEALLOCATE CUR2

В рассмотренном триггере отсутствует сравнение количества товара при изменении записи о сделке с его остатком на складе.

**Пример 5**. Исправим этот недостаток. Для генерирования сообщения об ошибке используем в теле триггера команду MS SQL Server RAISERROR, аргументами которой являются текст сообщения, уровень серьезности и статус ошибки.

ALTER TRIGGER Триггер\_upd

ON Сделка FOR UPDATE

AS

DECLARE @x INT, @x\_old INT, @y INT,

@y\_old INT,@o INT

DECLARE CUR1 CURSOR FOR

SELECT КодТовара, Количество

FROM inserted

DECLARE CUR2 CURSOR FOR

SELECT КодТовара, Количество

FROM deleted

OPEN CUR1

OPEN CUR2

FETCH NEXT FROM CUR1 INTO @x, @y

FETCH NEXT FROM CUR2 INTO @x\_old, @y\_old

WHILE @@FETCH\_STATUS=0

BEGIN

SELECT @o=остаток

FROM Склад

WHERE кодтовара=@x

IF @o<-@y

BEGIN

RAISERROR('откат',16,10)

CLOSE CUR1

CLOSE CUR2

DEALLOCATE CUR1

DEALLOCATE CUR2

ROLLBACK TRAN

RETURN

END

UPDATE Склад

SET Остаток=Остаток-@y\_old

WHERE КодТовара=@x\_old

IF NOT EXISTS (SELECT \* FROM Склад

WHERE КодТовара=@x)

INSERT INTO Склад(КодТовара,Остаток)

VALUES (@x,@y)

ELSE

UPDATE Склад

SET Остаток=Остаток+@y

WHERE КодТовара=@x

FETCH NEXT FROM CUR1 INTO @x, @y

FETCH NEXT FROM CUR2 INTO @x\_old, @y\_old

END

CLOSE CUR1

CLOSE CUR2

DEALLOCATE CUR1

DEALLOCATE CUR2

**Пример 6**. В примере [5](https://www.intuit.ru/studies/courses/5/5/lecture/148?page=3#example.14.5) происходит отмена всех изменений при невозможности реализовать хотя бы одно из них. Создадим триггер, позволяющий отменять изменение только некоторых записей и выполнять изменение остальных.

В этом случае триггер выполняется не после изменения записей, а вместо команды изменения.

ALTER TRIGGER Триггер\_upd

ON Сделка INSTEAD OF UPDATE

AS

DECLARE @k INT, @k\_old INT

DECLARE @x INT, @x\_old INT, @y INT

DECLARE @y\_old INT ,@o INT

DECLARE CUR1 CURSOR FOR

SELECT КодСделки, КодТовара, Количество

FROM inserted

DECLARE CUR2 CURSOR FOR

SELECT КодСделки, КодТовара, Количество

FROM deleted

OPEN CUR1

OPEN CUR2

FETCH NEXT FROM CUR1 INTO @k,@x, @y

FETCH NEXT FROM CUR2 INTO @k\_old, @x\_old,

@y\_old

WHILE @@FETCH\_STATUS=0

BEGIN

SELECT @o=остаток

FROM Склад

WHERE КодТовара=@x

IF @o>=-@y

BEGIN

RAISERROR('изменение',16,10)

UPDATE Сделка SET количество=@y,

КодТовара=@x

WHERE КодСделки=@k

UPDATE Склад

SET Остаток=Остаток-@y\_old

WHERE КодТовара=@x\_old

IF NOT EXISTS (SELECT \* FROM Склад

WHERE КодТовара=@x)

INSERT INTO Склад(КодТовара, Остаток)

VALUES (@x,@y)

ELSE

UPDATE Склад

SET Остаток=Остаток+@y

WHERE КодТовара=@x

END

ELSE

RAISERROR('запись не изменена',16,10)

FETCH NEXT FROM CUR1 INTO @k,@x, @y

FETCH NEXT FROM CUR2 INTO @k\_old, @x\_old,

@y\_old

END

CLOSE CUR1

CLOSE CUR2

DEALLOCATE CUR1

DEALLOCATE CUR2

Пример 6. Триггер, позволяющий отменять изменение только некоторых записей и выполнять изменение остальных.

**2.5. Курсоры – принципы работы**

**Понятие курсора**

Запрос к реляционной базе данных обычно возвращает несколько рядов (записей) данных, но приложение за один раз обрабатывает лишь одну запись. Даже если оно имеет дело одновременно с несколькими рядами (например, выводит данные в форме электронных таблиц), их количество по-прежнему ограничено. Кроме того, при модификации, удалении или добавлении данных рабочей единицей является ряд. В этой ситуации на первый план выступает концепция курсора, и в таком контексте курсор – указатель на ряд.

**Курсор в SQL** – это область в памяти базы данных, которая предназначена для хранения последнего оператора SQL. Если текущий оператор – запрос к базе данных, в памяти сохраняется и строка данных запроса, называемая текущим значением, или текущей строкой курсора. Указанная область в памяти поименована и доступна для прикладных программ.

Обычно курсоры используются для выбора из базы данных некоторого подмножества хранимой в ней информации. В каждый момент времени прикладной программой может быть проверена одна строка курсора. Курсоры часто применяются в операторах SQL, встроенных в написанные на языках процедурного типа прикладные программы. Некоторые из них неявно создаются сервером базы данных, в то время как другие определяются программистами.

В соответствии со стандартом SQL при работе с курсорами можно выделить следующие основные действия:

* создание или объявление курсора ;
* открытие курсора, т.е. наполнение его данными, которые сохраняются в многоуровневой памяти;
* выборка из курсора и изменение с его помощью строк данных;
* закрытие курсора, после чего он становится недоступным для пользовательских программ;
* освобождение курсора, т.е. удаление курсора как объекта, поскольку его закрытие необязательно освобождает ассоциированную с ним память.

В разных реализациях определение курсора может иметь некоторые отличия. Так, например, иногда разработчик должен явным образом освободить выделяемую для курсора память. После освобождения курсора ассоциированная с ним память также освобождается. При этом становится возможным повторное использование его имени. В других реализациях при закрытии курсора освобождение памяти происходит неявным образом. Сразу после восстановления она становится доступной для других операций: открытие другого курсора и т. д.

В некоторых случаях применение курсора неизбежно. Однако по возможности этого следует избегать и работать со стандартными командами обработки данных: SELECT, UPDATE, INSERT, DELETE. Помимо того, что курсоры не позволяют проводить операции изменения над всем объемом данных, скорость выполнения операций обработки данных посредством курсора заметно ниже, чем у стандартных средств SQL.

**Реализация курсоров в среде MS SQL Server**

SQL Server поддерживает три вида курсоров:

1. курсоры SQL применяются в основном внутри триггеров, хранимых процедур и сценариев;
2. курсоры сервера действуют на сервере и реализуют программный интерфейс приложений для ODBC, OLE DB, DB\_Library;
3. курсоры клиента реализуются на самом клиенте. Они выбирают весь результирующий набор строк из сервера и сохраняют его локально, что позволяет ускорить операции обработки данных за счет снижения потерь времени на выполнение сетевых операций.

Различные типы многопользовательских приложений требуют и различных типов организации параллельного доступа к данным. Некоторым приложениям необходим немедленный доступ к информации об изменениях в базе данных. Это характерно для систем резервирования билетов. В других случаях, например, в системах статистической отчетности, важна стабильность данных, ведь если они постоянно модифицируются, программы не смогут эффективно отображать информацию. Различным приложениям нужны разные реализации курсоров.

В среде SQL Server типы курсоров различаются по предоставляемым возможностям. Тип курсора определяется на стадии его создания и не может быть изменен. Некоторые типы курсоров могут обнаруживать изменения, сделанные другими пользователями в строках, включенных в результирующий набор. Однако SQL Server отслеживает изменения таких строк только на стадии обращения к строке и не позволяет отслеживать изменения, когда строка уже считана.

Курсоры делятся на две категории: последовательные и прокручиваемые. Последовательные позволяют выбирать данные только в одном направлении – от начала к концу. Прокручиваемые же курсоры предоставляют большую свободу действий – допускается перемещение в обоих направлениях и переход к произвольной строке результирующего набора курсора. Если программа способна модифицировать данные, на которые указывает курсор, он называется прокручиваемым и модифицируемым. Говоря о курсорах, не следует забывать об изолированности транзакций. Когда один пользователь модифицирует запись, другой читает ее при помощи собственного курсора, более того, он может модифицировать ту же запись, что делает необходимым соблюдение целостности данных.

SQL Server поддерживает курсоры статические, динамические, последовательные и управляемые набором ключей.

В схеме со статическим курсором информация читается из базы данных один раз и хранится в виде моментального снимка (по состоянию на некоторый момент времени), поэтому изменения, внесенные в базу данных другим пользователем, не видны. На время открытия курсора сервер устанавливает блокировку на все строки, включенные в его полный результирующий набор. Статический курсор не изменяется после создания и всегда отображает тот набор данных, который существовал на момент его открытия.

Если другие пользователи изменят в исходной таблице включенные в курсор данные, это никак не повлияет на статический курсор.

В статический курсор внести изменения невозможно, поэтому он всегда открывается в режиме "только для чтения".

Динамический курсор поддерживает данные в "живом" состоянии, но это требует затрат сетевых и программных ресурсов. При использовании динамических курсоров не создается полная копия исходных данных, а выполняется динамическая выборка из исходных таблиц только при обращении пользователя к тем или иным данным. На время выборки сервер блокирует строки, а все изменения, вносимые пользователем в полный результирующий набор курсора, будут видны в курсоре. Однако если другой пользователь внес изменения уже после выборки данных курсором, то они не отразятся в курсоре.

Курсор, управляемый набором ключей, находится посередине между этими крайностями. Записи идентифицируются на момент выборки, и тем самым отслеживаются изменения. Такой тип курсора полезен при реализации прокрутки назад – тогда добавления и удаления рядов не видны, пока информация не обновится, а драйвер выбирает новую версию записи, если в нее были внесены изменения.

Последовательные курсоры не разрешают выполнять выборку данных в обратном направлении. Пользователь может выбирать строки только от начала к концу курсора. Последовательный курсор не хранит набор всех строк. Они считываются из базы данных, как только выбираются в курсоре, что позволяет динамически отражать все изменения, вносимые пользователями в базу данных с помощью команд INSERT, UPDATE, DELETE. В курсоре видно самое последнее состояние данных.

Статические курсоры обеспечивают стабильный взгляд на данные. Они хороши для систем "складирования" информации: приложений для систем отчетности или для статистических и аналитических целей. Кроме того, статический курсор лучше других справляется с выборкой большого количества данных. Напротив, в системах электронных покупок или резервирования билетов необходимо динамическое восприятие обновляемой информации по мере внесения изменений. В таких случаях используется динамический курсор. В этих приложениях объем передаваемых данных, как правило, невелик, а доступ к ним осуществляется на уровне рядов (отдельных записей). Групповой доступ встречается очень редко.

**Управление курсором в среде MS SQL Server**

Управление курсором реализуется путем выполнения следующих команд:

* DECLARE – создание или объявление курсора ;
* OPEN – открытие курсора, т.е. наполнение его данными;
* FETCH – выборка из курсора и изменение строк данных с помощью курсора;
* CLOSE – закрытие курсора ;
* DEALLOCATE – освобождение курсора, т.е. удаление курсора как объекта.

**Объявление курсора**

В стандарте SQL для создания курсора предусмотрена следующая команда:

<создание\_курсора>::=

  DECLARE имя\_курсора

    [INSENSITIVE][SCROLL] CURSOR

  FOR SELECT\_оператор

  [FOR { READ\_ONLY | UPDATE

    [OF имя\_столбца[,...n]]}]

При использовании ключевого слова INSENSITIVE будет создан статический курсор. Изменения данных не разрешаются, кроме того, не отображаются изменения, сделанные другими пользователями. Если ключевое слово INSENSITIVE отсутствует, создается динамический курсор.

При указании ключевого слова SCROLL созданный курсор можно прокручивать в любом направлении, что позволяет применять любые команды выборки. Если этот аргумент опускается, то курсор окажется последовательным, т.е. его просмотр будет возможен только в одном направлении – от начала к концу.

SELECT-оператор задает тело запроса SELECT, с помощью которого определяется результирующий набор строк курсора.

При указании аргумента FOR READ\_ONLY создается курсор "только для чтения", и никакие модификации данных не разрешаются. Он отличается от статического, хотя последний также не позволяет менять данные. В качестве курсора "только для чтения" может быть объявлен динамический курсор, что позволит отображать изменения, сделанные другим пользователем.

Создание курсора с аргументом FOR UPDATE позволяет выполнять в курсоре изменение данных либо в указанных столбцах, либо, при отсутствии аргумента OF имя\_столбца, во всех столбцах.

В среде MS SQL Server принят следующий синтаксис команды создания курсора:

<создание\_курсора>::=

  DECLARE имя\_курсора CURSOR [LOCAL | GLOBAL]

  [FORWARD\_ONLY | SCROLL]

  [STATIC | KEYSET | DYNAMIC | FAST\_FORWARD]

  [READ\_ONLY | SCROLL\_LOCKS | OPTIMISTIC]

  [TYPE\_WARNING]

  FOR SELECT\_оператор

  [FOR UPDATE [OF имя\_столбца[,...n]]]

При использовании ключевого слова LOCAL будет создан локальный курсор, который виден только в пределах создавшего его пакета, триггера, хранимой процедуры или пользовательской функции. По завершении работы пакета, триггера, процедуры или функции курсор неявно уничтожается. Чтобы передать содержимое курсора за пределы создавшей его конструкции, необходимо присвоить его параметру аргумент OUTPUT.

Если указано ключевое слово GLOBAL, создается глобальный курсор; он существует до закрытия текущего соединения.

При указании FORWARD\_ONLY создается последовательный курсор; выборку данных можно осуществлять только в направлении от первой строки к последней.

При указании SCROLL создается прокручиваемый курсор; обращаться к данным можно в любом порядке и в любом направлении.

При указании STATIC создается статический курсор.

При указании KEYSET создается ключевой курсор.

При указании DYNAMIC создается динамический курсор.

Если для курсора READ\_ONLY указать аргумент FAST\_FORWARD, то созданный курсор будет оптимизирован для быстрого доступа к данным. Этот аргумент не может быть использован совместно с аргументами FORWARD\_ONLY и OPTIMISTIC.

В курсоре, созданном с указанием аргумента OPTIMISTIC, запрещается изменение и удаление строк, которые были изменены после открытия курсора.

При указании аргумента TYPE\_WARNING сервер будет информировать пользователя о неявном изменении типа курсора, если он несовместим с запросом SELECT.

**Открытие курсора**

Для открытия курсора и наполнения его данными из указанного при создании курсора запроса SELECT используется следующая команда:

OPEN {{[GLOBAL]имя\_курсора }

     |@имя\_переменной\_курсора}

После открытия курсора происходит выполнение связанного с ним оператора SELECT, выходные данные которого сохраняются в многоуровневой памяти.

**Выборка данных из курсора**

Сразу после открытия курсора можно выбрать его содержимое (результат выполнения соответствующего запроса) посредством следующей команды:

FETCH [[NEXT | PRIOR | FIRST | LAST

  | ABSOLUTE {номер\_строки

  | @переменная\_номера\_строки}

  | RELATIVE {номер\_строки |

    @переменная\_номера\_строки}]

  FROM ]{{[GLOBAL ]имя\_курсора }|

    @имя\_переменной\_курсора }

  [INTO @имя\_переменной [,...n]]

При указании FIRST будет возвращена самая первая строка полного результирующего набора курсора, которая становится текущей строкой.

При указании LAST возвращается самая последняя строка курсора. Она же становится текущей строкой.

При указании NEXT возвращается строка, находящаяся в полном результирующем наборе сразу же после текущей. Теперь она становится текущей. По умолчанию команда FETCH использует именно этот способ выборки строк.

Ключевое слово PRIOR возвращает строку, находящуюся перед текущей. Она и становится текущей.

Аргумент ABSOLUTE {номер\_строки | @переменная\_номера\_строки} возвращает строку по ее абсолютному порядковому номеру в полном результирующем наборе курсора. Номер строки можно задать с помощью константы или как имя переменной, в которой хранится номер строки. Переменная должна иметь целочисленный тип данных. Указываются как положительные, так и отрицательные значения. При указании положительного значения строка отсчитывается от начала набора, отрицательного – от конца. Выбранная строка становится текущей. Если указано нулевое значение, строка не возвращается.

Аргумент RELATIVE {кол\_строки | @переменная\_кол\_строки} возвращает строку, находящуюся через указанное количество строк после текущей. Если указать отрицательное значение числа строк, то будет возвращена строка, находящаяся за указанное количество строк перед текущей. При указании нулевого значения возвратится текущая строка. Возвращенная строка становится текущей.

Чтобы открыть глобальный курсор, перед его именем требуется указать ключевое слово GLOBAL. Имя курсора также может быть указано с помощью переменной.

В конструкции INTO @имя\_переменной [,...n] задается список переменных, в которых будут сохранены соответствующие значения столбцов возвращаемой строки. Порядок указания переменных должен соответствовать порядку столбцов в курсоре, а тип данных переменной – типу данных в столбце курсора. Если конструкция INTO не указана, то поведение команды FETCH будет напоминать поведение команды SELECT – данные выводятся на экран.

**Изменение и удаление данных**

Для выполнения изменений с помощью курсора необходимо выполнить команду UPDATE в следующем формате:

UPDATE имя\_таблицы SET {имя\_столбца={

  DEFAULT | NULL | выражение}}[,...n]

  WHERE CURRENT OF {{[GLOBAL] имя\_курсора}

  |@имя\_переменной\_курсора}

За одну операцию могут быть изменены несколько столбцов текущей строки курсора, но все они должны принадлежать одной таблице.

Для удаления данных посредством курсора используется команда DELETE в следующем формате:

DELETE имя\_таблицы

  WHERE CURRENT OF {{[GLOBAL] имя\_курсора}

  |@имя\_переменной\_курсора}

В результате будет удалена строка, установленная текущей в курсоре.

**Закрытие курсора**

CLOSE {имя\_курсора | @имя\_переменной\_курсора}

После закрытия курсор становится недоступным для пользователей программы. При закрытии снимаются все блокировки, установленные в процессе его работы. Закрытие может применяться только к открытым курсорам. Закрытый, но не освобожденный курсор может быть повторно открыт. Не допускается закрывать неоткрытый курсор.

**Освобождение курсора**

Закрытие курсора необязательно освобождает ассоциированную с ним память. В некоторых реализациях нужно явным образом освободить ее с помощью оператора DEALLOCATE. После освобождения курсора освобождается и память, при этом становится возможным повторное использование имени курсора.

DEALLOCATE { имя\_курсора |

  @имя\_переменной\_курсора }

Для контроля достижения конца курсора рекомендуется применять функцию: @@FETCH\_STATUS

Функция @@FETCH\_STATUS возвращает:

0, если выборка завершилась успешно;

-1, если выборка завершилась неудачно вследствие попытки выборки строки, находящейся за пределами курсора ;

-2, если выборка завершилась неудачно вследствие попытки обращения к удаленной или измененной строке.

**Пример 1.** Объявление курсора.

DECLARE abc CURSOR SCROLL FOR

SELECT \* FROM Клиент

**Пример 2.** Использование переменной для объявления курсора.

DECLARE @MyCursor CURSOR

SET @MyCursor=CURSOR LOCAL SCROLL FOR

SELECT \* FROM Клиент

**Пример 3.** Объявление и открытие курсора.

DECLARE abc CURSOR GLOBAL SCROLL FOR

SELECT \* FROM Клиент

OPEN abc

**Пример 4.** Использование переменной для переприсваивания курсора.

DECLARE @MyCursor CURSOR

SET @MyCursor=abc

**Пример 5**. Разработать курсор для вывода списка фирм и клиентов из Москвы.

DECLARE  @firm    VARCHAR(50),

         @fam     VARCHAR(50),

         @message VARCHAR(80)

PRINT ' Список клиентов'

DECLARE klient\_cursor CURSOR LOCAL FOR

    SELECT Фирма, Фамилия

    FROM Клиент

    WHERE Город='Москва'

    ORDER BY Фирма, Фамилия

OPEN klient\_cursor

FETCH NEXT FROM klient\_cursor INTO @firm, @fam

WHILE @@FETCH\_STATUS=0

BEGIN

    SELECT @message='Клиент '+@fam+

                    ' Фирма '+ @firm

    PRINT @message

-- переход к следующему клиенту--

    FETCH NEXT FROM klient\_cursor

      INTO @firm, @fam

END

CLOSE klient\_cursor

DEALLOCATE klient\_cursor

**Пример 6**. Разработать курсор для вывода списка приобретенных клиентами из Москвы товаров и их общей стоимости. В один курсор заносятся все московские клиенты, затем для каждой строки курсора, т.е. для каждого клиента, определяется и распечатывается другой курсор – его покупки. Подсчитывается общая стоимость покупок клиента.

DECLARE @id\_kl    INT,

        @firm VARCHAR(50),

        @fam VARCHAR(50),

        @message VARCHAR(80),

        @nam VARCHAR(50),

        @d DATETIME,

        @p INT,

        @s INT

SET @s=0

PRINT '  Список покупок'

DECLARE klient\_cursor CURSOR LOCAL FOR

    SELECT КодКлиента, Фирма, Фамилия

    FROM Клиент

    WHERE Город='Москва'

    ORDER BY Фирма, Фамилия

OPEN klient\_cursor

FETCH NEXT FROM klient\_cursor

INTO @id\_kl, @firm, @fam

WHILE @@FETCH\_STATUS=0

BEGIN

    SELECT @message='Клиент '+@fam+

        ' Фирма '+ @firm

    PRINT @message

    SELECT @message='Наименование товара Дата

        покупки Стоимость'

    PRINT @message

    DECLARE tovar\_cursor CURSOR FOR

        SELECT Товар.Название, Сделка.Дата,

            Товар.Цена\*Сделка.Количество AS

    Стоимость

        FROM Товар INNER JOIN Сделка ON Товар.

    КодТовара=Сделка.КодТовара

        WHERE Сделка.КодКлиента=@id\_kl

    OPEN tovar\_cursor

    FETCH NEXT FROM tovar\_cursor

      INTO @nam, @d, @p

    IF @@FETCH\_STATUS<>0

        PRINT ' Нет покупок'

    WHILE @@FETCH\_STATUS=0

    BEGIN

        SELECT @message='   '+@nam+'   '+

           CAST(@d AS CHAR(12))+'  '+

           CAST(@p AS CHAR(6))

        PRINT @message

        SET @s=@s+@p

        FETCH NEXT FROM tovar\_cursor

        INTO @nam, @d, @p

    END

    CLOSE tovar\_cursor

    DEALLOCATE tovar\_cursor

    SELECT @message='Общая стоимость '+

            CAST(@s AS CHAR(6))

    PRINT @message

-- переход к следующему клиенту--

    FETCH NEXT FROM klient\_cursor

    INTO @id\_kl, @firm, @fam

END

CLOSE klient\_cursor

DEALLOCATE klient\_cursor

**Пример 7**. Разработать прокручиваемый курсор для клиентов из Москвы. Если номер телефона начинается на 1, удалить клиента с таким номером и в первой записи курсора заменить первую цифру в номере телефона на 4.

DECLARE @firm     VARCHAR(50),

        @fam      VARCHAR(50),

        @tel      VARCHAR(8),

        @message  VARCHAR(80)

PRINT '  Список клиентов'

DECLARE klient\_cursor CURSOR GLOBAL SCROLL

KEYSET FOR

        SELECT Фирма, Фамилия, Телефон

        FROM Клиент

        WHERE Город='Москва'

        ORDER BY Фирма, Фамилия

FOR UPDATE

OPEN klient\_cursor

FETCH NEXT FROM klient\_cursor

    INTO  @firm, @fam, @tel

WHILE @@FETCH\_STATUS=0

BEGIN

    SELECT @message='Клиент '+@fam+

        '  Фирма  '+@firm '  Телефон '+ @tel

    PRINT @message

-- если номер телефона начинается на 1,

-- удалить клиента с таким номером

    IF @tel LIKE ‘1%’

       DELETE Клиент

       WHERE CURRENT OF klient\_cursor

    ELSE

-- переход к следующему клиенту

    FETCH NEXT FROM klient\_cursor

           INTO @firm, @fam, @tel

END

FETCH ABSOLUTE 1 FROM klient\_cursor

    INTO @firm, @fam, @tel

-- в первой записи заменить первую цифру в

-- номере телефона на 4

   UPDATE Клиент SET Телефон=’4’ +

   RIGHT(@tel,LEN(@tel)-1))

        WHERE CURRENT OF klient\_cursor

SELECT @message='Клиент '+@fam+'  Фирма  '+

    @firm '  Телефон '+ @tel

    PRINT @message

CLOSE klient\_cursor

DEALLOCATE klient\_cursor

**Пример 8**. Использование курсора как выходного параметра процедуры. Процедура возвращает набор данных – список товаров.

CREATE PROC my\_proc

@cur CURSOR VARYING OUTPUT

AS

SET @cur=CURSOR FORWARD\_ONLY STATIC FOR

SELECT Название FROM Товар

OPEN @cur

Пример 8. Использование курсора как выходного параметра процедуры.

Вызов процедуры и вывод на печать данных из выходного курсора осуществляется следующим образом:

DECLARE @my\_cur CURSOR

DECLARE @n VARCHAR(20)

EXEC my\_proc @cur=@my\_cur OUTPUT

  FETCH NEXT FROM @my\_cur INTO @n

  SELECT @n

WHILE (@@FETCH\_STATUS=0)

BEGIN

  FETCH NEXT FROM @my\_cur INTO @n

  SELECT @n

END

CLOSE @my\_cur

DEALLOCATE @my\_cur

**3. Задание и методические указания по выполнению работы.**

**3.1. Разработка пользовательских функций**

В ранее разработанной БД MS SQL Server "Платежи студентов" создать пользовательские функции – по одной функции каждого из типов:

3.1.1 скалярные функции;

3.1.2 табличные функции Inline;

3.1.3 табличные функции Multistatement.

согласно приводимым ниже вариантам (номер варианта V соответствует номеру студента N в журнале студенческой группы V=N, если N<16 и V=N-15, если N>15).

**Варианты заданий на создание скалярных функций**

Создать пользовательскую скалярную функцию, которая будет возвращать:

1. сумму всех хранящихся платежей (без параметров);
2. сумму платежей для конкретной группы (параметр – название группы);
3. сумму платежей для конкретного студента (параметры – ФИО студента);
4. сумму платежей для конкретного города (параметр – код города);
5. количество платежей для конкретного города (параметр – код города);
6. количество улиц в конкретном городе (параметр – название города);
7. количество студентов из конкретного города (параметр – название города);
8. количество платежей конкретного студента (параметры – ФИО студента);
9. среднюю сумму платежа для конкретного студента (параметры – ФИО студента);
10. количество студентов на заданном факультете (параметр - сокращенное название факультета);
11. название группы, в которой учится студент (параметры – ФИО студента);
12. полное название факультета, на котором учится студент (параметры – ФИО студента);
13. максимальную сумму платежа студента (параметры – ФИО студента);
14. минимальную сумму платежа на заданном факультете (параметр - факультет);
15. название улицы, на которой живёт студент (параметры – ФИО студента).

**Варианты заданий на создание табличных функций Inline**

Создать пользовательскую табличную функцию Inline, которая будет возвращать следующую информацию:

1. данные о всех платежах, произведенных в заданный промежуток времени;
2. данные о всех платежах студентов заданного факультета;
3. данные о всех платежах студентов заданной группы;
4. данные о суммах оплат студентов заданного факультета;
5. телефоны студентов, производивших платежи в заданный период времени;
6. города, в которых живут студенты заданной группы;
7. города, в которых живут студенты заданного факультета;
8. ФИО студентов заданного факультета, не совершавших платежей;
9. количество студентов заданного факультета, проживающих в заданном городе (для списка городов);
10. количество студентов, проживающих в заданном городе (для списка городов);
11. ФИО студентов, имеющих платежи, превышающие заданную сумму;
12. количество студентов имеющих платежи, превышающие заданную сумму (для списка факультетов);
13. количество студентов заданного факультета (для списка факультетов);
14. ФИО и телефоны студентов заданной группы;
15. ФИО и телефоны студентов заданного факультета.

**Варианты заданий на создание табличных функций Multistatement**

Создать пользовательскую табличную функцию Multistatement, которая будет выполнять следующую обработку данных:

1. вывести данные всех студентов с годом поступления ранее заданного и переместить их в архивную таблицу;
2. вывести данные студентов заданного факультета, которые выполнили платежи в текущем году на сумму меньшую заданной и удалить данные оних их таблицы Студенты;
3. вывести сведения о студентах, которые оплатили дважды оплатили указанную услугу в текущем семестре;
4. вывести сведения о студентах, оплативших указанную услугу в текущем месяце и перевести их из одной заданной группы в другую;
5. вывести сведения о студентах (ФИО, факультет, дата рождения), достигших 18 лет в текущем семестре, перенести в таблицу с заданным именем;
6. студентам из указанного города уменьшить оплату по указанной услуге на заданный процент и вывести сведения об этих студентах;
7. вывести сведения о студентах (ФИО, факультет, группа), у которых нет заданного платежа и перенести в таблицу с заданным именем;
8. сведения о студентах (ФИО, факультет, группа), оплативших заданную услугу до заданной даты, перенести в таблицу с заданным именем;
9. вывести, а потом удалить все сведения о студентах, которые за текущий год не оплатили заданную образовательную услугу;
10. вывести сведения, а затем удалить все сведения о платежах, выполненных ранее указанного срока;
11. вывести, а потом обновить дату оплаты указанной услуги на дату последнего дня того семестра, в котором выполнялась оплата;
12. вывести сведения, а затем удалить все записи о платежах студентов-пятикурсников;
13. вывести разность средней суммы оплаты студентов двух различных заданных факультетов за заданный год;
14. вывести сведения о студентах (ФИО, факультет, группа), которые оплатили хотя бы одну образовательную услугу, оплачиваемую заданным студентом;
15. вывести сведения о студентах (ФИО, факультет, группа) и перевести на второй курс всех первокурсников: тех, кто оплатил обучение за первый курс, и студентов не выполнявших платежи за время учебы на первом курсе.

**3.2. Разработка хранимых процедур**

В ранее разработанной БД MS SQL Server "Платежи студентов" создать хранимые процедуры согласно приводимым ниже вариантам (номер варианта V соответствует номеру студента N в журнале студенческой группы V=N, если N<16 и V=N-15, если N>15).

**Варианты заданий на создание хранимых процедур.**

(***Все процедуры должны содержать входные и выходные параметры, значения параметров по умолчанию***)

Разработать хранимую процедуру:

1. с входными параметрами, которая выводит сумму платежей, уплаченную студентами заданного факультета.
2. которая возвращает сумму платежей (выходной параметр) по заданной группе (входной параметр).
3. для изменения названия группы заданного студента (входные параметры: ФИО, старая группа, новая группа).
4. для определения количества полных лет студента (выходной параметр), входные параметры: ФИО, группа.
5. которая выводит сведения о студентах заданного факультета (входной параметр), у которых сумма оплат ниже заданной (входной параметр).
6. с входными параметрами, которая выводит суммы платежей по каждому из факультетов, уплаченную студентами после заданной даты.
7. которая по заданным ФИО (задаются входными параметрами), возвращает с помощью выходных параметров год поступления и группу студента.
8. для обновления фамилии студента (входные параметры: ФИО, группа, новая фамилия).
9. для определения количества полных лет студентов заданной группы; входные параметры: группа, факультет.
10. которая изменяет название группы студентов с заданного на заданное.
11. с входными параметрами, которая выводит суммы платежей по каждому из факультетов, уплаченную студентами между двумя указанными датами.
12. которая возвращает (выходной параметр) количество студентов заданной группы (входной параметр), у которых сумма оплаты меньше заданного значения (входной параметр).
13. для обновления сокращенного и полного названий факультета и группы заданного студента (входные параметры: ФИО, группа, новый факультет, новая группа).
14. с входными параметрами, которая выводит список студентов, выполнявших платежи после указанной даты.
15. с входными и выходными параметрами, которая выводит суммы платежей по каждой из групп, уплаченных студентами до заданной даты.

**3.3. Разработка триггеров**

В ранее разработанной БД MS SQL Server "Платежи студентов" создать триггеры согласно приводимым ниже вариантам (номер варианта V соответствует номеру студента N в журнале студенческой группы V=N, если N<16 и V=N-15, если N>15):

3.3.1 DML- триггеры типа AFTER

3.3.2 DML- триггеры типа INSTEAD OF

3.3.3 DML- триггеры типа AFTER

3.3.4 DDL-триггеры.

**Варианты заданий на создание DML-триггеров типа AFTER**

Триггеры должны выполнять следующую обработку данных:

1. отправка клиенту сообщения при добавлении данных в таблицу Студенты;
2. отправка клиенту сообщения при удалении данных из таблицы Платежи;
3. отправка клиенту сообщения при изменении данных в таблице Улицы;
4. отправка клиенту сообщения при вставке строки в таблицу Платежи с суммой оплаты менее 16000;
5. отправка клиенту сообщения при попытке вставить данные в таблицу Студенты с некорректным годом поступления;
6. отправка клиенту сообщения при попытке удалить данные из таблицы Платежи с суммой оплаты не менее 2800000;
7. отправка клиенту сообщения при попытке ввода отрицательной или нулевой суммы оплаты в таблицу Платежи;
8. при удалении строки из таблицы Студенты удалить соответствующие строки из таблицы Платежи;
9. отправка клиенту сообщения при попытке изменения сокращенного названия факультета в таблице Факультеты;
10. отправка клиенту сообщения при изменении номера телефона в таблице Студенты;
11. отправка клиенту сообщения при изменении цели оплаты в таблице НазначенияОплат;
12. при удалении строки из таблицы НазначенияОплат удалить соответствующие записи из таблицы Платежи;
13. при добавлении строки в таблицу Студенты выводить сообщение о том, нужно ли для этого студента заносить данные в таблицу Платежи;
14. при добавлении строки в таблицу Улицы, выводить сообщение о том, нужно ли для этой улицы заносить новое название города в таблицу Города;
15. при удалении строки из таблицы Платежи, выводить предупреждающее сообщение со сведениями о студенте (ФИО, факультет, группа), который выполнил удаляемый платеж.

**Варианты заданий на создание DML-триггеров типа INSTEAD OF**

Триггеры должны выполнять следующую обработку данных:

1. Отменить удаление строк в таблице Студенты и вывести сообщение об ошибке.
2. Отменить вставку строк в таблицу Студенты и вывести сообщение об ошибке.
3. Отменить изменение данных в таблице Студенты и вывести сообщение об ошибке.
4. Отменить удаление строки из таблицы НазначенияОплат, если имеются связанные строки в таблице Платежи.
5. Отменить удаление строки из таблицы Студенты, если имеются связанные строки в таблице Платежи.
6. При удалении строки из таблицы Города, каскадно удалить строки из таблицы Улицы, Студенты и Платежи.
7. При удалении строки из таблицы Студенты, вывести ФИО студентов, поступивших в 2019 году.
8. При добавлении строки в таблицу Платежи, вывести ФИО студентов, у которых сумма оплаты по всем платежам меньше 100000.
9. При добавлении строки в таблицу Студенты, вывести сведения о количестве студентов из каждого города.
10. При добавлении строки в таблицу Улицы, вывести сведения о количестве улиц в каждом городе.
11. При обновлении строк в таблице Платежи оставить эту таблицу без изменений, но вывести набор строк, в которых произошло бы изменение.
12. При добавлении строки в таблицу Студенты, удалить строку со сведениями о студенте с кодом 1.
13. При удалении строк из таблицы Платежи вывести набор строк, которые были бы удалены, но оставить таблицу без изменений.
14. При удалении строки из таблицы Платежи удалять только строки с нечетным кодом студента.
15. При изменении строки в таблице Студенты выводить информацию о платежах соответствующего студента за текущий семестр.

**Варианты заданий на создание DDL-триггеров**

1. Создать DDL триггер, запрещающий удалять таблицы из БД с выводом соответствующего сообщения.
2. Создать DDL триггер, запрещающий изменять таблицы в БД с выводом соответствующего сообщения.
3. Создать DDL триггер, запрещающий создавать таблицы в БД с выводом соответствующего сообщения.
4. Создать DDL триггер, запрещающий создавать представления в БД с выводом соответствующего сообщения.
5. Создать DDL триггер, запрещающий изменять представления в БД с выводом соответствующего сообщения.
6. Создать DDL триггер, запрещающий удалять представления в БД с выводом соответствующего сообщения.
7. Создать DDL триггер, сообщающий о создании новой БД на сервере.
8. Создать DDL триггер, запрещающий удалять БД с выводом соответствующего сообщения.
9. Создать DDL триггер, запрещающий изменять БД с выводом соответствующего сообщения.
10. Создать DDL триггер, протоколирующий работу пользователей с таблицей Студенты.
11. Создать DDL триггер, протоколирующий работу пользователей с таблицей Платежи.
12. Создать DDL триггер, протоколирующий работу пользователей с таблицей Города.
13. Создать DDL триггер, протоколирующий работу пользователей с таблицей Улицы.
14. Создать DDL триггер, протоколирующий работу пользователей с таблицей Факультеты.
15. Создать DDL триггер, протоколирующий работу пользователей с таблицей НазначенияОплат.

**3.4. Разработка курсоров**

В ранее разработанной БД MS SQL Server "Платежи студентов" создать курсоры:

3.4.1 динамический;

3.4.2 для выборки данных;

3.4.3 для модификации данных.

Самостоятельно определить таблицы, над данными которых будут производиться манипуляции, назначение и выполняемые операции создаваемых курсоров; протестировать их работу.

**4. Требования к содержанию отчета.**

После выполнения заданий в отчет по работе включить:

1. *код инструкций на языке Transact-SQL*, полученных при создании программных объектов БД, с *подробными комментариями* по используемым операторам, функциям и другим элементам синтаксиса;
2. *описание разработанных и системных объектов MS SQL Server*, обеспечивающих обработку данных;
3. *результаты обработки* данных с помощью разработанных программных объектов БД.

**5. Контрольные вопросы (ответы нужно знать для защиты работы).**

1. MS SQL Server: группы и примеры встроенных функций.
2. MS SQL Server: общая характеристика и примеры скалярных пользовательских функций.
3. MS SQL Server: общая характеристика и примеры табличных пользовательских функций типа Inline.
4. MS SQL Server: общая характеристика и примеры табличных пользовательских функций типа Multi-statement.
5. MS SQL Server: создание и использование хранимых процедур.
6. Задание и использование переменных в языке Transact-SQL.
7. Общая характеристика встроенных функций языка Transact-SQL.
8. Основные встроенные математические функции языка Transact-SQL, их использование.
9. Встроенные строковые функции языка Transact-SQL, их использование.
10. Встроенные функции языка Transact-SQL для работы с датой и временем, их использование.
11. Встроенные функции конфигурирования языка Transact-SQL, их использование.
12. Встроенные функции системы безопасности языка Transact-SQL, их использование.
13. Встроенные функции управления метаданными языка Transact-SQL, их использование.
14. Общая характеристика типов пользовательских функций языка Transact-SQL, для каких операций по обработке данных они используются?
15. Общий синтаксис скалярных функций языка Transact-SQL.
16. Общий синтаксис внедренных табличных функций (функций Inline) языка Transact-SQL.
17. Общий синтаксис табличных функций с множеством инструкций (функций Multi-statement) языка Transact-SQL.
18. Общая характеристика и использование хранимых процедур в клиент-серверных приложениях.
19. Типы хранимых процедур в среде MS SQL Server, особенности их использования.
20. Общий синтаксис хранимых процедур языка Transact-SQL.
21. Что такое триггеры, и как они используются?
22. DML-триггеры, их типы и использование.
23. DDL-триггеры, их использование.
24. Назначение и использование курсоров.
25. Управляющие конструкции языка Transact-SQL.