Транзакции

- Транзакция это последовательность операций, объединенных в единый логический рабочий модуль.
- Механизм транзакций позволяет контролировать выполнение операций в этом логическом модуле и производить откаты (отмену уже сделанной операции), если этого требует логика приложения.
- Рабочий модуль должен соответствовать основным требованиям к транзакциям, сокращенно называемые ACID (Atomicity, Consistency, Isolation, Durability)

- Atomicity (атомарность)Логика приложения должна предполагать, что должны быть проделаны либо все изменения данных, входящие в транзакцию либо ни одного;
- Consistensy(постоянство) После завершения транзакции не должна быть нарушена целостность данных,система не может оказаться в неком промежуточном состоянии;
- Isolation (изолированность)Изменения, производимые в рамках одной транзакции, изолируются от других (конкурирующих) транзакций;(4-уровня изоляции:0-двум процессам запрещается изменять одни и те же данные; 1-запрещено считывание пока идут изменения; 2-в промежутках чтения в одной TRAN не допускаются изменения в другой; 3-запрещаются в это время вставки и удаления)
- Durability (устойчивость)После завершения транзакции все сделанные изменения будут сделаны в любом случае, даже если во время этого процесса произошел сбой системы или потеря связи – после восстановления работоспособности SQL сервер обращается к журналу транзакций и производит изменения.

Запуск транзакции

SQL сервер позволяет запустить явную, автоматически совершаемую или неявную транзакцию

- Explicit (явная) транзакция предваряется выражением BEGIN TRANSACTION
- Autocommit (автоматически совершаемая) транзакция – режим, в котором работает SQL сервер по умолчанию, каждая отдельная инструкция T-SQL совершается (изменения в данные вносятся физически) после отрабатывания инструкции. Не нужно указывать никаких ключевых слов, чтобы начать такую транзакцию
- Implicit (неявная) транзакция. Такой режим транзакции устанавливается инструкцией SET IMPLICIT_TRANSACTIONS ON, следующая за этой инструкцией конструкция T-SQL автоматически начинает новую транзакцию. Когда эта транзакция завершается, следующее выражение начинает новую транзакцию.

Завершение транзакции.

- Для завершения транзакции используется конструкция COMMIT
- Если все прошло успешно, конструкция COMMIT гарантирует, что все изменения будут сделаны на физическом уровне.
- Если же во время выполнения транзакции произошла ошибка, используется конструкция ROLLBACK – данные возвращаются к первоначальному состоянию, или к некоторой точке сохранения, системные ресурсы освобождаются.

Синтаксис

- SAVE TRAN [SACTION] { *savepoint_name* | *@savepoint_variable* } – объявить savepoint
- BEGIN TRAN [SACTION] [*transaction_name* | *@tran_name_variable*
  [WITH MARK ['*description*']]]
- ROLLBACK [TRAN [SACTION]
  [*transaction_name* | *@tran_name_variable*
  | *savepoint_name* | *@savepoint_variable*]]
- COMMIT [TRAN [SACTION] [*transaction_name* | *@tran_name_variable*]]

Примеры:

```
CREATE TABLE ImplicitTran (Cola int PRIMARY KEY,
                           Colb char(3) NOT NULL)
SET IMPLICIT_TRANSACTIONS ON
/* Первая implicit транзакция начнется конструкцией INSERT*/
INSERT INTO ImplicitTran VALUES (1, 'aaa')
INSERT INTO ImplicitTran VALUES (2, 'bbb')
/* Commit first transaction */
COMMIT TRANSACTION
/* Вторая implicit транзакция начнется конструкцией SELECT */
SELECT COUNT(*) FROM ImplicitTran
INSERT INTO ImplicitTran VALUES (3, 'ccc')
SELECT * FROM ImplicitTran
/* Commit second transaction */
COMMIT TRANSACTION
SET IMPLICIT_TRANSACTIONS OFF
```

В autocommit режиме в случае возникновения ошибки компиляции SQL сервер делает rollback всему пакету инструкций. При возникновении runtime error откат не производится, не выполняется лишь инструкция, в которой произошел runtime error:

```
--example 1(autocommit ошибка компиляции)
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb   CHAR(3))
INSERT INTO TestBatch VALUES (1, 'aaa')
INSERT INTO TestBatch VALUES (2, 'bbb')
INSERT INTO TestBatch VALUSE (3, 'ccc')  /* Syntax error */
SELECT * FROM TestBatch   /* Returns no rows */
--example 2(autocommit ошибка runtime error)
CREATE TABLE TestBatch (Cola INT PRIMARY KEY, Colb CHAR(3))
INSERT INTO TestBatch VALUES (1, 'aaa')
INSERT INTO TestBatch VALUES (2, 'bbb')
INSERT INTO TestBatch VALUES (1, 'ccc')  /* Duplicate key error */
SELECT * FROM TestBatch   /* Returns rows 1 and 2 */
```

В SQL Server контроль имени объекта производится в (во время выполнения) execution time. Поэтому в следующем примере будет runtime error, а не compile error:

```
USE pubs
GO
CREATE TABLE TestBatch (Cola INT PRIMARY KEY,
Colb CHAR(3))
GO
INSERT INTO TestBatch VALUES (1, 'aaa')
INSERT INTO TestBatch VALUES (2, 'bbb')
INSERT INTO TestBch VALUES (3, 'ccc')  /* Table name error */
GO
SELECT * FROM TestBatch   /* Returns rows 1 and 2 */
GO
```

Обработка исключений

- @@ERROR возвращает номер ошибки, возникшей при выполнении предыдущего SQL-выражения или 0, если ошибок не было. Сообщения, соответствующие каждому коду ошибки хранятся в таблице sysmessages
- @@ROWCOUNT возвращает количество строк, затронутых предыдущим запросом
- RAISEERROR – генерирует исключение

RAISERROR **(** { *msg_id* | *msg_str* } { **,** *severity* **,** *state* }
[**,** *argument* [**,**...*n*]] **)**
[WITH *option* [**,**...*n*]]

severity - тип проблемы (использовать 11-16)

state – произвольное число от 1 до 127

Схема обработки ошибок в explicit транзакции:

```
begin tran
Update Authors
    set contact=1 where au_id=1000
Save transaction Authors_done
Update AU_titles
    set au_num=au_num+3 where au_id=1000
if @@error <> 0 or @@rowcount >1
begin
    raiserror('Couldn''t update ',16,1)
    print ‘error_update’
    rollback tran Authors_done
    return
end
commit tran
Print ‘tran’’s ok!’
```

Триггеры

Триггер - особая разновидность хранимой процедуры, которая выполняется в тех случаях, когда пользователь пытается добавить, удалить или модифицировать данные. Триггеры часто используются для реализации бизнес-логики и проверки целостности данных. В триггере определяется тип запроса (INSERT, DELETE или UPDATE) и таблица, с которыми он связан.

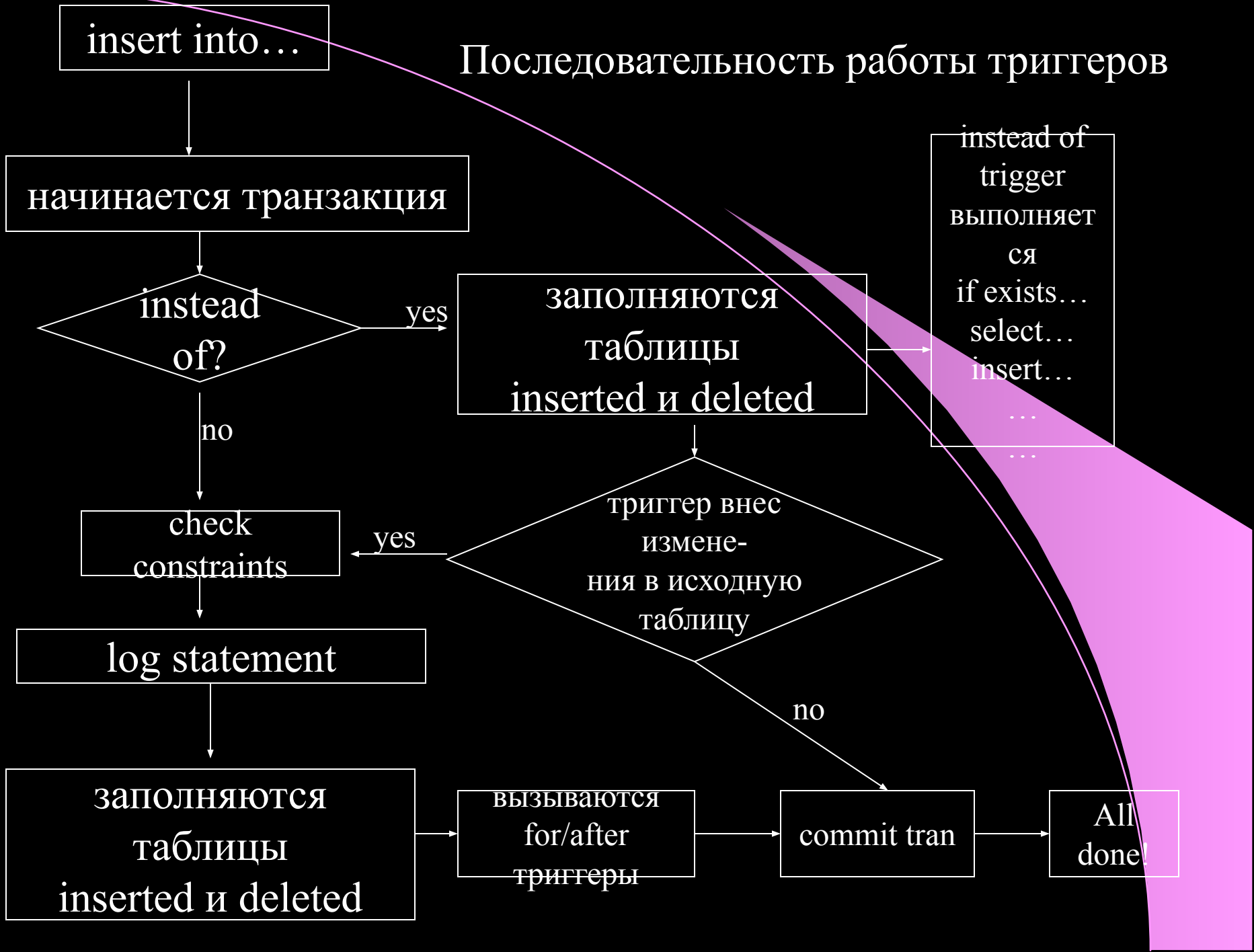
Во время выполнения триггера создаются две специальные таблицы - INSERTED и DELETED. В них находятся записи, соответственно добавляемые или удаляемые запросами в таблице, для которой создан триггер.

Синтаксис

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
   { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE ] }
      [ WITH APPEND ]
      [ NOT FOR REPLICATION ]
      AS
      [ { IF UPDATE ( column )
         [ { AND | OR } UPDATE ( column ) ]
            [ ...n ]
      | IF ( COLUMNS_UPDATED ( ) { bitwise_operator }
updated_bitmask )
            { comparison_operator } column_bitmask [ ...n ]
      } ]
      sql_statement [ ...n ]
   }
}
```

- FOR (или AFTER) и INSTEAD OF устанавливают тип триггера. FOR(AFTER) – все операции в триггере выполняются **после** того, как отработал запрос, на который наложен триггер. INSTEAD OF – триггер выполняется **вместо** запроса (таблицы deleted и inserted создаются и заполняются, однако модификация данных в первичной таблице не производится).
- WITH APPEND – для совместимости с предыдущими версиями, доступна только если MS SQL сервер работает в режиме совместимости с предыдущими версиями (в предыдущих версиях нельзя было в явном виде создать несколько однотипных триггеров на одной и той же таблице)
- IF UPDATE(COLUMN NAME) – true, если колонка была затронута исходным выражением

Последовательность работы триггеров



Пример 1

```
CREATE TABLE my_table (a int NULL, b int NULL)
GO
ALTER TRIGGER my_trig ON my_table
FOR INSERT
AS
PRINT '1'
GO
ALTER TRIGGER my_trig1 ON my_table
FOR INSERT
AS
PRINT '2'
--ROLLBACK TRAN
GO
insert into my_table(a) values(1)
insert into my_table(a,b) values(1,2)
```

Пример 2

```
CREATE TRIGGER tri_FirmsInsert ON Firms
FOR INSERT
AS
/* занесем Departments из init-таблицы */
INSERT INTO Deps (F__ID, Dep_Name, Dep_Parent)
SELECT I.F_ID, DI.DI_Name, DI.DI_Parent
FROM INSERTED I, DepsInit DI
GO
```

```
CREATE TRIGGER tri_FirmsUpdate ON Firms
FOR Update
AS
UPDATE T
SET T.F_ID = I.F_ID
FROM Deps T, INSERTED I, DELETED D
WHERE T.F_ID = D.F_ID
AND I.F_Name = D.F_Name
AND I.F_ID <> D.F_ID
GO
```

```
CREATE TRIGGER tri_FirmsDelete ON Firms
FOR Delete
AS
DELETE T FROM Deps T, DELETED D
WHERE T.F_ID = D.F_ID
GO
```

Пример 3

```
CREATE TRIGGER CustomerHasOrders
ON Customers
FOR DELETE
AS
 IF EXISTS
      (
            SELECT * FROM DELETED D
            INNER JOIN ORDERS O
            ON D.CustomerID = O.CustomerID
      )
      BEGIN
            RAISERROR('Customer has order history, delete failed',16,1)
            ROLLBACK TRAN
      END
```

- Оператор return прекращает выполнение триггера
- Триггер может быть рекурсивным (по умолчанию это отключено в настройках)
- Глубина рекурсии не более 32, для определения глубины рекурсии (nested level) можно использовать такую конструкцию:
- trigger_nestlevel(object_id(<Название триггера>))
- Можно настроить каскадный вызов триггеров – при выполнении каких-то операций над таблицами базы внутри триггера будут вызываться триггеры, соответствующие этим операциям.
- Используя INSTEAD OF триггеры можно изменять информацию в таблицах через VIEW

Обновление данных триггером через view

```
CREATE   VIEW dbo.FileTable
AS
SELECT
        FT_ID = 'up_' + CAST ( U.Up_ID AS varchar ),
        Native_ID = U.Up_ID,
FROM Uploads U
UNION
SELECT
        FT_ID = 'lib_' + CAST ( Lib_ID AS varchar ),
        Native_ID = L.Lib_ID,
FROM Library L
```

Обновление данных триггером через view

```
CREATE    TRIGGER tri_FileTableUpdate ON FileTable
INSTEAD OF DELETE
AS
BEGIN
        IF ( SELECT COUNT(*) FROM DELETED ) > 0
        BEGIN
                DELETE U FROM Uploads U, DELETED D
                WHERE D.Ft_ID = 'up_' + CAST( U.Up_ID AS varchar )

                DELETE L FROM Library L, DELETED D
                WHERE 'lib_' + CAST( L.Lib_ID AS varchar ) = D.Ft_ID
        END
END
```

Пример:

```
CREATE TABLE NestedTest (
        NT_ID int NOT NULL ,
        NT_Name varchar (50) NOT NULL ,
        NT_Parent int NULL ,
        CONSTRAINT [PK_NestedTest] PRIMARY KEY  CLUSTERED
        (
                [NT_ID]
        )  ON [PRIMARY]
) ON [PRIMARY]
GO
```

```
INSERT INTO NestedTest(NT_ID,NT_Name)
VALUES (1,'Obj1')

DECLARE @i int
SET @i = 1

WHILE ( @i <= 50 )
BEGIN
 INSERT INTO NestedTest(NT_ID,NT_Name,NT_Parent)
 VALUES (@i+1, 'Obj' + CAST(@i AS varchar), @i )

 SET @i = @i + 1
END
GO
```

```
ALTER TRIGGER tri_NestedTestDelete ON NestedTest
FOR DELETE
AS
IF @@ROWCOUNT = 0
RETURN
PRINT trigger_nestlevel( object_id( 'tri_NestedTestDelete' ) )
IF ( trigger_nestlevel( object_id( 'tri_NestedTestDelete' ) ) > 31 )
RETURN
DELETE T FROM NestedTest T, DELETED D
WHERE D.NT_ID = T.NT_Parent
GO

DELETE FROM NestedTest WHERE NT_ID = 1

SELECT * FROM NestedTest
```

User Defined Functions (UDFs)

- Могут возвращать скалярный результат так, как это делает, например, функция getdate()
- Результатом выполнения функции может быть таблица, в этом случае мы можем пользоваться этой таблицей как VIEW, при это мы имеем возможность передавать параметры (view лишены этого)
- При разработке SQL Server 2005 специалисты Microsoft вкдючили поддержку для независимых от языка UDF (например, UDF, записанное в VBScript). В 2000 версии, в силу существующих программных ограничений, UDFs пока можно создавать лишь на языке SQL (T-SQL).

Синтаксис

CREATE FUNCTION [*owner_name.*] *function_name*
([{ *@parameter_name* [AS] *scalar_parameter_data_type* [= *default*] } [*,...n*]])
RETURNS *scalar_return_data_type*
[WITH < function_option> [[,] ...*n*]]
[AS]
BEGIN
function_body
RETURN *scalar_expression*
END

CREATE FUNCTION [*owner_name.*] *function_name*
([{ *@parameter_name* [AS] *scalar_parameter_data_type* [= *default*] } [*,...n*]])
RETURNS TABLE
[WITH < function_option > [[,] ...*n*]]
[AS]
RETURN [( ] *select-stmt* [ )]

Пример 1

```
CREATE FUNCTION CubicVolume
-- Входные размеры в сантиметрах.
(@CubeLength decimal(4,1),   @CubeWidth decimal(4,1),
@CubeHeight decimal(4,1) )
RETURNS decimal(12,3) - Cubic centimeters.
AS
BEGIN
RETURN ( @CubeLength * @CubeWidth * @CubeHeight )
END


SELECT CubicVolume(1,2,3)
```

В качестве параметров естественно могут выступать названия полей, а не только константы или переменные

Хранимые процедуры

- *"трехзвенная архитектура"* - имеется хранилище данных (1-е звено), имеется сервер приложений (2-е звено), который выбирает из этого хранилища данные и определенным образом эти данные обрабатывает и после обработки конечный результат уже посылает на терминал клиента (3-е звено).

- *"клиент-сервер"* - имеется хранилище данных (сервер) и клиент, который с этого сервера выбирает данные с помощью определенного языка запросов (SQL) (Устаревший взгляд, возвращающий нас во времена СУБД типа FoxPRO со встроенной поддержкой sql-запросов).

- Более современное описание технологии "клиент-сервер" выглядит так:
  Имеется хранилище данных (1-е звено) и клиент (3-е звено), который с этого сервера выбирает данные с помощью определенного языка запросов (SQL), но помимо этого есть сервер приложений (2-е звено), уже встроенный в базу данных, с помощью которого можно обрабатывать данные любыми известными реляционной алгебре способами и уже после этого передавать конечный результат на клиента.

- Хранимые процедуры как раз и выполняют роль сервера приложений. С их помощью с данными можно делать все. Для этого достаточно вызвать заранее написанный код в виде хранимой процедуры со стороны клиента.

Важная область применения хранимых процедур - ограничение доступа к базе данных. Например, можно запретить для пользователей доступ на добавление записей в таблицу, и выполнять добавление записей с помощью специальной хранимой процедуры, доступ к которой открыт для всех.

Синтаксис

CREATE PROC [EDURE] ***procedure_name*** **[;** ***number*** **]**
[{ *@parameter data_type* }
[VARYING] [= ***default*** **] [OUTPUT]**
] [,...*n*]

[WITH
{ RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION }]

[FOR REPLICATION]

AS ***sql_statement*** **[...*n*]**

;number *– необязательный параметр, используется для создания группы процедур с одинаковым именем, отличающимися только указанными номерами. В дальнейшем вся эта группа процедур может быть удалена одной командой* ***drop procedure procgrname.*** *Например процедуры* ***orderproc;1*** *и* ***orderproc;2*** *удаляются командой* ***drop procedure orderproc***

- ***@parameter*** ***-*** *параметр, подаваемый на вход процедуры (их может быть несколько) Описывается как переменная.*

- ***data_type*** ***-*** *тип параметра*

- ***VARYING*** *используется для спецификации изменяемого result set'а, который может быть помещен в выходной параметр процедуры (varying применяется только с параметрами типа CURSOR)*

- ***default*** *– задает значение для параметра по умолчанию, если значение по умолчанию задано таким образом передавать этот параметр при вызове процедуры необязательно*

- ***OUTPUT*** *– указывает, что параметр используется для возвращения значений.*

- ***RECOMPILE*** *– план запросов, вызываемых в процедуре не кэшируется.*

- По умолчанию все параметры nullable
- Число параметров не должно превышать 2100
- Максимальный размер хранимой процедуры – 128Мб
- Можно создать временную хранимую процедуру (имя начинается с # для локальной и с ## для глобальной)

- Хранимые процедуры могут вызывать друг друга (nesting), для определения уровня вложенности вызовов имеется функция @@NESTLEVEL. Если уровень вложенности превысил максимальный будет сделан rollback всем вызовам

- Хранимая процедура вызывается конструкцией exec[ute]

Внутри хранимой процедуры имена объектов, использующиеся в определенных конструкциях должны обязательно начинаться с имени object owner'a, если процедура вызывается несколькими пользователями. Список этих выражений:

- **ALTER TABLE**
- **CREATE INDEX**
- **CREATE TABLE**
- **All DBCC statements (консольные команды базы данных)**
- **DROP TABLE**
- **DROP INDEX**
- **TRUNCATE TABLE (удаляет все содержимое таблицы)**
- **UPDATE STATISTICS (statistic – гистограмма частот, которая может быть создана для определенных колонок таблицы или представления. Используется оптимизатором запросов)**

Примеры

Создадим таблицы, которые будем в дальнейшем использовать в наших процедурах и заполним их:

```
CREATE TABLE Master1
(
Master1ID int IDENTITY (1,1) NOT NULL,
Detail1ID int NULL,
Detail2ID int NULL,

Name varchar(200),
  CONSTRAINT PK_Master1 PRIMARY KEY CLUSTERED
  (Master1ID)
)
GO
```

```
CREATE TABLE Detail1
(
Detail1ID int IDENTITY (1,1) NOT NULL,
Name varchar(200),
  CONSTRAINT PK_Detail1 PRIMARY KEY CLUSTERED
  (Detail1ID)
)
GO


CREATE TABLE Detail2
(
Detail2ID int IDENTITY (1,1) NOT NULL,
Name varchar(200),
  CONSTRAINT PK_Detail2 PRIMARY KEY CLUSTERED
  (Detail2ID)
)
GO
```

```
insert Detail1 (Name) values('Рабочий')
insert Detail1 (Name) values('Инженер')
insert Detail1 (Name) values('Дворник')
insert Detail1 (Name) values('Программист')

GO
insert Detail2 (Name) values('высшее')
insert Detail2 (Name) values('среднее')
insert Detail2 (Name) values('неполное высшее')
insert Detail2 (Name) values('кандидат наук')
GO

insert Master1 (Detail1ID,Detail2ID,Name) values(1,1,'Иванов')
insert Master1 (Detail1ID,Detail2ID,Name) values(2,2,'Петров')
insert Master1 (Detail1ID,Detail2ID,Name) values(1,2,'Сидоров')
insert Master1 (Detail1ID,Detail2ID,Name) values(4,3,'Лаврененко')
insert Master1 (Detail1ID,Detail2ID,Name) values(null,1,'Кошкин')
insert Master1 (Detail1ID,Detail2ID,Name) values(3,null,'Самойлов')
GO
```

Процедура, возвращающая набор данных:

```
CREATE PROCEDURE msp_List1 @ID int
AS
select a.Master1ID, a.Name, b.Name, c.Name
from Master1 a
left join Detail1 b on b.Detail1ID=a.Detail1ID
left join Detail2 c on c.Detail2ID=a.Detail2ID
where a.Master1ID=@ID
GO
```

Выполнение:

```
exec msp_List1 4
GO
```

Результат:

```
Master1ID   Name                 Name                 Name
----------- -------------------- -------------------- --------------------
4           Лаврененко           Программист          неполное высшее


(1 row(s) affected)
```

Для использования набора данных, возвращаемого хранимой процедурой для обработки на сервере (в других процедурах и т.п.) можно вставить его во временную таблицу и уже с ней работать:

```
CREATE TABLE #tmp (Master1ID int, Name1 varchar(200), Name2 varchar(200), Name3 varchar(200))

INSERT #tmp exec msp_List1 1

select * from #tmp
GO
```

Результат:

```
Master1ID   Name1                Name2                Name3
----------- -------------------- -------------------- --------------------
1           Иванов               Рабочий              высшее

(1 row(s) affected)
```

Процедура, возвращающая данные:

```
CREATE PROCEDURE msp_List2 @ID int, @Name varchar(200) OUTPUT
AS
select @Name=Name
from Master1
where Master1ID=@ID
GO
```

Выполнение:

```
declare @s varchar(200)
exec msp_List2 5, @s OUTPUT
print @s
GO
```

Результат:

Кошкин

Ничего не возвращающая процедура:

```
CREATE    PROCEDURE sp_VendorsImport @file varchar(200)
AS
--begin of procedure
BEGIN TRAN
CREATE TABLE #Import (
      Company varchar (50)   NULL ,
      [Name] varchar (50)   NULL ,
      Email varchar (30)   NULL,
)
if @@error <> 0
begin
      raiserror('Could not create import table',16,1)
      goto error_end
end
```

```
DECLARE @bulk_insert nvarchar(2000)
SET @bulk_insert='BULK INSERT #Import FROM ''' + @file + ''' '+
    'WITH ('+
    'CODEPAGE=''ACP'','+
    'FIELDTERMINATOR=''\t'','+
    'KEEPNULLS,'+
    'ROWTERMINATOR=''\n'','+
    'FIRSTROW=2'+
    ''+
    ')'
EXEC sp_executesql @stmt=@bulk_insert
```

```
if @@error <> 0 or @@rowcount = 0
begin
	raiserror('Could not parse vendors file',16,2)
	goto error_end
end
COMMIT TRAN
RETURN
error_end:
ROLLBACK TRAN
--end of procedure
GO
```

Пример 2

```
CREATE  FUNCTION ReturnedTable( @ID int )
RETURNS TABLE
AS
RETURN(
SELECT @ID as 'ID', 'Test' as 'Name'
)

SELECT * FROM ReturnedTable(1)
```