Лабораторная работа № 5
«Использование хранимых процедур и функций для организации сложных выборок данных»

Цель работы: Использование хранимых процедур и функций для организации сложных выборок данных.

Краткие теоретические сведения

Сценарии

Сценарием считается совокупность операторов, хранящихся в виде отдельного файла, который может вызываться на выполнение и использоваться повторно. Сценарии храняться в виде текстовых файлов. В программе SQL Server Management Studio предусмотрен ряд инструментальных средств, позволяющих упростить написание сценариев:

- в основном меню ввода запросов этой программы применяется цветовое выделение, которое позволяет не только распознавать ключевые слова, но и определять их назначение;
- технология Intellisense (обеспечивает набор возможностей, которые упрощают доступ к справочным сведениям по языку. Во время программирования нет необходимости покидать редактор кода для поиска элементов языка, можно сохранить контекст, найти необходимые сведения, вставить элементы языка прямо в код и даже позволить функции IntelliSense завершать ввод) и пошаговый отладчик

Сценарии обычно рассматриваются как отдельная единица работы. Это означает, что в обычных обстоятельствах сценарий либо выполняется полностью, либо вообще не выполняется. В сценариях могут использоваться операторы, локальные и системные переменные.

В качестве примера рассмотрим сценарий, предназначенный для вставки строк в БД MYDB.

```
USE MYDB;
DECLARE @Ident  int;

INSERT INTO Orders
(CustomerNo, OrderDate, EmployeeID)
VALUES
(1, GetDATE(), 1);
SELECT @Ident =@@IDENTITY;
INSERT INTO  OrdersDetails
(OrderID. PartNo. Description. UnitPrice. Qty)
VALUES
(@Ident, '2R2416 ', 'Cylinder Head', 1300, 2)
SELECT , 'The OrderID of the Inserted rowb is ' + CONVERT (varchar(8), @Ident);
```

В этом сценарии применяются шесть различных операторов, которые показывают, какие разнообразные действия могут осуществляться в сценариях: системные и локальные переменные, операторы USE и INSERT, а также две версии оператора SELECT – с операцией присваивания и обычный.

Оператор ***USE*** сообщает какая база данных должна стать текущей. Действие, осуществляемое этим оператором, отражается во всех местах сценария, в которых используются заданные по умолчанию значения для той части полностью уточненного имени объекта, которая применяется для обозначения имени базы данных.

С помощью оператора ***DECLARE*** можно объявить одну или сразу несколько переменных. Синтаксис оператора

DECLARE @ <variable name> <variable type> [=<value>] [,
@ <variable name> <variable type> [=<value>] [,
@ <variable name> <variable type> [=<value>]]]

В приведенном выше примере объявлена одна локальная переменная @Ident.

Задание значений переменных. В настоящее время в языке SQL предусмотрены два способа задания переменной значений: оператор SET (если должна быть выполнена простая операция присваивания значения переменной) и оператор SELECT (если присваивание значения переменной основано на запросе).

Оператор SET обычно используется для задания значений переменных в такой форме, какая встречается в процедурных языках, например,

SET @TotalCost = 10

SET @TotalCost = @UnitCost * 1.1

Во всех этих операторах непосредственно осуществляются операции присваивания, в которых используются либо явно заданные значения, либо другие переменные. С помощью оператора SET невозможно присвоить переменной значение, полученное с помощью запроса; запрос должен быть выполнен отдельно, и только после этого полученный результат может быть присвоен с помощью оператора SET.

Оператор SELECT обычно используется для присваивания значений переменным, если источником информации, которая должны быть сохранена в переменной, является запрос

USE MYDB;

DECLARE @Test money;

SELECT @Test = MAX(UnitPrice) FROM Sales;

SELECT @Test

Пакеты

Пакет – это средство группировки операторов T-SQL в виде одной логической единицы. Все операторы, содержащиеся в пакете, объединяются в один план выполнения, поэтому в процессе синтаксического анализа рассматриваются все операторы, и все они должны успешно пройти проверку синтаксиса, так как в противном случае не будет выполнен ни один из операторов. Но успешное завершение проверки синтаксиса не исключает возможности возникновения ошибок на этапе прогона пакета.

Все сценарии можно считать отдельными пакетами. Чтобы разделить сценарий на несколько пакетов, можно воспользоваться оператором ***GO,*** который характеризуется следующими особенностями: этот оператор

- должен находиться на отдельной строке;
- вызывает компиляцию всех операторов от начала сценария или от предыдущего оператора GO (в зависимости от того, что ближе);
- не оператор языка T-SQL, а скорее команда, распознаваемая утилитами SQL Server с интерфейсом командной строки.

Ошибки в пакетах подразделяются на две категории – синтаксические ошибки и ошибки во время выполнения программы.

Пакеты имеют несколько назначений, но для всех областей их применения характерна общая особенность – пакеты используются для выполнения каких-либо действий, которые должны быть осуществлены либо заблаговременно, либо отдельно от всех прочих действий в сценарии.

Операторы, которые должны безусловно включать в состав отдельных пакетов: CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, CREATE VIEW.

Операторы управления ходом выполнения

В T-SQL предусмотрены следующие программные средства управления ходом выполнения программы:

- IF . . . ELSE

- GOTO
- WHILE
- WAITFOR
- TRY/CATCH
- CASE

IF ... ELSE

В этом операторе должно быть вычислено логической выражение, результатом которого могут быть только True (истина) или False (ложь). Есть много способов создать такое выражение: наибольшее количество способов связано с операторами отношения, такими как >, <, = и NOT. Операторы отношения могут быть соединены со строковыми функциями, другими математическими выражениями или операциями отношения между значениями локальных или даже системных переменных. Можно, также, разместить оператор SELECT внутри блока IF. . .ELSE, если возвращается единственное значение.

Основная конструкция IF. . .ELSE может выглядеть следующим образом:

IF выражение THEN

Оператор, выполняемый при условии True

ELSE

Оператор, выполняемый при условии False

Операторы IF. . .ELSE могут быть вложены друг в друга. Для группирования операторов внутри IF. . .THEN их нужно окружить блоком BEGIN . . . END.

Пример ниже показывает, как может выглядеть подобный код:

IF выражение THEN

BEGIN

Оператор 1, выполняемый при условии True

Оператор 1, выполняемый при условии True

Оператор 1, выполняемый при условии True

END

ELSE

Оператор, выполняемый при условии False

Оператор CASE

В случае, если выражение в зоне проверки может иметь не только простые результаты True или False, а несколько потенциальных результатов, то может применяться оператор CASE Синтаксис оператора:

CASE выражение

WHEN значение THEN

Оператор1

[[WHEN значение 2 THEN]

[оператор2]]

[[ELSE]

Прочие ситуации]

END

За ключевым словом CASE стоит проверяемое выражение. Результатом этого выражения может быть значение переменной или столбец, возвращенный оператором T-SQL, или любое корректное в SQL-сервере выражение. Проверяемое выражение затем используется для определения значения, которое далее будет сопоставлено определенному оператору WHEN.

В рамках оператора CASE можно иметь произвольное количество операторов WHEN. При этом нет необходимости предусматривать каждое возможное значение условия. Если значение, полученное при вычислении CASE-выраженння, совпадает с условным значением, заданным в операторе WHEN, то будут выполняться только операторы внутри данного блока WHEN. С помощью условия ELSE можно предусмотреть возможность выполнения кода для любых значений проверяемого выражения, не определенных в операторах WHEN. Значение проверяемого выражения, для которого не выполнено ни одно из условий операторов WHEN,

подпадает под действие условия ELSE, и для него выполняется код, определенный для этого условия.

Оператор CASE может применяться как самостоятельно, так и внутри операторов SELECT или UPDATE. Имеется возможность присвоить значения столбцу в наборе записей на основании оператора CASE и результирующего значения. Оператор CASE не может быть частью оператора DELETE.

Команда ***GOTO***

Структуры IF...ELSE и CASE управляют порядком выполнения операторов, основываясь на результатах вычисления булевого выражения. Команда GOTO является безусловной. Она передает выполнение непосредственно к оператору, следующему после метки, которая на него указывает.

В SQL Server метки являются неисполняемыми операторами, имеющими следующий синтаксис:

имя_метки:

Имя метки должно удовлетворять правилам, принятым для идентификаторов. Сама команда GOTO имеет очень простой синтаксис:

GOTO имя_метки

Оператор ***WHILE*** помогает организовать цикл. В этом операторе до начала каждого прохода по циклу проверяется некоторое условие. Если перед очередным проходом по циклу проверка условия приводит к получению значения TRUE, осуществляется проход по циклу, в противном случае выполнения оператора завершается.

Синтаксис оператора:

WHILE <Boolean expression>
<SQL statement>
[BEGIN
< statement block>
[BREAK]
<SQL statement> < statement block>
[CONTINUE]
END]

С помощью оператора WHILE можно обеспечить выполнение в цикле только одного оператора.

Оператор BREAK позволяет выйти из цикла, не ожидая того, как будет выполнен проход до конца цикла и произойдет повторная проверка условного выражения.

Оператор CONTINUE обеспечивает переход в начало цикла WHILE. Сразу после обнаружения оператора CONTINUE в цикле, независимо от того, где он находится, происходит переход в начало цикла и повторное вычисление условного выражения (а если значение этого выражения больше не равно TRUE, осуществляется выход из цикла).

Оператор ***WAIRFOR*** ожидает наступление того момента времени, который указан в параметре его вызова.

WAIRFOR
DELAY <'time '> | TIME <'time '>

Можно указывать либо явно заданное время суток (TIME), в которое с помощью этого оператора должно быть осуществлено определенное действие, либо указан промежуток времени, в течение которого необходимо организовать ожидание (DELAY) .

Блок TRY…CATCH

Блок TRY…CATCH предназначен для обработки исключительных ситуаций (ошибок).

Обработчик состоит из блока TRY, где находится выполняемый код, и блока CATCH, в котором происходит перехват ошибки.

Синтаксис

BEGIN TRY { sql_statement | statement_block } END TRY BEGIN CATCH { sql_statement | statement_block } END CATCH [;]

Хранимая процедура

Хранимая процедура - это набор операторов T-SQL, который компилируется системой SQL Server в единый «план исполнения». Хранимые процедуры T-SQL аналогичны процедурам в других языках программирования в том смысле, что они допускают входные параметры и возвращают выходные значения в виде параметров или сообщения о состоянии (успешное или неуспешное завершение). Все операторы процедуры обрабатываются при вызове процедуры. Они могут использоваться различными пользователями для согласованного повторяемого выполнения одинаковых задач и даже в различных приложениях.

Ваше приложение может взаимодействовать с SQL Server двумя способами: можно запрограммировать в приложении отправку операторов T-SQL от клиента на SQL Server или можно создать хранимые процедуры, которые хранятся и выполняются на сервере. При отправке операторов T-SQL на сервер, эти операторы передаются через сеть и перекомпилируются SQL Server каждый раз, когда происходит их запуск. Используя хранимые процедуры, можно выполнять их путем вызова из приложения с помощью одного оператора. Хранимые процедуры можно использовать и для проверки какого-либо условия, так как в нее можно включить условные операторы (например, конструкции IF и WHILE).

Имеется три типа хранимых процедур: системные, расширенные и простые определяемые пользователем процедуры.

Системные хранимые процедуры предоставляет SQL Server, и они имеют префикс sp_. Они используются для управления SQL Server и вывода на экран информации о базах данных и пользователях.

Расширенные хранимые процедуры являются динамически подключаемыми библиотеками (DLL), которые может динамически загружать и выполнять SQL Server. Обычно их пишут на C или C++, и они исполняют процедуры, внешние относительно SQL Server. Расширенные хранимые процедуры имеют префикс xp_. Они выполняются непосредственно в адресном пространстве SQL Server, и вы можете программировать их, используя интерфейс прикладного программирования (API) SQL Server Open Data Services. Расширенные хранимые процедуры пишут вне SQL Server. Закончив разработку расширенной хранимой процедуры, вы регистрируете ее в SQL Server с помощью операторов T-SQL или через Enterprise Manager.

Простые определяемые пользователем хранимые процедуры создаются пользователем и настраиваются для выполнения тех задач, которые требуются данному пользователю. Вы не должны использовать префикс sp_ при создании определяемых пользователем хранимых процедур. Разумнее назвать процедуру просто myproc.

Существует три метода создания хранимой процедуры:

- использование оператора T-SQL CREATE PROCEDURE,
- использование Enterprise Manager
- использование мастера Create Stored Procedure Wizard.

Использование оператора CREATE PROCEDURE

Оператор CREATE PROCEDURE имеет следующий синтаксис:

CREATE PROC[EDURE] *имя_процедуры*

[{@имя_параметра тип_данных}] [= по умолчанию] [OUTPUT] [,...,n]

AS оператор(ы) t-sql

Создадим какую-либо хранимую процедуру. Эта процедура будет выбирать (и возвращать) три колонки данных для каждой строки таблицы Orders, в которой дата колонки ShippedDate больше даты колонки RequiredDate. Хранимая процедура может быть создана только в текущей базе данных, к которой осуществляется доступ, поэтому сначала следует указать эту базу данных с помощью оператора USE. Прежде чем создать эту процедуру, нужно выяснить также, существует ли хранимая процедура с именем, которое мы хотим использовать. Если она существует, то ее нужно удалить и затем создать новую процедуру с этим именем. Ниже показан набор T-SQL, используемый для создания этой процедуры:

```
USE Nortwind
GO
```

```
IF EXISTS       (SELECT   name
            FROM     sysobjects
 WHERE    name = "LateShipments" AND type="P")
DROP PROCEDURE LateShipments
GO
CREATE PROCEDURE LateShipments
AS
SELECT RequiredDate, ShippedDate Shippers.CompanyName
FROM Orders, Shippers
WHERE           ShippedDate > RequiredDate AND
Orders.ShipVia = Shippers.ShipperlD
GO
```

Если запустить данный набор, то будет создана хранимая процедура. Для запуска хранимой процедуры просто обратитесь к ней по имени, как это показано ниже:

```
LateShipments
GO
```

Если оператор, вызывающий данную процедуру, входит в пакет операторов и не является первым оператором этого пакета, то вы должны использовать вместе с вызовом процедуры ключевое слово EXECUTE (сокращается до «EXEC»), как это показано в следующем примере:

```
SELECT      getdate()
EXECUTE LateShipments
GO
```

Вы можете использовать ключевое слово EXECUTE, даже если процедура выполняется как первый оператор в пакете или является единственным оператором, который у вас выполняется.

Теперь добавим к хранимой процедуре ***входной параметр***, чтобы можно было передавать данные в эту процедуру. Чтобы задать входные параметры в хранимой процедуре, укажите список этих параметров с символом @ перед именем каждого параметра, т.е. @имя параметра. Вы можете задать в хранимой процедуре до 1024 параметров. В нашем примере мы создадим параметр с именем @shipperName. При запуске хранимой процедуры укажем имя компании-грузоотправителя (shipperName), и результатом запроса будут строки только для этого грузоотправителя.

```
USE Northwind
GO
IF EXISTS  (SELECT   name
FROM     sysobjects
WHERE    name = "LateShipments" AND   type="P")
DROP PROCEDURE LateShipments
GO
CREATE PROCEDURE LateShipments @shipperName char(40)
AS
SELECT RequiredDate,   Shipped Date, Shippers.CompanyName
FROM Orders, Shippers
WHERE           ShippedDate  > RequiredDate AND
Orders.ShipVia  = Shippers.ShipperlD AND
Shippers.CompanyName = @shipperName
GO
```

Для запуска этой хранимой процедуры нужно указать входной параметр. Если параметр не указан, SQL Server выведет сообщение об ошибке.

Чтобы получить строки для грузоотправителя Грузоотправитель_1, выполните следующие операторы:

```
USE Northwind
```

```
GO
EXECUTE LateShipments "Грузоотпрвитель_1"
GO
```

Можно задать для параметра значение по умолчанию, которое будет использоваться, когда этот параметр не указан в обращении к процедуре. Например, чтобы использовать для вашей хранимой процедуры значение по умолчанию Пример. Текст создаваемой процедуры изменится следующим образом:

```
USE Northwind
GO
IF EXISTS  (SELECT   name
FROM     sysobjects
WHERE    name = "LateShipments" AND type="P")
DROP PROCEDURE LateShipments GO
CREATE PROCEDURE LateShipments ©shipperName char(40) = "Пример"
AS SELECT RequiredDate, Shipped Date, Shippers.CompanyName
FROM Orders, Shippers
WHERE          ShippedDate  > RequiredDate AND
Orders.ShipVia  = Shippers.ShipperlD AND
Shippers.CompanyName = @shipperName GO
```

Теперь при запуске процедуры LateShipments без входного параметра (@shipperName) по умолчанию для этого параметра будет использоваться значение Пример.

Для ***возврата значения параметра*** хранимой процедуры в вызывающую программу используйте ключевое слово OUTPUT после имени этого параметра. Чтобы сохранить значение в переменной, которую можно использовать в вызывающей программе, используйте при вызове хранимой процедуры ключевое слово OUTPUT. Для примера создадим хранимую процедуру, которая выбирает цену единицы указанного продукта. Входной параметр @prod_id - это идентификатор продукта, а выходной параметр @unit price - это возвращаемое значение цены единицы продукта. В вызывающей программе будет объявлена локальная переменная с именем @price, которая будет использоваться для сохранения возвращаемого значения. Ниже приводится набор операторов, используемый для создания хранимой процедуры GetUnitPrice:

```
USE Northwind
GO
IF EXISTS   (SELECT   name
FROM     sysobjects
WHERE    name = "GetUnitPrice" AND  type="P")
DROP PROCEDURE GetUnitPrice
GO
CREATE PROCEDURE GetUnitPrice @prod_id int, @unit_price money OUTPUT
AS
SELECT @imit_price = UnitPrice
FROM Products
WHERE ProductID = @prod id
GO
```

Прежде чем использовать переменную в вызове хранимой процедуры, вы должны объявить эту переменную в вызывающей программе. Например, в следующей программе мы сначала объявим переменную @price и присвоим ей тип данных money (который должен быть совместим с типом данных выходного параметра), а затем выполним хранимую процедуру:

```
DECLARE @price money
EXECUTE GetUnitPrice 77, @unit_price = @price OUTPUT
PRINT CONVERT(varchar(6), @price)
GO
```

Оператор PRINT возвращает значение в переменную @price. Оператор CONVERT использовался для преобразования значения @price в данные типа varchar, чтобы их можно было напечатать как строку, как символьный тип данных или как тип данных, которые могут быть неявно преобразованы в символьный тип, что требуется для оператора PRINT.

В хранимой процедуре и в вызывающей программе для выходных переменных использовались различные имена, чтобы было легче следить за положением этих переменных в примере и чтобы показать, что можно использовать различные имена.

При обращении к хранимой процедуре можно задавать входное значение в выходном параметре. Это означает, что это значение будет введено в хранимую процедуру, где это значение может быть изменено или использовано для операций и затем новое значение возвращается в вызывающую программу. Чтобы поместить входное значение в выходной параметр, нужно просто присвоить это значение соответствующей переменной в вызывающей программе перед выполнением процедуры или выполнить запрос, который считывает значение в переменную и затем передает эту переменную в хранимую процедуру.

Использование ***локальных переменных*** внутри хранимой процедуры. Для создания локальных переменных используется ключевое слово DECLARE. При создании локальной переменной нужно задать для нее имя и тип данных, а также поставить перед именем переменной символ @. При объявлении переменной ей первоначально присваивается значение NULL.

Локальные переменные можно объявлять в пакете, в сценарии (или вызывающей программе) или в хранимой процедуре. Переменные часто используются в хранимых процедурах для хранения значений, которые будут проверяться в условном операторе или возвращаться оператором RETURN хранимой процедуры. Переменные в хранимых процедурах часто используются как счетчики. Область действия локальной переменной в хранимой процедуре начинается с точки объявления этой переменной и заканчивается выходом изданной процедуры. После выхода из хранимой процедуры эта переменная уже недоступна для использования.

Рассмотрим пример хранимой процедуры, содержащей локальные переменные. Эта процедура вставляет пять строк в таблицу с помощью циклической конструкции WHILE. Сначала создадим таблицу с именем mytable и затем создадим хранимую процедуру InsertRows. В этой процедуре будем использовать локальные переменные @loop counter и @start_val, которые объявим вместе и разделим запятой.

```
USE MyDB
GO
CREATE TABLE mytable  (column 1 int, column2char(10))
GO
CREATE PROCEDURE InsertRows @start value int
AS
DECLARE    @loop_counter int. @start_val int
SET        @start_val = @start_value - 1
SET        @loop_counter = 0
WHILE (@loop_counter < 5)
BEGIN
INSERT INTO mytable VALUES (@start_val + 1, "new row")
PRINT (@start_val)
SET @start_val = @start_val + 1
SET @loop_counter = @loop counter + 1
END
GO
```

Теперь выполним эту хранимую процедуру с начальным значением 1:

```
EXECUTE InsertRows 1
GO
```

Вы увидите пять значений, выведенных для @start_val: 0, 1, 2, 3 и 4. После завершения хранимой процедуры переменные @loop_counter и @start_val уже недоступны.

Вы можете возвращаться из любой точки хранимой процедуры в вызывающую программу с помощью ключевого слова ***RETURN***, обеспечивающего безусловный выход из процедуры. RETURN можно также использовать для выхода из пакета или блока операторов. При выполнении оператора RETURN в хранимой процедуре работа процедуры прекращается в этой точке, и происходит переход к следующему оператору вызывающей программы. Операторы, следующие после RETURN в хранимой процедуре, не выполняются. С помощью RETURN вы можете также возвращать целое значение.

Рассмотрим пример использования RETURN просто для выхода из хранимой процедуры. Создадим модифицированную версию процедуры GetUnitPrice, которая проверяет, было ли задано входное значение, и если нет, то выводит сообщение для пользователя и возвращается в вызывающую программу. Для этого определим входной параметр со значением по умолчанию NULL и затем будем проверять это значение в процедуре; если входной параметр имеет значение NULL, это означает, что входное значение не задано. Ниже приводится пример удаления и повторного создания этой процедуры:

```
USE Northwind
GO
IF EXISTS   (SELECT   name
FROM     sysobjects
WHERE    name = "GetUnitPrice" AND type = "P")
DROP PROCEDURE GetUnitPrice
GO
CREATE PROCEDURE GetUnitPrice @prod id int = NULL
AS
IF @prod_id lS NULL
BEGIN
PRINT "Please enter a product ID number"
RETURN
END
ELSE
BEGIN
SELECT UnitPrice
FROM Products
WHERE ProductID = @prod id
END
GO
```

Запустим GetUnitPrice без ввода входного значения и посмотрим результаты. Для запуска этой хранимой процедуры нужно указать оператор EXECUTE, поскольку вызов процедуры не является первым оператором этого пакета. Используйте следующую последовательность:

```
PRINT "Before procedure"
EXECUTE GetUnitPrice
PRINT "After procedure returns from stored procedure"
GO
```

Второй оператор PRINT включен для того, чтобы показать, что после оператора RETURN хранимой процедуры выполнение данного пакета продолжается с оператора PRINT.

Рассмотрим использование RETURN для возврата значения в вызывающую программу. Возвращаемое значение должно быть целым. Это может быть константа или переменная. Нужно объявить в вызывающей программе переменную, в которой будет храниться возвращаемое значение для дальнейшего использования в этой программе. Например, следующая процедура возвратит значение 1, если цена единицы продукции для продукта, указанного во входном параметре, меньше $100; иначе она возвратит значение 99.

```
CREATE PROCEDURE CheckUnitPrice @prod_id int
AS
IF (SELECT UnitPrice
FROM Products
WHERE ProductID = @prod id) < 100
RETURN 1
ELSE
RETURN 99
GO
```

Для вызова этой хранимой процедуры и использования возвращаемого значения объявите в вызывающей программе переменную и приравняйте ее возвращаемому значению хранимой процедуры (указав значение 66 в ProductID для входного параметра):

```
DECLARE @retum_val int
EXECUTE ©return J/al = CheckUnitPrice 66
IF (@return val =1) PRINT "Unit price is less than $100"
GO
```

Убедитесь в том, что вы задали целый тип данных, когда объявляете переменную, которая используется для хранения значения, возвращаемого оператором RETURN, поскольку этот оператор возвращает целое значение.

Использование ***оператора SELECT***, находящегося внутри хранимой процедуры для возвращаемых из процедуры значений. Рассмотрим пример. Создадим хранимую процедуру с именем PrintUnitPrice, которая возвращает цену единицы продукции для продукта, указанного во входном параметре (с помощью идентификатора этого продукта). Используется следующая последовательность:

```
CREATE PROCEDURE PrintUnitPrice @prod_id int
AS
SELECT      ProductID, UnitPrice FROM Products
WHERE       ProductID = @prod_id
GO
```

Вызовите эту процедуру со значением входного параметра 66, как это показано ниже:

```
PrintUnitPrice66
GO
```

Для возврата значений переменных с помощью оператора SELECT, укажите после SELECT имя соответствующей переменной. Рассмотрим пример. Создадим хранимую процедуру CheckUnitPrice, которая возвращает значение переменной, а также зададим заголовок выводимой колонки:

```
USE Northwind
GO
IF EXISTS      (SELECT   name
FROM    sysobjects
WHERE   name = "CheckUnitPrice" AND type = "P")
DROP PROCEDURE CheckUnitPrice
GO
CREATE PROCEDURE CheckUnitPrice @prod_id INT
AS
DECLARE @var1 int
IF        (SELECT   UnitPrice
FROM    Products
WHERE   ProductID = @prod id) > 100
SET     @var1 = 1
ELSE
SET @var1 = 99
```

```
SELECT'Variable 1"=@var1
PRINT "Can add more T-SQL statements here"
GO
```

Вызовите эту процедуру со значением входного параметра 66, как это показано ниже:

```
CheckUnitPrice 66
GO
```

Использование Management Studio

процесс создания хранимой процедуры Transact-SQL в среде SQL Server Management Studio с помощью обозревателя объектов.

1. В обозревателе объектов подключитесь к экземпляру компонента Database Engine и разверните его.
2. Последовательно разверните узел Базы данных, базу данных, которой принадлежит хранимая процедура, и узел Программирование.
3. Щелкните правой кнопкой мыши элемент Хранимые процедуры, а затем выберите команду Создать хранимую процедуру.
4. В меню Запрос щелкните Задание значений для параметров шаблона.
5. В диалоговом окне Задание значений для параметров шаблона в столбце Значение содержатся предлагаемые значения параметров. Предложенные значения можно принять или заменить новыми, после чего необходимо нажать кнопку ОК.
6. В редакторе запросов замените инструкцию SELECT текстом создаваемой процедуры.
7. Для проверки синтаксиса в меню Запрос выберите пункт Синтаксический анализ.
8. Чтобы создать хранимую процедуру, в меню Запрос выберите пункт Выполнить.
9. Чтобы сохранить скрипт, в меню Файл выберите пункт Сохранить. Можно принять предложенное имя файла или заменить его новым, после чего следует нажать кнопку Сохранить.

Управление хранимыми процедурами

Хранимая процедура может быть *выполнена* с помощью оператора EXECUTE:

```
EXEC[UTE]
[@СтатусВозврата =] ИмяПроцедуры
[ [@параметр=] {Значение | Выражение} [OUTPUT] ] [,...]
```

При использовании выходного параметра, его необходимо описать с использование ключевого слова *OUTPUT* при создании процедуры, а также использовать это слово и при указании соответствующего параметра при вызове процедуры. В противном случае процедура не передает выходное значение.

Параметр *@СтатусВозврата* используется для получения значения кода возврата из хранимой процедуры, выполненного с помощью оператора RETURN *Статус*. Пользователю рекомендовано использовать положительные значения статуса.

Для *изменения* существующей процедуры используется оператор ALTER PROC, параметры этой команды аналогичны параметрам команды создания процедуры.

При использовании оператора ALTER PROCEDURE сохраняются исходные полномочия, установленные для данной хранимой процедуры, а изменения не влияют любые зависимые процедуры или триггеры.

Синтаксис оператора ALTER PROCEDURE аналогичен синтаксису оператора CREATE PROCEDURE:

```
ALTER PROCEDURE]  имя_процедуры
[ {@имя_параметра тип_данных} ] [= по умолчанию] [OUTPUT] [,...,n]
AS оператор(ы) t-sql
```

В операторе ALTER PROCEDURE можно переписать всю хранимую процедуру, внося нужные изменения.

Для *переименования* необходимо использовать специальную системную хранимую процедуру:

sp_rename ‘ИмяОбъекта’ ‘НовоеИмяОбъекта’.

Для *удаления* хранимой процедуры используется команда Transact-SQL:

DROP PROC ИмяПроцедуры.

Вы не сможете восстановить хранимую процедуру после ее удаления. Если вам нужно использовать удаленную процедуру, вы должны полностью воссоздать ее с помощью оператора CREATE PROCEDURE. Все полномочия по удаленной хранимой процедуре будут утрачены, и они должны быть предоставлены снова. Ниже приводится пример использования DROP PROCEDURE для удаления процедуры GetUnitPrice:

```
USE Northwind
GO
DROP PROCEDURE GetUnitPrice
GO
```

Пользовательские функции

Пользовательские функции – представляют собой упорядоченное множество операторов SQL, которые заранее оптимизированы, откомпилированы и могут быть использованы для выполнения работы в виде единого модуля. Основное различие между хранимыми процедурами и пользовательскими функциями заключается в том, как в них осуществляется возврат полученных результатов. Кроме того, функции могут вызываться на выполнение как непосредственно встроенные в запрос (например, могут входить в состав оператора SELECT).

Пользовательские функции подразделяются на два вида:

- возвращающие скалярные значения,
- возвращающие таблицу.

Скалярные пользовательские функции возвращают скалярный (однозначный) результат, такой как строка или число, которые могут относится к любому допустимому типу данных SQL Server (включая определенные пользователем), кроме таких как курсоры, временные отметки и таблицы.

Скалярные функции могут использоваться везде, где может использоваться возвращаемое функцией значение с соответствующим типом данных. Это может быть список столбцов в фразе WHERE оператора SELECT, выражение, условие ограничения в определении таблицы или даже описание типа данных в столбце таблицы.

Табличные пользовательские функции возвращают таблицу и не заменяют хранимые процедуры или представления, но в определенных ситуациях они могут предоставить более широкие возможности, которые трудно реализовать с помощью этих объектов.

Функция является агрегатной, если она оперирует некоторым количеством значений, а возвращает единственное итоговое значение.

Кроме того, пользовательские функции различаются по детерминизму. Детерминизм функции определяется постоянством ее результатов.

Функция является детерминированной, если при одном и том же заданном входном значении она всегда возвращает один и тот же результат. Так, встроенная функция DATEADD является детерминированной – добавление трех дней к дате 20 апреля 2010 г. всегда дает дату 23 апреля 2010 г.

Функция является недетерминированной, если она может возвращать различные значения при одном и том же заданном входном значении. Так, встроенная функция GETDATE является недетерминированной. Она будет при каждом вызове возвращать различные значения.

Детерминизм пользовательской функции не зависит от того, является ли она скалярной или табличной, – функции обоих этих типов могут быть как детерминированными, так и недетерминированными.

Создание пользовательских функций

Пользовательские функции создаются с помощью соответствующей разновидности команды CREATE.

Оператор CREATE для скалярных пользовательских функций:

```
CREATE FUNCTION имя_функции ([список_параметров])
RETURNS тип_данных
AS
BEGIN
    [операторы_tsql]
    RETURN (возвращаемое_значение)
END
```

Имя_функции должно удовлетворять правилам, действующим для идентификаторов. Хотя список_параметров в описании оператора CREATE имеет следующий синтаксис:

@имя_параметра тип_данных [= значение_по_умолчанию]

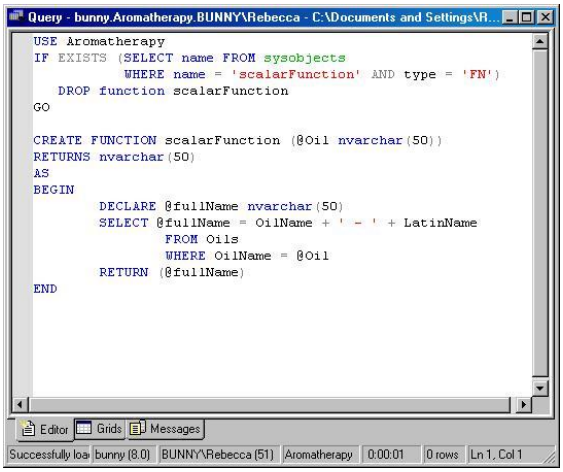
Имя_параметра должно соответствовать правилам, принятым для идентификаторов, и начинаться с символа @. Пользовательские функции могут иметь до 1024 входных параметров. Выходные параметры пользовательские функции не поддерживают – единственным значением, возвращаемым функцией, является результат ее выполнения. Заметим, что список_параметров является необязательным, но наличие скобок обязательно.

Фраза RETURNS определяет тип возвращаемых функцией значений.

Операторы BEGIN...END, которыми ограничиваются операторы_tsql, составляющие тело функции, являются обязательными, даже если тело функции состоит из одного оператора RETURN.

Для создания скалярной пользовательской функции:

1. В обозревателе объектов подключитесь к экземпляру компонента Database Engine и разверните его.
2. Последовательно разверните узел Базы данных, базу данных, которой принадлежит пользовательская функция, и узел Программирование и выберите узел Функции.
3. Щелкните правой кнопкой мыши на элементе Скалярная или Табличная функция, а затем выберите команду Создать функцию.
4. В редакторе запросов вы увидете шаблон на создание пользовательской процедуры, замените инструкции текстом создаваемой функции.
5. Для проверки синтаксиса в меню Запрос выберите пункт Синтаксический анализ.
6. Чтобы создать пользовательскую функцию, в меню Запрос выберите пункт Выполнить.
7. Чтобы сохранить скрипт, в меню Файл выберите пункт Сохранить. Можно принять предложенное имя файла или заменить его новым, после чего следует нажать кнопку Сохранить.

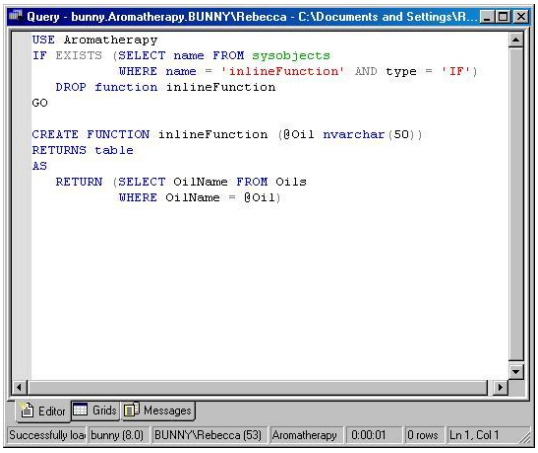


Оператор CREATE FUNCTION поддерживает создание двух различных типов табличных функций: подставляемых и многооператорных. Тело подставляемой табличной функции

состоит из единственного оператора SELECT, в то время как многооператорная табличная функция может состоять из любого числа операторов Transact-SQL.

Синтаксис для подставляемой табличной функции является усеченной разновидностью оператора CREATE FUNCTION. Блок BEGIN...END отсутствует, и нет никаких других операторов, кроме RETURN:

```
CREATE FUNCTION имя_функции (список_параметров)
RETURNS таблица
AS
RETURN (оператор_выборки)
```

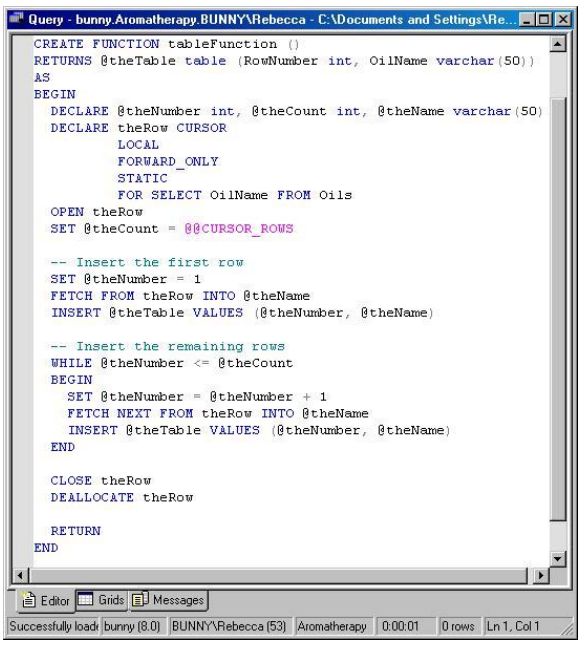


Синтаксис оператора CREATE FUNCTION для многооператорной табличной функции сочетает элементы синтаксиса для скалярной и для подставляемой функций:

```
CREATE FUNCTION имя_функции (список_параметров)
RETURNS @локальная_табличная_переменная TABLE
    (определение_таблицы)
AS
BEGIN
    операторы_tsql
    RETURN
END
```

Подобно скалярным функциям, в многооператорной табличной функции команды Transact-SQL располагаются внутри блока BEGIN...END. Поскольку блок может содержать несколько операторов SELECT, в фразе RETURNS вы должны явно определить таблицу, которая будет возвращаться.

Поскольку оператор RETURN в многооператорной табличной функции всегда возвращает таблицу, заданную во фразе RETURNS, он должен выполняться без аргументов, – например, RETURN, а не RETURN @myTable.



Применение пользовательских функций

Синтаксис вызова скалярных функций схож с синтаксисом, используемым для встроенных функций Transact-SQL:

имя_владельца.имя_функции([список_параметров])

Имя_владельца для скалярной функции является обязательным. Вы не можете использовать синтаксис с именованными параметрами (например, @имя_параметра = значение), а также не можете не указывать (опускать) параметры, но вы можете применять ключевое слово DEFAULT для указания значения по умолчанию, как вы это делаете для хранимых процедур.

Для скалярной функции вы также можете использовать оператор EXECUTE:

EXECUTE @возвращаемое_значение = имя_функции(список_параметров)

Если вы используете оператор EXECUTE для пользовательской функции, вам не нужно указывать имя_владельца. В этом синтаксисе вы можете использовать именованные параметры:

EXECUTE @возвращаемое_значение = имя_функции @параметр = значение

[, @параметр = значение [,...]]

Если вы используете именованные параметры, параметры не обязательно должны следовать в том порядке, в котором они указаны в объявлении функции, но вам необходимо указать все параметры; нельзя опускать ссылку на параметр для использования значения по умолчанию.

Для табличных пользовательских функций, как подставляемых, так и многооператорных, должен всегда использоваться тот же синтаксис, что и для встроенных функций:

имя_функции([список_параметров])

Имя_владельца здесь указывать не требуется, но необходимо включить все определенные параметры, как и при вызове любой пользовательской функции.

Скалярные пользовательские функции могут использоваться везде, где допустимо использовать тип данных, который они возвращают. Табличные пользовательские функции могут быть использованы только во фразе FROM оператора SELECT.

Примеры.

А) Используйте скалярную функцию в операторе PRINT

1.Введите в панели инструментов редактора запросов следующий оператор SQL в окне Query (Запрос):

PRINT dbo.scalarFunction('German Chamomile')

2.Нажмите кнопку Execute Query (Выполнить запрос) в панели инструментов редактора запросов, который выполнит оператор и отобразит результат.

Б) Используйте скалярную функцию в операторе SELECT

1.Введите в панели инструментов редактора запросов следующий оператор SQL в окне Query (Запрос):

2. SELECT OilID, dbo.scalarFunction(OilName)
FROM Oils

3.Нажмите кнопку Execute Query (Выполнить запрос) в панели инструментов редактора запросов, который выполнит оператор и отобразит результат.

В) Используйте табличную функцию в операторе SELECT

1.Введите в панели инструментов редактора запросов следующий оператор SQL в окне Query (Запрос):

SELECT * FROM tableFunction()

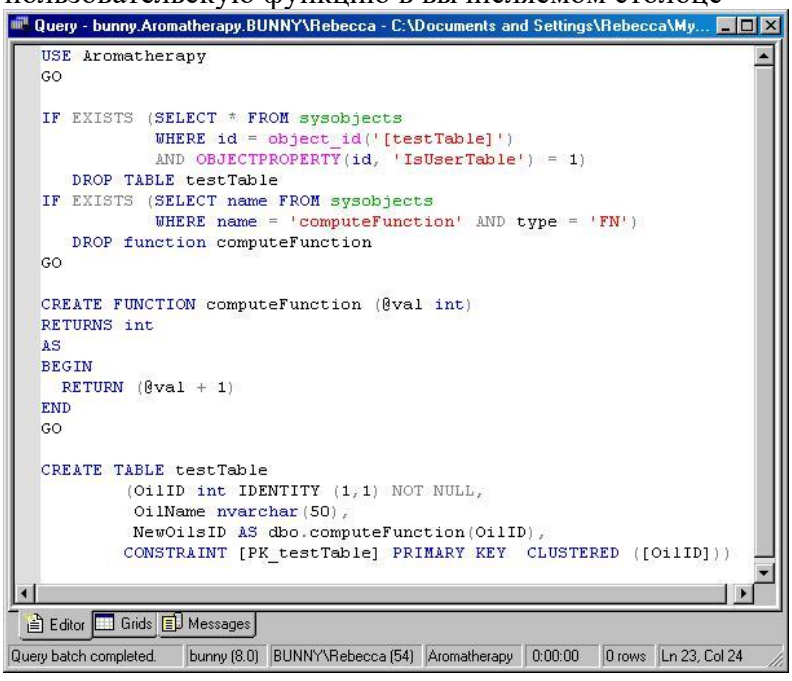
2. Нажмите кнопку Execute Query (Выполнить запрос) в панели инструментов, который выполнит оператор и отобразит результат.

Применение пользовательских функций в определениях таблиц

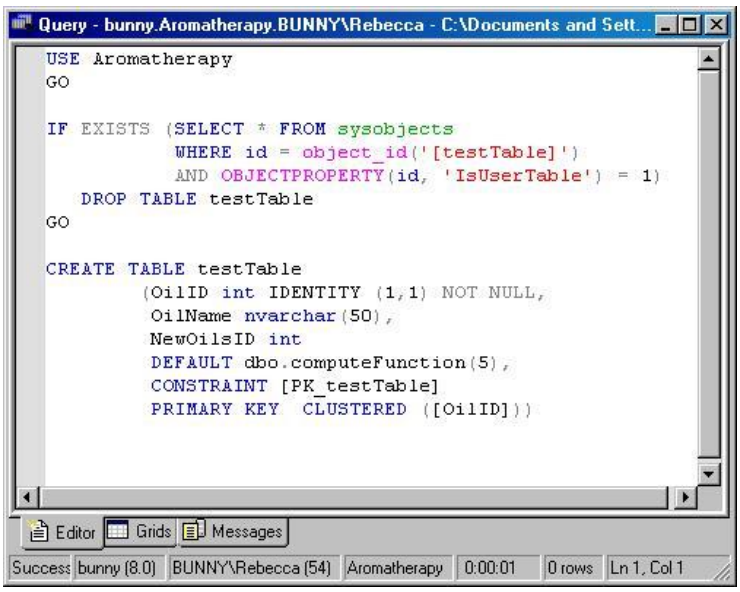
Пользовательские функции могут быть использованы в определениях таблиц, предоставляемых владельцем таблицы, но на параметры, используемые в этих функциях, накладываются некоторые ограничения.

При использовании в качестве типа данных для вычисляемого столбца, параметры пользовательской функции должны быть либо другими столбцами в таблице, либо константами. Это справедливо и в том случае, если пользовательская функция используется в качестве проверочного ограничения типа CHECK. Если пользовательская функция используется как значение по умолчанию для столбца, параметры должны быть константами.

Г) Примените пользовательскую функцию в вычисляемом столбце



Д) Примените пользовательскую функцию в определении DEFAULT



Лабораторная работа рассчитана на 2 часа аудиторных занятий и состоит в изучении теоретического материала и получении практических навыков по созданию хранимых процедур. Сдача лабораторной работы заключается в ответах на контрольные вопросы и демонстрации индивидуального задания.

Порядок выполнения работы:

1. Изучение теоретических сведений по теме лабораторной работы
2. Выполнение задания:

а) Создать хранимые процедуры в соответствии с требованиями индивидуального варианта задания на разработку курсового проекта.

б) Создать пользовательские функции (*назначение выбирается по своему усмотрению*):

- возвращающие скалярные значения,
- возвращающие таблицу.

3. Оформление отчета по лабораторной работе, который должен содержать:
   - Название, цель и задание к лабораторной работе
   - Результаты выполнения задания: скрипты созданных хранимых процедур и пользовательских функций

Скрипт

	код_преподавателя	Преподаватель	Кафедра
1	9	Белов С.К.	Кафедра базы данных

	День_недели	Номер_пары	Начало	Окончание	Предмет	Тип_занятия	Аудитория	Преподаватель
1	Понедельник	1	08:30:00.0000000	10:00:00.0000000	Базы данных	лекции	201	Белов С.К.
2	Понедельник	2	10:10:00.0000000	11:40:00.0000000	Базы данных	компьют	301	Белов С.К.

	Аудитория	Тип_аудитории	Посадочных_мест
1	101	лекции	150
2	102	лекции	120

	Нагрузка
1	0.00

	Проверка
1	Группа помещается

Пара	Начало	Конец	Предмет	Тип	Группа	Аудитория

4. Защита лабораторной работы

Скрипт:

```
-- ================================================
-- ЛАБОРАТОРНАЯ РАБОТА №5 - ХРАНИМЫЕ ПРОЦЕДУРЫ И ФУНКЦИИ
-- БАЗА ДАННЫХ: Расписание_занятий_Л
-- ВАРИАНТ: 8
-- ================================================

USE Расписание_занятий_Л;
GO

-- ================================================
-- ЧАСТЬ а): ХРАНИМЫЕ ПРОЦЕДУРЫ СОГЛАСНО ВАРИАНТУ 8
-- ================================================

-- 1. Процедура: У каких преподавателей можно прослушать курс лекций по «I-ой» дисциплине
CREATE PROCEDURE sp_ПреподавателиПоДисциплине
    @код_дисциплины INT
AS
BEGIN
    SELECT DISTINCT
        п.код_преподавателя,
        п.ФИО AS Преподаватель,
        к.наименование AS Кафедра
    FROM Преподаватели п
    INNER JOIN Расписание р ON п.код_преподавателя = р.код_преподавателя
    INNER JOIN Предметы пр ON р.код_предмета = пр.код_предмета
    INNER JOIN Тип_занятий_аудиторий т ON р.код_типа_занятий = т.код
    INNER JOIN Кафедры к ON п.код_кафедры = к.код_кафедры
    WHERE пр.код_предмета = @код_дисциплины
      AND т.наименование = 'лекции';
END;
GO

-- 2. Процедура: Расписание занятий для студентов I-ой группы
CREATE PROCEDURE sp_РасписаниеГруппы
    @код_группы INT
AS
BEGIN
    SELECT
        р.день_недели AS День_недели,
        р.номер_занятия AS Номер_пары,
        пер.время_начала AS Начало,
        пер.время_окончания AS Окончание,
        пр.наименование AS Предмет,
        т.наименование AS Тип_занятия,
        ау.номер_ауд AS Аудитория,
        прп.ФИО AS Преподаватель
    FROM Расписание р
    INNER JOIN Период_проведения_занятий пер ON р.номер_занятия = пер.номер_занятия
    INNER JOIN Предметы пр ON р.код_предмета = пр.код_предмета
    INNER JOIN Тип_занятий_аудиторий т ON р.код_типа_занятий = т.код
    INNER JOIN Аудитории ау ON р.номер_ауд = ау.номер_ауд
    INNER JOIN Преподаватели прп ON р.код_преподавателя = прп.код_преподавателя
    WHERE р.код_группы = @код_группы
    ORDER BY
        CASE р.день_недели
            WHEN 'Понедельник' THEN 1
            WHEN 'Вторник' THEN 2
            WHEN 'Среда' THEN 3
            WHEN 'Четверг' THEN 4
            WHEN 'Пятница' THEN 5
            WHEN 'Суббота' THEN 6
        END,
        р.номер_занятия;
END;
```

```
GO

-- 3. Процедура: Из таблицы Аудитории выбрать строки по условию: количество посадочных
мест > 50
CREATE PROCEDURE sp_АудиторииБольше50
AS
BEGIN
    SELECT
        а.номер_ауд AS Аудитория,
        т.наименование AS Тип_аудитории,
        а.количество_посадочных_мест AS Посадочных_мест
    FROM Аудитории а
    INNER JOIN Тип_занятий_аудиторий т ON а.код_типа_занятий = т.код
    WHERE а.количество_посадочных_мест > 50
    ORDER BY а.количество_посадочных_мест DESC;
END;
GO

-- ==================================================
-- ЧАСТЬ б): ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ
-- ==================================================

-- 1. Скалярная функция: Расчет общей нагрузки преподавателя в часах в неделю
CREATE FUNCTION dbo.ufn_НагрузкаПреподавателя
(
    @код_преподавателя INT
)
RETURNS DECIMAL(5,2)
AS
BEGIN
    DECLARE @общая_нагрузка DECIMAL(5,2);

    SELECT @общая_нагрузка = COUNT(*) * 1.5
    FROM Расписание
    WHERE код_преподавателя = @код_преподавателя;

    RETURN ISNULL(@общая_нагрузка, 0);
END;
GO

-- 2. Скалярная функция: Проверка возможности размещения группы в аудитории
CREATE FUNCTION dbo.ufn_ПроверитьВместимость
(
    @код_группы INT,
    @номер_аудитории INT
)
RETURNS NVARCHAR(50)
AS
BEGIN
    DECLARE @результат NVARCHAR(50);
    DECLARE @численность_группы INT;
    DECLARE @вместимость_аудитории INT;

    SELECT @численность_группы = численность_группы
    FROM Группа
    WHERE код_группы = @код_группы;

    SELECT @вместимость_аудитории = количество_посадочных_мест
    FROM Аудитории
    WHERE номер_ауд = @номер_аудитории;

    IF @численность_группы IS NULL OR @вместимость_аудитории IS NULL
        SET @результат = 'Ошибка: данные не найдены';
    ELSE IF @численность_группы <= @вместимость_аудитории
        SET @результат = 'Группа помещается';
```

```
    ELSE
        SET @результат = 'Аудитория мала для группы';

    RETURN @результат;
END;
GO

-- 3. Табличная функция: Расписание преподавателя на указанный день
CREATE FUNCTION dbo.ufn_РасписаниеПреподавателяНаДень
(
    @код_преподавателя INT,
    @день_недели NVARCHAR(15)
)
RETURNS TABLE
AS
RETURN
(
    SELECT
        р.номер_занятия AS Пара,
        пер.время_начала AS Начало,
        пер.время_окончания AS Конец,
        пр.наименование AS Предмет,
        т.наименование AS Тип,
        гр.наименование AS Группа,
        ау.номер_ауд AS Аудитория
    FROM Расписание р
    INNER JOIN Период_проведения_занятий пер ON р.номер_занятия = пер.номер_занятия
    INNER JOIN Предметы пр ON р.код_предмета = пр.код_предмета
    INNER JOIN Тип_занятий_аудиторий т ON р.код_типа_занятий = т.код
    INNER JOIN Группа гр ON р.код_группы = гр.код_группы
    INNER JOIN Аудитории ау ON р.номер_ауд = ау.номер_ауд
    WHERE р.код_преподавателя = @код_преподавателя
      AND р.день_недели = @день_недели
);
GO

-- ================================================
-- ТЕСТИРОВАНИЕ
-- ================================================

-- Тест процедур
PRINT '=== ТЕСТ ХРАНИМЫХ ПРОЦЕДУР ===';
EXEC sp_ПреподавателиПоДисциплине @код_дисциплины = 1;
EXEC sp_РасписаниеГруппы @код_группы = 1;
EXEC sp_АудиторииБольше50;

-- Тест функций
PRINT '=== ТЕСТ ФУНКЦИЙ ===';
SELECT dbo.ufn_НагрузкаПреподавателя(1) AS Нагрузка;
SELECT dbo.ufn_ПроверитьВместимость(1, 101) AS Проверка;
SELECT * FROM dbo.ufn_РасписаниеПреподавателяНаДень(1, 'Понедельник');

    PRINT '=== ЛАБОРАТОРНАЯ РАБОТА №5 ВЫПОЛНЕНА ===';
```

Контрольные вопросы:

Конечно, вот ответы на контрольные вопросы в чистом тексте, без специальных символов.

---

1. Назначение хранимых процедур?

Хранимые процедуры предназначены для выполнения предопределенных последовательностей SQL-операторов, которые хранятся на сервере базы данных. Их основное назначение:

Повышение производительности: процедуры компилируются и кэшируются, что ускоряет их повторное выполнение.

Инкапсуляция логики: позволяют выносить сложную бизнес-логику на уровень СУБД, централизуя её.

Безопасность: можно предоставлять пользователям права на выполнение процедуры, не давая им прямого доступа к таблицам.

Сокращение сетевого трафика: вместо отправки нескольких запросов клиент отправляет один вызов процедуры.

Упрощение поддержки: логика хранится в одном месте, и её изменения не требуют модификации клиентских приложений.

2. Классификация хранимых процедур?

Хранимые процедуры можно классифицировать по нескольким признакам:

По способу создания: системные (встроенные в СУБД) и пользовательские (созданные администратором или разработчиком).

По наличию параметров: без параметров, с входными параметрами, с выходными параметрами, с входными и выходными параметрами.

По возвращаемому результату: возвращающие набор данных (результирующий набор), возвращающие скалярное значение (через OUTPUT-параметр или возвращаемое значение), не возвращающие данных (выполняющие действия, например, вставку или обновление).

По языку реализации: в разных СУБД процедуры могут быть написаны на SQL или на других языках (например, T-SQL в MS SQL Server, PL/pgSQL в PostgreSQL, PL/SQL в Oracle).

3. Как можно осуществлять управление ходом процедуры?

Управление ходом выполнения хранимой процедуры осуществляется с помощью конструкций управления потоком. К ним относятся:

Условные операторы: IF ELSE, CASE.

Циклы: WHILE, LOOP (в зависимости от СУБД).

Операторы перехода: GOTO (не рекомендуется к использованию).

Обработка ошибок: конструкции TRY CATCH (в MS SQL Server) или EXCEPTION (в PostgreSQL, Oracle).

Операторы для работы с курсорами: OPEN, FETCH, CLOSE для построчной обработки результатов.

4. Способы создания хранимых процедур?

Создать хранимую процедуру можно двумя основными способами:

С помощью SQL-скриптов: используя оператор CREATE PROCEDURE с последующим определением тела процедуры на языке, поддерживаемом СУБД.

Пример:

```
CREATE PROCEDURE GetCustomer @CustomerID INT
AS
BEGIN
   SELECT * FROM Customers WHERE CustomerID = @CustomerID
END
```

С помощью графических инструментов: используя среды разработки, такие как Microsoft SQL Server Management Studio (SSMS), pgAdmin для PostgreSQL или Oracle SQL Developer. Эти инструменты предоставляют диалоговые окна и мастера для создания и редактирования процедур.

5. Способы передачи и возврата параметров в хранимую процедуру?

Передача параметров:

Входные параметры (IN): передают данные в процедуру. Указываются в списке параметров.

Выходные параметры (OUTPUT/OUT): позволяют процедуре возвращать данные обратно в вызывающий код. Помчаются ключевым словом OUTPUT (T-SQL) или OUT (другие СУБД).

Входно-выходные параметры (INOUT): параметр, который используется и для передачи значения в процедуру, и для возврата измененного значения из процедуры.

Возврат данных:

Через выходные параметры.

С помощью оператора RETURN, который возвращает целочисленное значение (часто используется для возврата кода ошибки или статуса).

Путем возврата одного или нескольких результирующих наборов (таблиц) с помощью оператора SELECT внутри процедуры.

6. В чем заключается управление хранимыми процедурами?

Управление хранимыми процедурами включает в себя следующие действия:

Создание: CREATE PROCEDURE.

Изменение (модификация): ALTER PROCEDURE.

Удаление: DROP PROCEDURE.

Выполнение: EXEC или EXECUTE (в T-SQL), CALL (в PostgreSQL, MySQL).

Просмотр метаданных: получение информации о процедуре (ее текст, параметры, зависимости) с помощью системных представлений или хранимых процедур.

Управление разрешениями: предоставление (GRANT) или отзыв (REVOKE) прав на выполнение процедуры (EXECUTE) для пользователей и ролей.

7. Назначение механизма пользовательских функций?

Механизм пользовательских функций предназначен для создания повторно используемых подпрограмм, которые возвращают единственное скалярное значение или таблицу. В отличие от хранимых процедур, функции можно использовать в контексте SQL-выражений. Их основное назначение:

Модуляризация кода: создание логических блоков, выполняющих специфические вычисления или операции.

Использование внутри SQL-запросов: функции можно вызывать прямо в выражениях, что невозможно с процедурами.

Упрощение сложных запросов: позволяют скрыть сложную логику вычислений внутри функции.

8. Способы создания пользовательских функций

Создание пользовательских функций аналогично созданию процедур:

С помощью SQL-скриптов: используя оператор CREATE FUNCTION с последующим определением тела функции.

Скалярная функция возвращает одно значение.

Пример:

```
CREATE FUNCTION dbo.GetTotalSales(@CustomerID INT)
RETURNS MONEY
AS
```

```
BEGIN
   DECLARE @Total MONEY
   SELECT @Total = SUM(OrderTotal) FROM Orders WHERE CustomerID = @CustomerID
   RETURN @Total
END
```

Табличная функция возвращает таблицу.

Пример:

```
CREATE FUNCTION dbo.GetCustomerOrders(@CustomerID INT)
RETURNS TABLE
AS
RETURN (SELECT * FROM Orders WHERE CustomerID = @CustomerID)
```

С помощью графических инструментов: использование сред разработки для визуального создания и редактирования функций через интерфейс.

Конечно, вот ответы на контрольные задания.

---

1. Опишите средства управления ходом выполнения хранимой процедурой

Для управления ходом выполнения хранимой процедуры в T-SQL используются следующие основные средства:

Блоки BEGIN...END: Группируют несколько операторов в единый логический блок, который выполняется как одно целое.

Условные операторы IF...ELSE: Позволяют выполнять операции по условию. Если условие после IF истинно, выполняется следующий за ним оператор или блок. Если ложно, выполняется блок после ELSE.

Оператор CASE: Обеспечивает условный переход в рамках одного выражения, часто используется внутри оператора SELECT.

Циклы WHILE: Организует повторное выполнение оператора или блока операторов, пока заданное условие истинно. Внутри цикла используются операторы BREAK для досрочного выхода и CONTINUE для перехода к следующей итерации.

Конструкция TRY...CATCH: Обеспечивает обработку ошибок. Код, который может вызвать ошибку, помещается в блок BEGIN TRY...END TRY. Если ошибка происходит, управление передается в блок BEGIN CATCH...END CATCH, где ее можно обработать.

Оператор RETURN: Немедленно завершает выполнение процедуры. Может возвращать целочисленное значение, которое обычно используется для передачи статуса выполнения (0 — успех, ненулевое значение — код ошибки).

---

2. Какие системные переменные используются хранимыми процедурами?

Системные переменные в T-SQL называются системными функциями. Они предоставляют информацию о текущем сеансе, системе и последних выполненных операциях. Наиболее часто используемые из них:

@@ROWCOUNT: Возвращает количество строк, затронутых последним выполненным оператором. Часто используется для проверки успешности операций вставки, обновления или удаления.

@@ERROR: Возвращает номер ошибки, вызванной последним выполненным оператором. Если ошибки не было, возвращает 0. После каждого оператора значение сбрасывается, поэтому его нужно проверять сразу.

@@IDENTITY: Возвращает последнее значение идентификатора, сгенерированного для любого столбца с свойством IDENTITY в текущем сеансе.

SCOPE_IDENTITY(): Аналогична @@IDENTITY, но возвращает последнее значение идентификатора, сгенерированное в текущей области видимости (например, внутри выполняемой хранимой процедуры), что делает ее более безопасной.

@@TRANCOUNT: Возвращает количество активных транзакций для текущего соединения.

@@VERSION: Возвращает информацию о версии, редакции и сборке экземпляра SQL Server.

@@SPID: Возвращает идентификатор серверного процесса для текущего пользовательского сеанса.

---

3. Составьте перечень системных хранимых процедур и их назначения в MS SQL Server

Системные хранимые процедуры в MS SQL Server имеют префикс "sp_" и используются для администрирования и получения метаданных.

sp_help: Выводит информацию об любом объекте базы данных (таблице, представлении, процедуре и т.д.), указанном в качестве параметра.

sp_helptext: Показывает определение указанного пользовательского объекта базы данных, например, текст хранимой процедуры, функции или триггера.

sp_who и sp_who2: Предоставляют информацию о текущих пользователях, сеансах и процессах в экземпляре SQL Server. sp_who2 является более расширенной версией.

sp_lock: Отображает информацию о блокировках в реальном времени. Для получения подобной информации в современных версиях SQL Server предпочтительнее использовать динамическое административное представление sys.dm_tran_locks.

sp_rename: Позволяет переименовать пользовательский объект в базе данных, например, таблицу, столбец или хранимую процедуру.

sp_configure: Отображает или изменяет глобальные конфигурационные параметры сервера.

sp_tables: Возвращает список всех объектов (таблиц, представлений), которые могут быть запрошены в текущей базе данных.

sp_columns: Возвращает подробную информацию о столбцах для указанной таблицы или представления.

sp_depends: Показывает зависимости объекта, то есть все объекты, которые ссылаются на указанный объект, и все объекты, на которые ссылается указанный объект.

sp_executesql: Выполняет динамическую SQL-строку или batch. Позволяет использовать параметры, что повышает безопасность и производительность по сравнению с простым оператором EXEC.