

Microsoft SQL Server

Программирование
и администрирование СУБД

Урок №2

Триггеры, хранимые процедуры. Пользовательские функции

Содержание

1. Триггеры.	3
2. Понятие Transact-SQL и его расширения	20
3. Транзакции в MS SQL Server.....	41
4. Хранимые процедуры.	50
5. Пользовательские функции	62

1. Триггеры

Триггер – это специализированная процедура, которая автоматически вызывается SQL Server при возникновении событий в базе данных. В SQL Server с версии SQL Server 2005 поддерживаются два **типа триггеров**:

1. **DML-триггеры** – выполняются при возникновении DML (Data Manipulation Language – язык манипулирования данными) событий: добавлении (INSERT), удалении (DELETE) или обновлении (UPDATE) записей таблиц или представлений. Такие триггеры всегда привязаны к определенной таблице или представлению и могут перехватывать данные только ее / его.
2. **DDL-триггеры** – выполняются при возникновении DDL (Data Definition Language – Язык Определения Данных) событий: создание (CREATE), изменение (ALTER) или удаление (DROP) объектов. Отдельную подгруппу образуют триггеры входа, которые срабатывают на событие LOGON, которое возникает при установлении сеанса пользователя. DDL-триггеры используются для администрирования базы данных, например, для аудита и управления доступом к объекту.

Триггеры могут быть созданы как с помощью инструкций Transact-SQL, так и с помощью методов сборок, созданных в среде CLR платформы .NET Framework и переданы экземпляру SQL Server. Все триггеры не имеют параметров и не выполняются явно. При возникновении

события, к которому привязан триггер, SQL Server автоматически его запускает.

Итак, начнем наше знакомство, а начнем мы с **DML-триггеров**. Синтаксис создания такого триггера имеет следующий вид:

```
CREATE TRIGGER [схема.] имя_триггера
ON { таблица | представление }      -- для кого создается
триггер
[ WITH ENCRYPTION                     -- зашифровать
исходный код триггера
[, EXECUTE AS условие ]              -- контекст
выполнения
]
{ FOR | AFTER                         -- после изменения данных
| INSTEAD OF }                       -- вместо SQL команды, для
которой они объявлены
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] } -- на
какое действие с данными
[ WITH APPEND ]                      -- добавить триггер существующего
типа
[ NOT FOR REPLICATION ] -- триггер не выполняется, если
в ходе репликации будет изменена
- таблица, на которую он указывает
AS
{ тело_триггера | EXTERNAL NAME сборка.класс.метод }
```

После указания имени триггера в инструкции ON необходимо указать название таблицы или представления, для которого создается триггер.

С помощью инструкции **WITH** можно включить шифрование кода триггера (ENCRYPTION) или указать контекст исполнения (EXECUTE AS), в котором будет работать триггер. Параметр EXECUTE AS в основном используется для проверки привилегий (прав доступа) на объекты базы данных, на которые ссылается триггер.

Как видно из синтаксиса, DML-триггеры можно запускать в **двух режимах**: AFTER и INSTEAD OF. Триггера BEFORE, который присутствует во многих СУБД, в MS SQL Server не существует.

Триггер INSTEAD OF может порождать событие (команду), для которой он объявлен. Причем это событие будет выполняться так, будто триггера INSTEAD OF не существовало. Например, если вы хотите проверить определенное условие для выполнения команды INSERT, вы можете объявить триггер INSTEAD OF INSERT. Триггер INSTEAD OF будет выполнять проверку, а затем выполнять команду INSERT для таблицы. Оператор INSERT будет выполняться привычным образом, порождая рекурсивных вызовов триггера INSTEAD OF.

Следует также помнить, что **по умолчанию все триггеры активные** и выполняются после проведенного действия, то есть имеют установленный параметр **AFTER**.

Замечания по построению триггеров:

1. Если при объявлении триггера, указать единственное ключевое слово FOR, то аргумент AFTER используется по умолчанию.
2. Нельзя создавать триггеры INSTEAD OF для модифицированных представлений.
3. Триггеры AFTER используются только для таблиц.
4. Параметр WITH ENCRYPTION не может быть указан для триггеров CLR.
5. Аргумент WITH APPEND установлен **только для совместимости** с предыдущими версиями (на уровне 65).

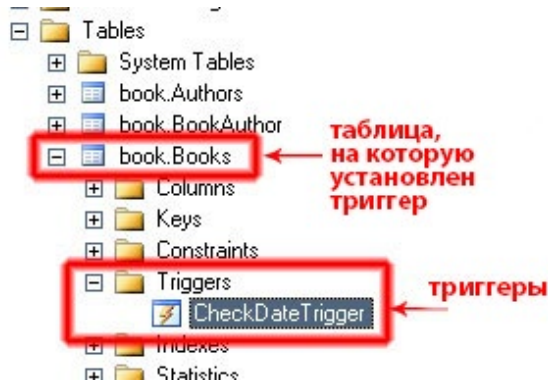
При этом он может использоваться только при указании параметра FOR без INSTEAD OF и не может указываться для триггеров CLR. **В связи с тем, что в следующей версии SQL Server аргумент WITH APPEND планируется полностью исключить, его рекомендуется избегать.**

Существует также ряд **правил**, которых следует придерживаться при **создании триггеров**:

- НЕЛЬЗЯ создавать триггеры для временных таблиц, но они могут обращаться к ним;
- НЕЛЬЗЯ создавать, изменять, удалять резервные копии или восстанавливать из резервной копии базы данных;
- триггеры могут использоваться для обеспечения целостности данных, но их не следует использовать вместо объявления целостности путем установления ограничения FOREIGN KEY;
- триггеры не могут возвращать результирующие наборы, поэтому при использовании оператора select в теле триггера следует быть очень внимательным. При этом во многих случаях, вместе с оператором select используется директива IF EXISTS;
- поддерживаются рекурсивные AFTER триггеры, при установке параметра базы данных RECURSIVE_TRIGGERS в значение ON;
- можно создавать вложенные триггеры (поддерживается до 32 уровней вложенности), которые фактически являются неявной рекурсией. Для их поддержки необходимо установить параметр NESTED TRIGGERS;

- в теле DML-триггера НЕЛЬЗЯ использовать операторы:
 - все операции CREATE / ALTER / DROP;
 - TRUNCATE TABLE;
 - RECONFIGURE;
 - LOAD DATABASE или TRANSACTION;
 - GRANT и REVOKE;
 - SELECT INTO;
 - UPDATE STATISTICS.

Триггеры также являются объектами БД и в SQL Management Studio они размещаются в папке "Triggers" таблицы, на которую установлен тот или иной триггер. К примеру:



В MS SQL Server в пределах DML-триггеров используют справочные таблицы **INSERTED** и **DELETED**, которые имеют ту же структуру, что и базовые таблицы триггера. Они размещаются в оперативной памяти, так как являются логическими таблицами. Работают они следующим образом:

- **когда в базовую таблицу добавляются новые данные** (новая запись), то эти же данные добавляются сначала

в базовую таблицу, а затем в таблицу inserted. Их наличие в таблице inserted избавляет от необходимости создавать специальные переменные для доступа к этой информации.

- **когда строка удаляется из таблицы**, то она записывается в таблицу deleted, а затем удаляется из базовой таблицы.
- **когда строка обновляется**, то старое значение записи записывается в таблицу deleted и удаляется из базовой таблицы, затем обновленная запись записывается в базовую таблицу, а дальше в таблицу inserted.

То есть мы можем использовать таблицу DELETED для получения значений записей, которые удаляются из таблицы, а таблицу INSERTED для получения новых записей перед их фактической вставкой. В других серверных СУБД такие задачи выполняют схожие механизмы, например, в InterBase / Firebird и Oracle – это контекстные переменные OLD и NEW.

Для лучшего понимания работы с триггерами напишем **несколько примеров**.

1. Классический триггер, который будет срабатывать при каждой вставке данных в таблицу Authors и возвращает сообщение о количестве измененных строк. В этом, нелегком на первый взгляд, деле, нам поможет глобальная переменная @@ **rowcount**, содержащая данные о количестве модифицированных строк, в результате работы триггера.

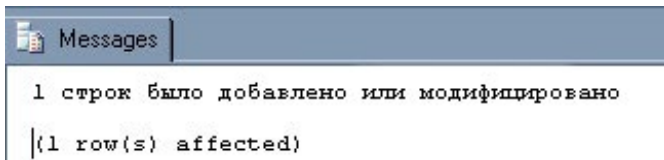

```
create trigger addAuthor
on book.Authors
for insert, update
```

```
as
    raiserror('%d строк было добавлено или модифицировано
', 0, 1, @@rowcount)
return
```

Добавляем нового автора в нашу БД:

```
insert into book.Authors(FirstName, LastName, id_country)
values ('Artur', 'Liliput', 1)
```

Результат:



1 строк было добавлено или модифицировано

2. Триггер, который при добавлении новых данных о книге, дата издательства которой больше месяца, будет выбрасывать сообщение об ошибке.

```

create trigger CheckDateTrigger
on book.Books
for insert
as
begin
    declare @InsDate smalldatetime
-- получаем дату издательства книги, которая добавляется
    select @InsDate = DateOfPublish
    from inserted
-- проверяем, сколько прошло дней со дня издания
    if (@InsDate <= getdate()-30)
    begin
        raiserror('Это старая книга и данные о ней
добавлены не будут ',0,1)
        rollback transaction
    end
end

```

```

else
    PRINT(' Данные добавлены успешно ')
End

```

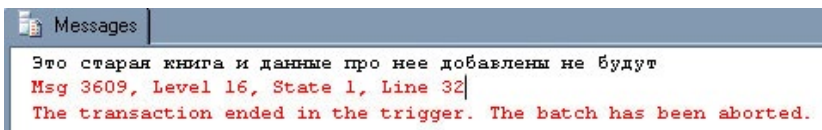
Добавляем новую книгу:

```

insert into book.Books(NameBook, id_theme, id_author,
Price, DrawingOfBook,
                        DateOfPublish, Pages)
values ('Администрирование MS SQL Server 2005', 21, 1,
125.0, 3500, '2007.09.01', 726)

```

Результат:



3. Триггер, запрещающий удаления книги, если она больше всего продается, то есть является лидером продаж.

```
create trigger CheckBookDelete
on book.Books
for delete
as
begin
    declare @NameBook varchar(25), @BestBook varchar(25)
-- Получаем название удаляемой книги
    select @NameBook = deleted.NameBook
    from deleted
    declare @Zvit table (nameB varchar(25), quantity int)
-- Получаем информацию о названиях книг и их количестве
    -- продаж (популярность)
    insert @Zvit
        select b.NameBook, count(s.id_book)
        from book.Books b, sale.Sales s
        where b.id_book = s.id_book
```

```

        group by b.NameBook
-- находим самую продаваемую книгу
    select @BestBook = z.nameB
    from @Zvit z
    where z.quantity = (select max(quantity)
                        from @Zvit)
-- проверяем, совпали ли названия
    if (@BestBook = @NameBook)
    begin
        raiserror("Вы не можете удалить эту книгу ",0,1)
        rollback transaction
    end
    else
    begin
        print ("Книга удалена успешно ")
    end
end
end

```

4. Триггер, который при удалении книги тематики "Computer Science" выдает сообщение об ошибке.

```

create trigger NotDeleteComputerScience
on Books
instead of delete
as
begin
    declare @ThemeId int
-- get the identifier of 'Computer Science' theme
    select @ThemeId = id
    from Themes
    where
        NameTheme = 'Computer Science'
-- check whether the identifier of removing book matches the
    @ThemeId
    if exists (select * from deleted where id_theme = @ThemeId)
        raiserror ('This book cannot be deleted!',0,1)
end;

```

Как уже было сказано выше, для DDL-операций, таких как CREATE, ALTER, DROP, GRANT, DENY, REVOKE и UPDATE STATISTICS, используются **DDL-триггеры**. Кроме контроля и мониторинга данных операций, DDL-триггеры позволяют ограничивать их выполнение, даже если пользователь имеет соответствующие права. Например, можно запретить пользователям определенных групп изменять и удалять таблицы. Для этого достаточно создать DDL-триггер для событий DROP TABLE и ALTER TABLE, который будет делать откат этих операций и выдавать соответствующее сообщение об ошибке.

Синтаксис создания DDL-триггера имеет следующий вид:

```
CREATE TRIGGER имя_триггера
ON { ALL SERVER | DATABASE }           -- область действия
триггера
[ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER } { имя_события | группа_событий } [ ,...n
]
AS
{ тело_триггера | EXTERNAL NAME сборка.класс.метод }
```

Как видно из синтаксиса, после инструкции ON необходимо указать **область действия триггера**:

- **ALL SERVER** – текущий сервер. Триггер будет срабатывать при возникновении определенных событий в любом месте в рамках текущего сервера. Например, CREATE_DATABASE, ALTER_LOGIN, ALTER_INSTANCE и тому подобное.
- **DATABASE** – текущая база данных. Триггер будет срабатывать при возникновении определенных событий на уровне текущей базы данных или низших уровнях.

Например, CREATE_TABLE, DROP_DEFAULT, ALTER_USER и тому подобное.

Для удобства администрирования можно использовать группы событий, например, группа событий DDL_TABLE_EVENTS включает в себя события, которые касаются таблицы: CREATE_TABLE, ALTER_TABLE и DROP_TABLE. Группы событий имеют иерархическую структуру. Например, группа событий DDL_TABLE_VIEW_EVENTS охватывает все инструкции T-SQL в группах DDL_TABLE_EVENTS, DDL_VIEW_EVENTS, DDL_INDEX_EVENTS и DDL_STATISTICS_EVENTS.

Детальный список названий событий приведен в разделе "DDL-события" электронной документации по SQL Server 2008, а групп событий в разделе "Группы DDL-событий".

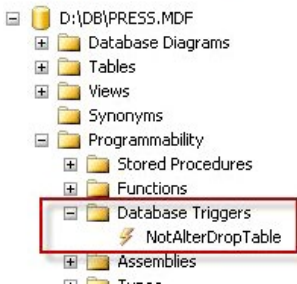
Проиллюстрируем код вышеприведенного примера использования DDL-триггера, то есть на запрет изменения и удаления таблиц:

```
create trigger NotAlterDropTable
on DATABASE
for DROP_TABLE, ALTER_TABLE
as
begin
    print 'Модификация и удаление таблиц запрещены.
    Обратитесь к администратору.'
    rollback
end
```

Стоит отметить, что DDL-триггеры не вызываются на события, которые влияют на временные таблицы и хранимые процедуры. DDL-триггеры также не ограничены областью схемы и после создания триггеры уровня базы

данных находятся в папке "**Programmability / Database Triggers**", а триггеры уровня сервера – в папке "Triggers" папки "**Server Objects**" (объекты сервера).

DDL-триггер уровня базы данных



DDL-триггер уровня сервера



Отдельную подгруппу DDL-триггеров составляют **триггеры входа (logon trigger)**:

```
CREATE TRIGGER имя_триггера
ON ALL SERVER
[ WITH ENCRYPTION [, EXECUTE AS условие ] ]
{ FOR | AFTER } LOGON
AS
{ тело_триггера | EXTERNAL NAME сборка.класс.метод }
```

Такие триггеры выполняются в ответ на событие **LOGON**, которое вызывается при установке пользовательского сеанса с экземпляром сервера. Триггеры входа срабатывают после завершения этапа аутентификации при входе, но перед тем, как сеанс пользователя реально устанавливается. Итак, все сообщения об ошибке функции raiserror или инструкции PRINT, которые вызываются в триггере, перенаправляются в журнал ошибок SQL Server. Если же

пользователь ввел неверный логин или пароль, то триггер входа не срабатывает.

Отметим, что в триггерах входа не поддерживаются распределенные транзакции. Триггеры входа могут создаваться из любой базы данных, но принадлежат они базе данных **master**.

В следующем примере запретим пользователю с логином 'vasja_pupkin' подключение к SQL Server.

```
use master;
go
create trigger TriggerConnection
on ALL SERVER
with execute as 'vasja_pupkin'
for logon
as
begin
    if ORIGINAL_LOGIN()= 'vasja_pupkin'
    begin
        print 'Такой логин запрещен на сервере.
Обратитесь к администратору.'
        rollback
    end
end
end
```

Триггеры можно изменять, временно отключить или удалить.

Модифицировать триггер можно с помощью инструкции **ALTER TRIGGER**:

```
-- для DML-триггера
ALTER TRIGGER [схема.] имя_триггера
    ON { таблица | представление }
    [ WITH ENCRYPTION [, EXECUTE AS условие ] ]
    { FOR | AFTER | INSTEAD OF }
```



```

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ NOT FOR REPLICATION ]
AS
    { тело_триггера | EXTERNAL NAME сборка.класс.метод }
-- для DDL-триггера
ALTER TRIGGER имя_триггера
    ON { ALL SERVER | DATABASE }
        [ WITH ENCRYPTION [ , EXECUTE AS условие ] ]
    { FOR | AFTER } { имя_события | группа_событий } [ , ...n ]
AS
    { тело_триггера | EXTERNAL NAME сборка.класс.метод }
-- для триггера входа
ALTER TRIGGER имя_триггера
    ON ALL SERVER
        [ WITH ENCRYPTION [ , EXECUTE AS условие ] ]
    { FOR | AFTER } LOGON
AS
    { тело_триггера | EXTERNAL NAME сборка.класс.метод }

```

Включить или выключить триггер можно с помощью инструкции ALTER TABLE или с помощью следующих инструкций:

```

-- включить триггер
ENABLE TRIGGER { [ схема. ] имя_триггера [ , ...n ] | ALL }
ON { таблица | представление | DATABASE | ALL SERVER }
-- выключить триггер
DISABLE TRIGGER { [ схема. ] имя_триггера [ , ...n ] | ALL }
ON { таблица | представление | DATABASE | ALL SERVER }

```

Опция **ALL** указывает на то, что все триггеры в области действия значение параметра ON будут включены или выключены.

Опции **DATABASE** и **ALL SERVER** указывают на то, что инструкция касается DDL-триггера уровня базы данных или сервера (включая триггер входа) соответственно.

Удаление триггера осуществляется оператором **DROP TRIGGER**:

```
-- для DML-триггера
DROP TRIGGER [схема.] имя_триггера [ ,...n ] [ ; ]
-- для DDL-триггера
DROP TRIGGER имя_триггера [ ,...n ]
ON { DATABASE | ALL SERVER }
-- для триггера входа
DROP TRIGGER имя_триггера [ ,...n ]
ON ALL SERVER
```

Имена всех триггеров хранятся в системной таблице **sysobjects** и системном представлении **sys.objects**. Метаданные DDL- и DML-триггеров уровня базы данных можно просмотреть с помощью нового представления **sys.triggers**. Если поле **parent_class_desc** данного представления имеет значение "DATABASE", тогда это DDL-триггер и его областью действия является база данных. Тело триггера можно получить по представлению **sys.sql_modules** (связав его с **sys.triggers** по полю **object_id**), а код создания – из системной таблицы **syscomments**. Метаданные CLR-триггеров доступны по представлению **sys.assembly_modules**, которое также следует связать с **sys.triggers** по полю.

Информация про DDL-триггеры уровня сервера, включая триггеры входа, сохраняются в системном представлении **sys.server_triggers**. Тело триггера уровня сервера можно получить по представлению **sys.server_sql_modules**, а метаданные CLR-триггера серверного уровня – по представлению **sys.server_assembly_modules**.

В завершение отметим, что SQL Server предоставляет информацию о событиях, которые он отслеживает, в виде XML. Они доступны через новую встроенную функцию **EVENTDATA** (), которая возвращает XML-данные. Эта возможность позволяет применять DDL-триггеры для аудита DDL-операций в базе данных.

Для этого, например, можно создать таблицу аудита с полем, которое содержит XML-данные. Затем создаем DDL-триггер с параметром EXECUTE AS SELF для DDL-событий или групп событий, которые вас интересуют. Тело такого DDL-триггера может просто выполнять вставку (INSERT) XML-данных, которые возвращаются EVENTDATA () в таблицу аудита.

2. Понятие Transact-SQL и его расширения

Transact-SQL (T-SQL) – это расширение языка запросов ANSI SQL, который был разработан компаниями Microsoft (для Microsoft SQL Server) и Sybase (для Sybase ASE). Одно из основных назначений языка T-SQL – позволить разработчикам баз данных с легкостью создавать запросы, возвращая при этом данные множеством способов.

Стандарт ANSI SQL был расширен набором элементов и операторов, которые часто используются в триггерах, транзакциях, хранимых процедурах и функциях. Рассмотрим **ряд расширений языка T-SQL**:

5. Операторы **BEGIN..END**, которые ограничивают несколько операторов определенного блока.
6. **Переменные**, которые служат для сохранения произвольных данных. Для того, чтобы создать переменную, нужно ее задекларировать:

```
declare {@имя_локальной_переменной | @@ имя_глобальной_
переменной } [ AS ] тип [ = value ]
```

Как видно из синтаксиса при создании переменной ее можно проинициализировать определенным значением. Присвоить же значение уже созданной переменной можно двумя способами: с помощью оператора **select** и с помощью оператора **set**:

```

-- Инициализация переменной
declare @find varchar(10) = 'Boo%';

-- Присвоение значения с помощью оператора select
select переменная1 = значение1 [, переменная N = значениеN
]
-- например
declare @var int, @a char(5)
select @var = 5, @a = 'Hello'
-- Присвоение значения с помощью оператора set
set переменная = значение
-- например
declare @var int, @a char(5)
set @var = 5
set @a = 'Hello'

```

Самый простой способ вывода значения с переменной – это воспользоваться помощью уже знакомого оператора `select`, который обладает способностью выводить литералы и значение переданных полей.

```

select 'Значение переменной @var = ' + convert(char(10), @var)
select 'Строка: ' + @a + ' ' + convert(char(10), @var)

```

Результат:

Results		Messages
	(No column name)	
1	Значение переменной @var = 5	
	(No column name)	
1	Строка: Hello 5	

С помощью T-SQL можно создавать также переменные табличного типа. Например, создадим табличную

переменную @MyTable и заполним ее значениями из таблицы Books.

```
-- создаем переменную табличного типа
declare @MyTable table( Id int NOT NULL,
                        number int);

-- заполняем переменную данными
insert @MyTable
select top (5) ID_BOOK, Pages
from book.Books

-- выводим на экран данные с переменной типа таблицы
select Id, number
from @MyTable;
```

Результат:

	Id	number
1	6	640
2	7	440
3	8	720
4	9	200
5	10	288

7. Оператор **PRINT** позволяет вывести строку в формате ASCII, переменную символьного типа или выражение, результатом которого также есть строка. Синтаксис оператора имеет следующий вид:

```
PRINT { 'строка_ASCII' | @локальная_строчная_переменная |
        выражение_которое_возвращает_строку }
```

Например:

```
PRINT 'Hello World'
DECLARE @msg nvarchar(50);
SET @msg = N'Сегодня ' + CAST(GETDATE() AS nvarchar(30)) + N'.';
PRINT @msg;
```

Результат:



Следует отметить, что в данном операторе нельзя использовать оператор конкатенации, а поэтому он не подходит для вывода форматированной строки. Чтобы выйти из такой ситуации, следует сохранить отформатированную строку в переменной, а затем вывести ее с помощью оператора PRINT.

8. Функция **Raiserror** выводит сообщение об ошибке. Это сообщение является строкой (не более 2047 символов), а потому его можно отформатировать произвольным образом. Синтаксис функции Raiserror очень похож на синтаксис функции printf библиотеки языка C и выглядит так:

```
Raiserror ( { 'строка_со_спецификаторами' |
идентификатор_ошибки | @переменная },
           степень_важности_ошибки,
           состояние_ошибки_на_момент_вызова,
           подставляемые_переменные) ;
```

В качестве первого параметра **может использоваться**:

- строка, содержащая сообщение об ошибке в отформатированном виде;
- **переменная**, которая должна иметь тип char или varchar, и содержать отформатированную строку об ошибке;
- **номер (идентификатор)** сообщение об ошибке, определенный пользователем и сохраненный в системном

представлении `sys.messages` с помощью процедуры `sp_addmessage`. Номер пользовательского сообщения должен быть больше 50 000.

Строка с сообщением об ошибке для форматирования может содержать спецификации преобразования следующего формата:

```
% [[flag] [width] [. precision] [{h | l}]] type
```

flag – это код, который позволяет определить выравнивание или промежуток подставляемого значения (-, +, 0, #, ").

width – минимальная ширина поля, в которое помещается значение аргумента. Символ (*) означает, что ширина определяется одним из аргументов в функции `raiserror`.

precision – точность (максимальное количество символов строки). Символ (*) означает, что точность определяется одним из аргументов в функции `raiserror`.

type – спецификаторы, указывающие на то, какого типа данные можно использовать в строке:

- **%d** или **%i** – целое число с знаком (signed int)
- **%s** – строка символов (string)
- **%u** – беззнаковое целое (unsigned int)
- **%o** – беззнаковое число в восьмеричной системе счисления (unsigned octal)
- **%x** или **%X** – беззнаковое число в 16-значной системе счисления (unsigned hexadecimal)
- Действительные числа не поддерживаются.

Второй параметр – это **степень важности** ошибки, который указывается в пределах от 0 до 25.

- от 0 до 18 – могут указываться пользователями.

- от 19 до 25 – критические ошибки, которые могут указывать только члены серверной роли sysadmin и пользователи с правами ALTER TRACE.

Считается, что ошибки от 20 до 25 невозможно устранить. В случае таких ошибок соединение клиента с сервером разрывается и регистрируется сообщение об ошибке в журналах приложения и ошибок.

Состояние ошибки на момент вызова должно быть целым числом в диапазоне от 0 до 255. По умолчанию это значение равно 1. Если одна и та же пользовательская ошибка возникает в нескольких местах, то с помощью этого уникального значения для каждого места расположения можно определить, где была сгенерирована ошибка.

Подставляемые переменные – это переменные, которые должны быть подставлены на месте спецификаторов.

Например:

```
RAISERROR (N'This is message %s %d.', 10, 1, /*аргументы:
*/'number', 5);

go

RAISERROR (N'<<%.*s>>', 10, 1,
          7, -- Первый аргумент используется для ширины
поля
          3 -- второй аргумент используется для
точности (количество символов)
          N'abcde'); -- Третий аргумент - сама строка

go

RAISERROR ('<<%7.3s>>', 10, 1, 'abcde');
```

Результат:



И еще маленький пример кода, в котором мы попытаемся воспользоваться собственным номером ошибки. Сообщение, которому мы хотим присвоить номер, следует добавить в системное представление `sys.messages` с помощью системной хранимой процедуры `sp_addmessage`:

```
sp_addmessage @msgnum = 50005, @severity = 10, @msgtext =
'""%7.3s""';
go
RAISERROR (50005, -- идентификатор сообщения
          10, 1, 'Hello');
go
sp_dropmessage @msgnum = 50005;
```

Результат:



9. Условный оператор **if.else**, который используется для проверки условия и имеет следующий синтаксис:

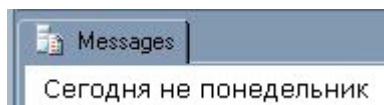
```
if [(|булевоe_выражение|)]
    действие
[else |булевоe_выражение|]
    действие
]
```

Например, нам необходимо определить текущий день недели. Если день недели "Понедельник", тогда выводим на экран значение "Текущий день недели", иначе указать на неверный результат.

```
if (datetime(dw, GetDate())) = 'Monday'
begin
    PRINT 'Сегодня понедельник'
end
```

```
else
  PRINT 'Сегодня не понедельник'
```

Результат:



Для получения названия недели мы воспользовались функцией **DATENAME**:

```
DATENAME ( datepart, date )
```

Параметр **datepart** указывает на то, что именно вы хотите получить из даты:

Datepart	Абревиатура	Datepart	Абревиатура
year	yy, yyyy	weekday	dw
quarter	qq, q	hour	hh
month	mm, m	minute	mi, n
dayofyear	dy, y	second	ss, s
day	dd, d	millisecond	ms
week	wk, ww		

Булевое выражение возле **if** может содержать оператор **SELECT**, который нужно ОБЯЗАТЕЛЬНО заключить в скобки. Если **SELECT** возвращает одно значение, тогда его можно использовать для сравнения с другим значением и построением булевого условного выражения. Например, определим цену книг и, если полученная цена будет больше 50 грн., тогда выведем соответствующее сообщение:

```
if (select avg(price)
    from book.Books) > 50
begin
```

```
PRINT ' Существуют книги, средняя цена которых больше
50 грн.'
End
```

Результат:



Если SELECT возвращает несколько значений, то в условном выражении if используется ключевое слово **EXISTS**, которое возвращает true, если SELECT вернул хотя бы одну запись, иначе – false. Синтаксис оператора if вместе с exists будет иметь следующий вид:

```
if exists (оператор SELECT)
    действие
[else [булевоe_выражение]
    действие
]
```

Например, выведем всю информацию про каждую книгу, дата издательства которой находится в промежутке от 01.01.2006 до сегодняшнего дня:

```
if exists (select *
           from book.Books
           where DateOfPublish between '2006.01.01' and
current_timestamp)
begin
    PRINT 'Информация про книги'
    select *
    from book.Books
    where DateOfPublish between '01.01.2006' and current_
timestamp
    return
end
```

Результат:

Results		Messages					
	id	NameBook	Price	Pages	Quantity	DateOfPublish	Id_author
1	5	CLR via C#	43	380	10	2012-11-25 00:00:00.000	2
2	6	Windows Runtime via C#	23	370	20	2013-11-25 00:00:00.000	2
3	11	Java: A Beginner Guide	15	376	10	2014-05-06 00:00:00.000	3
4	12	Java: The Complete Reference	37	420	15	2014-04-01 00:00:00.000	3
5	14	Swing: A Beginner Guide	34	380	15	2006-09-08 00:00:00.000	3
6	15	Programming ASP.NET Core (Developer Reference)	45	410	15	2016-10-19 00:00:00.000	4
7	16	Programming Microsoft ASP.NET MVC (3rd Edition)	36	470	15	2014-02-25 00:00:00.000	4
8	18	Creating a Web Site: The Missing Manual	46	608	10	2006-01-06 00:00:00.000	5
9	19	Pro WPF in C# 2010: Windows Presentation Foundation in .Net 4	51	1181	10	2010-03-19 00:00:00.000	5
10	20	Pro Wpf 4.5 in C#: Windows Presentation Foundation in .Net 4.5	48	1078	10	2012-12-27 00:00:00.000	5

10. Оператор ветвления CASE, который позволит вернуть разные значения в зависимости от определенного контролирующего значения или условия. Операторы, которые включают в себя структуру CASE, могут использовать одну из двух синтаксических форм, в зависимости от того, будет ли меняться оцениваемое выражение:

- 1) Простая**, в которой результирующее значение возвращается только в случае, если выражение после **WHEN** логично равно указанному значению. Вы можете использовать любое количество инструкций **WHEN**. Инструкция **ELSE** необязательна и выполняется только, если все инструкции **WHEN** оцениваются как **FALSE**.

```

case условие_выражение
when выражение_константа1 then результирующее_значение1
when выражение_константа2 then результирующее_значение 2
[, ... n]
[else результирующее_значение N]
End

```

- 2) **С поиском.** В этой форме CASE можно указать условное выражение при каждой инструкции WHEN.

```
case
when условие_выражение1 then результирующее_значение1
when условие_выражение2 then результирующее_значение2
[, ... n]
[else результирующее_значениеN]
End
```

Примечание! В SQL Server CASE является функцией, а не командой. В связи с этим CASE может использоваться только как часть оператора SELECT или UPDATE, в отличие от оператора IF, который работает самостоятельно.

А теперь напомним **несколько примеров:**

- а) Напишем запрос, который будет выводить на экран название книги и ее тематику в расширенном виде:

```
select 'Book title' = b.NameBook,
       'Topic' = case t.NameTheme
                  when 'Computer Science' then
                    'Everything about programming'
                  when 'Web Technologies' then
                    'For Web developers'
                  else 'Leisure reading'
                end
from books b, Themes t
where b.id_theme = t.id;
```

Результат:

	Book title	Topic
11	Java: A Beginner Guide	Everything about programming
12	Java: The Complete Reference	Everything about programming
13	C++: The Complete Reference, 4th Edition	Everything about programming
14	Swing: A Beginner Guide	Everything about programming
15	Programming ASP.NET Core (Developer Reference)	For Web developers
16	Programming Microsoft ASP.NET MVC (3rd Edition)	For Web developers
17	ASP.Net: The Complete Reference	Everything about programming
18	Creating a Web Site: The Missing Manual	For Web developers
19	Pro WPF in C# 2010: Windows Presentation Foundation in .Net 4	Everything about programming
20	Pro Wpf 4.5 in C#: Windows Presentation Foundation in .Net 4.5	Everything about programming

- б) Запрос, в котором нужно проверить цену книги. В результирующий запрос возвращается значение, соответствующее первой условию true:

```
select 'Book title' = b.NameBook,
      'Book price' = case
        when b.price < 50 then 'Book cheaper than 50'
        when b.price between 50 and 100 then
          'Price ranges between 50 and 100'
        else 'Book more expensive than 100'
      end
from books b
```

Результат:

	Book title	Book price
12	Java: The Complete Reference	Book cheaper than 50
13	C++: The Complete Reference, 4th Edition	Book cheaper than 50
14	Swing: A Beginner Guide	Book cheaper than 50
15	Programming ASP.NET Core (Developer Reference)	Book cheaper than 50
16	Programming Microsoft ASP.NET MVC (3rd Edition)	Book cheaper than 50
17	ASP.Net: The Complete Reference	Book cheaper than 50
18	Creating a Web Site: The Missing Manual	Book cheaper than 50
19	Pro WPF in C# 2010: Windows Presentation Foundation in .Net 4	Price ranges between 50 and 100
20	Pro Wpf 4.5 in C#: Windows Presentation Foundation in .Net 4.5	Book cheaper than 50
21	The Invisible Man	Book cheaper than 50
22	How to become a Hacker	Price ranges between 50 and 100
23	Microsoft SQL Server 2005 Complete Handbook	Book more expensive than 100

Иногда возникает ситуация, когда необходимо использовать оператор CASE для проверки на **IS NOT NULL**. В результате структура case принимает вид:

```
case
when значение1 IS NOT NULL then выражение1
when значение2 IS NOT NULL then выражение2
[, ... n]
[else результирующее_значениеN]
End
```

В таком случае рекомендуется использовать функцию **COALESCE**, которая служит для получения значений не равных NULL.

Например, в таблице Books вместо одного поля цены у нас существует два поля: оптовая и розничная цена. Нам необходимо узнать стоимость каждой книги:

```
-- с оператором case
select 'Book title' = NameBook,
      'Price' = case
        when TradePrice IS NOT NULL then TradePrice *
Quantity
        when RetailPrice IS NOT NULL then RetailPrice *
Quantity
      end
from book.Books;

-- с использованием функции coalesce
select 'Book title' = NameBook,
      'Price' = coalesce (TradePrice * Quantity,
RetailPrice* Quantity)
from book.Books;
```

Идем дальше. Оператор case может вернуть NULL, если сравниваемые выражения являются одинаковыми, иначе он возвращает первое значение. В таком случае структура данного оператора приобретает следующий вид:


```

case
when значение1=значениеX then NULL
when значение2=значениеY then NULL
[,... n]
[else результирующее_значениеN]
End

```

Чтобы упростить работу следует воспользоваться функцией **NULLIF**. Предположим, что нам необходимо вывести на экран названия книг и их тираж. Если значение тиража отсутствует (т.е. равно нулю), тогда выводим NULL.

```

-- с оператором case
select 'Books' = NameBook,
       'Pressrun' = case
                     when Quantity=0 then NULL
                     else Quantity
                     end
from books;

-- с использованием функции nullif
select 'Books' = NameBook,
       'Pressrun' = NULLIF(Quantity,0)
from books;

```

Результат:

Results		Messages
	Books	Pressrun
13	C++: The Complete Reference, 4th Edition	10
14	Swing: A Beginner Guide	15
15	Programming ASP.NET Core (Developer Reference)	15
16	Programming Microsoft ASP.NET MVC (3rd Edition)	15
17	ASP.Net: The Complete Reference	10
18	Creating a Web Site: The Missing Manual	10
19	Pro WPF in C# 2010: Windows Presentation Foundation in .Net 4	10
20	Pro Wpf 4.5 in C#: Windows Presentation Foundation in .Net 4.5	10
21	The Invisible Man	10
22	How to become a Hacker	NULL
23	Microsoft SQL Server 2005 Complete Handbook	20

А теперь подсчитаем сколько книг имеют значение тиража.

```
select 'Number of books with not zero quantity' =  
count(NULLIF(Quantity,0))  
from books;
```

Результат:

Results		Messages	
		Number of books with not zero quantity	
1		22	

- 11. Оператор безусловного перехода GOTO.** Данный оператор передает выполнение оператору, который идет после метки, которая на него указывает. В SQL Server метки являются невыполнимыми операторами и имеют следующий синтаксис:

имя_метки:

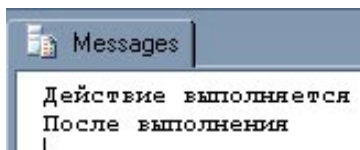
Сама команда GOTO имеет простой синтаксис:

GOTO имя_метки

GOTO как всегда является нежелательной для использования командой, поскольку код с его применением становится трудным для восприятия и анализа. Напишем пример использования оператора GOTO для обработки ошибок.

```
PRINT 'Действие выполняется'  
GOTO label  
PRINT 'Действие не выполняется'  
  
label:  
PRINT 'После выполнения'
```

Результат:



Но не следует забывать, что в данном случае намного гибче будет применить инструкцию `try..catch` или использовать механизм обработки транзакций (об этом позже).

12. T-SQL также поддерживает циклы, которые являются последовательностью действий, которые могут выполняться несколько раз подряд. Циклы представляются с помощью единого оператора цикла с предусловием **WHILE**:

```
while [( ) условие ( )]
    операторы;
```

В циклах также позволяет использование операторов **break** и **continue**. Оператор **BREAK** приводит к выходу из цикла. После этого выполнение продолжается с оператором, следующего после оператора **END** (указывает на конец блока цикла). Оператор **CONTINUE** используется, если необходимо перейти на начало цикла, и начать всю работу сначала. Кстати, в большинстве случаев эти два оператора используются в пределах условного оператора **if**.

Напишем несколько примеров использования цикла:

```
-- создаем переменную
declare @i int
set @i = 1
```

```
-- запускаем цикл
while @i<10
begin
    print @i
    set @i = @i + 1
    if @i > 5
        break
end
```

Результат:



А теперь обсчитаем среднюю цену всех книг. Если она меньше 200 грн., то все цены повысить на 10% до тех пор, пока средняя цена не станет больше 200 грн.

```
while (select avg(price)
       from book.Books) < 200
begin
    update book.Books
    set price = price* 1.1
end
```

13. Общие табличные выражения (Common Table Expressions, CTE)

позволяют задавать временный именуемый набор данных, функционально похожий на представления и доступный в рамках пакета. В связи с этим их еще

называют виртуальными представлениями. Синтаксис их создания:

```
WITH имя_представления [ ( название_поля [ ,...n ] ) ]
AS ( подзапрос )
```

При этом список полей при объявлении виртуального представления должен соответствовать количеству полей в подзапросе. В подзапросе CTE не могут использоваться операторы:

- COMPUTE или COMPUTE BY;
- ORDER BY (за исключением случаев, когда используется инструкция TOP);
- INTO;
- FOR XML;
- FOR BROWSE.

Приведем пример. Необходимо выбрать все книги, цена которых больше средней цены на книги отдельного автора.

```
-- с использованием виртуального представления
with AvgPrice(id_author, NameAuthor, Price)
as
(select a.id, a.LastName + ' ' + a.FirstName, avg(b.price)
from books b, authors a
where b.id_author = a.id
group by a.id, a.LastName + ' ' + a.FirstName)
select b.NameBook, b.price
from books as b, AvgPrice as p
where b.id_author = p.id_author and
      b.price > p.price;
```

```
-- с подзапросом
select b.NameBook, b.Price
from books b,
    (select a.id as Id_Author, a.LastName + ' ' + a.FirstName
    as AuthorName,
    avg(b.price) as Price
    from books b, authors a
    where b.id_author = a.id
    group by a.id, a.LastName + ' ' + a.FirstName) as p
where b.id_author = p.id_author and
    b.price > p.price;
```

Результат:

Results		Messages
	NameBook	Price
1	Ring Around the Sun	25
2	Time is the Simplest Thing	27
3	Way Station	25
4	Applied Microsoft .NET Framework Programming	69
5	The Art of Computer Programming, vol.3	50
6	Java: The Complete Reference	37
7	C++: The Complete Reference, 4th Edition	40
8	Swing: A Beginner Guide	34
9	Programming ASP.NET Core (Developer Reference)	45
10	Creating a Web Site: The Missing Manual	46
11	Pro WPF in C# 2010: Windows Presentation Foundation in .Net 4	51
12	Pro Wpf 4.5 in C#: Windows Presentation Foundation in .Net 4.5	48

Как видно из примера, виртуальные представления и вложенные запросы работают одинаково и дают аналогичный результат. Но в случае повторного использования такого запроса, например, в хранимой процедуре, уменьшить количество SQL кода поможет именно виртуальное представление.

Рассмотрим еще один пример. Напишем виртуальное представление, которое отображает информацию о количестве книг отдельного автора.

Одним из основных преимуществ виртуальных представлений, является использование **рекурсивных выражений**. Использование таких представлений очень полезно, если необходимо отразить данные в виде иерархии. Например, можно показать родство дочерних компаний или же зависимость работников от их руководителей и тому подобное. Общий принцип построения рекурсивных виртуальных представлений следующий:

```
WITH имя_представления [ ( название_поля [ , ...n ] ) ]
AS
(
    SELECT ... -- Начальная выборка
    UNION ALL -- Объединение результатов
    SELECT ... -- Выборка, которая определяет шаг рекурсии
    INNER JOIN СТЕ.ID = Таблица.ID -- сопоставление таблиц
)
```

В рекурсивных виртуальных представлениях **запрещается** использование следующих операторов:

- SELECT DISTINCT;
- GROUP BY;
- HAVING;
- Функции агрегирования;
- TOP;
- LEFT, RIGHT, OUTER JOIN (INNER JOIN допускается)
- Вложенные запросы.

Например, выведем в виде иерархического списка перечень тематик и книг, которые им принадлежат и издаются издательством.

```
WITH Reports(ID_THEME, ID_BOOK, Level_) AS
(
    SELECT ID_THEME, ID_BOOK, 1 AS Level_
    FROM book.Books
    WHERE ID_THEME IS NULL
    UNION ALL
    SELECT b.ID_THEME, b.ID_BOOK, Level_ + 1
    FROM book.Books b INNER JOIN Reports r
        ON b.ID_THEME = r.ID_BOOK
)

select *
from Reports
order by 1
```

Результат, к сожалению, не будет достаточно наглядный, поскольку у нас не существует принципа подчиненности тематик. Поэтому уровень у всех книг будет равен единице.

Без использования виртуального представления, для достижения аналогичного результата придется написать гораздо более сложный запрос. Кроме того, рекурсивные виртуальные представления производительнее, чем временные таблицы.

3. Транзакции в MS SQL Server

Все данные, которые хранятся в базе данных, должны быть корректными и задача разработчиков это обеспечить. Основным механизмом, который обеспечивает такую согласованность данных на программном уровне, являются транзакции. **Транзакция** – это группа последовательных операций, которые логически выполняются как одно целое. Фактически любая последовательность операторов или оператор, которые выполняется в базе данных, рассматривается как транзакция и регистрируется в журнале транзакций. Транзакции могут быть очень полезными при тестировании кода, который вносит изменения в базу данных.

Давайте рассмотрим более подробно данное определение. Операции, о которых идет речь – это INSERT / UPDATE / DELETE, SELECT и др. Если транзакция объединяет какие-либо операции в единый блок, то говорят, что эти действия выполняются в контексте данной транзакции. Следует также отметить, что после запуска транзакции все внесенные изменения будут по умолчанию видны только в вашем соединении. Для пользователей, которые просматривают данные в других соединениях, эти изменения будут невидимы. Далее вы или подтверждаете транзакцию, сохраняя все изменения в базе данных, или делаете откат транзакции, возвращая тем самым все данные в их прежнее состояние (до начала транзакции).

Каждая транзакция должна обладать следующими 4 свойствами:

1. **Atomicity (атомарности).** Гарантирует, что ни одна транзакция не будет зафиксирована в системе частично. То есть операторы, которые входят в транзакции могут быть либо выполнены все и полностью или не исполнен ни один из них. Частичное выполнение транзакций не допускается.
2. **Consistency (целостности)** указывает на то, что система находится в согласованном состоянии, как до начала транзакции, так и после ее завершения, а это в свою очередь не нарушает бизнес-логику и отношения между объектами базы данных. Это свойство очень важно при разработке клиент-серверных приложений, поскольку в базе данных происходит большое количество транзакций для различных объектов базы от разных клиентов. И если хотя бы одна из транзакций нарушает целостность данных, то все остальные могут выдать неверные результаты.
3. **Isolation (изолированности),** то есть транзакция не взаимодействует и не конфликтует с другими транзакциями в базе данных. Это включает и то, что во время выполнения транзакции другие процессы не должны видеть данные в промежуточном состоянии.
4. **Durability (долговременности или надежности)** указывает на гарантированность выполнения всех действий, независимо от внешних событий (сбой в системе, "падение" сервера и т.д.).

В MS SQL Server выделяют следующие **типы транзакций**:

- явные;
- неявные;
- автоматические.

Явные транзакции – это транзакции, которые объявляются в программе направления. Для определения начала и конца транзакции используются **такие операторы**:

- `BEGIN TRAN [SACTION] [имя_транзакции]` – определяет начало транзакции.
- `COMMIT TRAN [SACTION] [имя_транзакции]` – сообщает сервер, что транзакция закончилась и нужно сохранить (зафиксировать) все изменения. Можно указывать имя транзакции для подтверждения только ее действий.
- `COMMIT WORK` – аналогично предыдущему оператору, но имя транзакции не указывается.
- `ROLLBACK TRAN [SACTION] [имя_транзакции] | имя_точки_сохранения]` – отмена всех действий текущей транзакции, или транзакции с определенным именем, или ДО точки сохранения. Если в программе существует точка сохранения, то все изменения, сделанные в базе данных ТОЛЬКО до этой точки, можно отменить.
- `ROLLBACK WORK` – аналогично предыдущему оператору, но имя транзакции не указывается.
- `SAVE TRAN [SACTION] [имя_точки_сохранения]` – позволяет установить точку сохранения.

Рассмотрим все по порядку. Допустим, у нас существует три действия, которые мы хотим объединить в одну

транзакцию, причем определить ее нужно явно. В таком случае код будет иметь следующий вид:

```
begin transaction    -- начало транзакции
-- 1
select distinct FirstName
from book.Authors;
-- 2
insert into book.Themes (NameTheme)
values ('MFC')

-- 3
update book.Authors
set id_country = (select id_country
                  for global.Country
                  where NameCountry = 'USA')

commit transaction -- подтвердите выполнение транзакции
- Rollback transaction - отменить выполнение транзакции
```

Операторы Rollback существуют для отмены выполнения действий, определенных в рамках транзакции. То есть любые изменения, сделанные в базе данных для данного оператора, отменяются. Причем существует возможность создания точки сохранения, и тогда можно отменить только те действия, которые были осуществлены после нее. Точка сохранения создается с помощью оператора **Save Transaction**. В пределах одной транзакции может существовать несколько точек сохранения, а также допускается наличие нескольких точек с одинаковыми именами. В случае существования точек с одинаковыми именами, отмена операций осуществляется до последней точки (от начала транзакции), то есть точки с аналогичными именами, расположенные выше будут игнорироваться.

К примеру:

```
begin transaction    -- начало транзакции
-- 1
select distinct FirstName
from book.Authors;
save transaction pt1  -- первая точка сохранения
-- 2
insert into book.Themes (NameTheme)
values ('MFC')
save transaction pt2  -- вторая точка сохранения
-- 3
update book.Authors
set id_country = (select id_country
                  for global.Country
                  where NameCountry = 'USA')
rollback transaction pt2 -- отменить выполнение
транзакции до точки pt2 (до update)
--rollback transaction pt1 - отменить выполнение
транзакции до точки pt1 (до insert)
```

Или:

```
begin transaction    -- начало транзакции
-- 1
select distinct FirstName
from book.Authors;
save transaction pt1  -- первая точка сохранения
-- 2
insert into book.Themes (NameTheme)
values ('MFC')
save transaction pt1  -- вторая точка сохранения
(с аналогичным именем)
-- 3
update book.Authors
set id_country = (select id_country
                  for global.Country
                  where NameCountry = 'USA')
rollback transaction pt1 -- отменить выполнение
транзакции до точки pt 1.
-- Отмена состоится до оператора update
```

В SQL Server существует ряд глобальных системных переменных, среди которых есть и полезные в работе с транзакциями. Наиболее часто используемыми являются:

- глобальная системная переменная @@ error, которая содержит результат выполнения транзакции. Если транзакция завершилась успешно, то она содержит 0, иначе – код ошибки.
- глобальная системная переменная @@ tranccount, которая является счетчиком транзакций. Работает она следующим образом:
 - при вызове оператора begin transaction значение этой переменной увеличивается на 1.
 - оператор save transaction на ее значение не влияет
 - оператор rollback transaction влияет двояко. Если имя транзакции не указано, то значение переменной обнуляется, иначе значение уменьшается на 1.

Приведем маленький пример того, как мы можем манипулировать транзакциями, используя значение первой глобальной системной переменной.

```
begin transaction    -- начало транзакции
-- 1
select distinct FirstName
from book.Authors;

save transaction pt1 -- первая точка сохранения
-- 2
insert into book.Themes (NameTheme)
values ('MFC')
save transaction pt2 -- вторая точка сохранения
```

```
-- 3
update book.Authors
set id_country = (select id_country
                  for global.Country
                  where NameCountry = 'USA')
-- выбор действия в зависимости от текущего состояния
-- ошибки
if(@@error >=1 and @@error <= 10)
begin
    print 'Значение ошибки 1..10'
    rollback transaction pt2
end
else if(@@error > 10)
    rollback transaction
else
    commit transaction
```

Для более гибкой обработки ошибок в T-SQL можно использовать инструкцию **try ... catch**. В блоке try (защищенный блок) размещается код, который может генерировать исключения (ошибки). В случае возникновения ошибки, обработка немедленно приостанавливается, все инструкции try-блока, которые остались, игнорируются и управление передается catch-блоку, который идет за ним. Итак, в catch-блок передается управление, если сгенерировано исключение.

Синтаксис инструкции try ... catch имеет следующий вид:

```
begin try
    -- блок кода, который проверяется на ошибки
end try
begin catch
    -- обработчик исключения
end catch
```

После попадания в блок `catch`, вы можете с помощью системных функций выявить причину ошибки или получить подробную информацию о ней. Самые распространенные функции:

- `ERROR_NUMBER` – номер ошибки;
- `ERROR_MESSAGE` – текст сообщения об ошибке;
- `ERROR_LINE` – номер строки, в которой содержится ошибка;
- `ERROR_SEVERITY` – важность сообщения об ошибке;
- `ERROR_STATE` – номер состояния об ошибке.

В явных транзакциях **запрещается одновременно использовать** следующие операторы:

- `ALTER`, `DROP`, `RESTORE` и `CREATE DATABASE`;
- `BACKUP` и `RESTORE LOG`;
- `RECONFIGURE`;
- `UPDATE STATISTICS`.

Неявные транзакции включены в T-SQL для совместимости со стандартом ANSI. Когда включается режим неявных транзакций, автоматически выполняется оператор `begin transaction`. Завершить транзакцию можно только путем явного вызова оператора `commit` или `rollback transaction`.

Включить этот режим можно с помощью оператора **`set implicit_transaction`**:

```
set implicit_transaction [on          -- включить
                          | off]     -- выключить
```

При этом до конца сеанса следующие операции должны обязательно быть зафиксированы или отменены:

- `ALTER`, `TRUNCATE TABLE`;
- все операции `CREATE` и `DROP`;

- SELECT;
- GRANT и REVOKE;
- INSERT, DELETE, UPDATE;
- FETCH;
- OPEN.

К примеру:

```
set implicit_transaction on
select distinct FirstName
from book.Authors;
commit transaction
```

Следует также помнить, что использование неявных транзакций не рекомендуется, так как каждую транзакцию нужно завершить или отменить явно. Если этого не сделать, то транзакция будет открыта и данные будут надолго заблокированы.

Автоматические транзакции – это транзакции, которые выполняются, если даже операторы для работы с транзакциями не прописаны в коде явно. То есть любые изменения данных сервером расцениваются как транзакция. Проще говоря, автоматические транзакции осуществляются без явных рамок и с использованием оператора **GO**, которые посылают пакет данных для обработки на сервер.

4. Хранимые процедуры

Хранимые процедуры (stored procedures) – это последовательность компилируемых операторов, хранящаяся в базе данных. Следует отметить, что код хранимых процедур компилируется при первом запуске и дальше сохраняется в откомпилированном виде, поэтому их эффективность намного выше, чем в обычных запросах. Хранимые процедуры являются основным интерфейсом, который должен использоваться прикладными приложениями для обращения к произвольным данным в базе данных. Кроме управления доступом к базе данных, они также позволяют изолировать код базы данных. Теперь не нужно писать SQL команды для осуществления определенных изменений, а также это гарантирует безопасность между пользователями и таблицами в базе.

В SQL Server существует набор системных хранимых процедур, которые начинаются с префикса **sp_xxx (stored procedure)**, а также можно создавать свои собственные пользовательские хранимые процедуры. Пользовательские хранимые процедуры создаются с помощью оператора **CREATE PROCEDURE** и они могут содержать почти произвольные команды.

```

CREATE PROC[EDURE] [схема.] имя_процедуры [;число]
[@параметр [схема.] тип
    [VARYING] /*для управления курсором*/
    [ = значение_по_умолчанию | NULL]
    [OUT | OUTPUT] /*указывает на то,
что данный параметр является возвратным */
    [READONLY] /*только для чтения.
Для табличных типов */
]
[,...n]
[WITH { RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION
    | EXECUTE AS { CALLER | SELF | OWNER | 'логин_
пользователя' } } ]
[FOR REPLICATION] /* принимает участие в репликации */
AS
{
    [BEGIN]
        тело_процедуры с оператором SELECT
    [END]
    | EXTERNAL NAME сборка.класс.метод
}

```

Хранимые процедуры могут как принимать данные (input parameters), так и возвращать (output parameters). В процедуре также допускается использование локальных переменных. С параметром **ENCRYPTION** вы уже знакомы. Он используется для создания шифрованных хранимых процедур. Если указать параметр **RECOMPILE** SQL Server перекомпилирует процедуру при каждом ее запуске. Более подробно с этим параметром мы познакомимся позже. Параметр **EXECUTE AS** определяет контекст безопасности для хранимой процедуры:

- **CALLER** (по умолчанию) – указывает, что инструкции, содержащиеся в процедуре, выполняются в контексте пользователя, который ее вызвал.

- **"логин_пользователя"** – указывается, какой именно пользователь может изменять хранимую процедуру.
- **SELF** – инструкции выполняются в контексте пользователя, который создал хранимую процедуру или может ее менять.
- **OWNER** – все инструкции выполняются в контексте текущего владельца этой хранимой процедуры. Если владелец процедуры не указан, тогда подразумевают владельца схемы.

Параметр EXTERNAL NAME указывает на сборку .NET Framework, на которую должна ссылаться хранимая процедура CLR. При этом следует указать название класса в этой сборке и необходимый метод (должен быть статический). Если имя класса включает в себя название пространства имен, отделенную точками (.), тогда его следует отделять квадратными скобками ([]) или двойными кавычками (" ").

Вызвать процедуру можно с помощью оператора **execute**, упрощенный синтаксис которого следующий:

```
exec[ute] имя_процедуры [:число] [список_параметров]
[WITH RECOMPILE]
```

Более подробно каждый из параметров рассмотрим на практике, но для начала еще немного теории.

Итак, при создании хранимой процедуры следует придерживаться следующих правил:

- в хранимых процедурах нельзя использовать операторы:
 - CREATE RULE,
 - CREATE DEFAULT,
 - CREATE PROCEDURE,

- CREATE TRIGGER,
 - CREATE VIEW,
 - USE база_данных,
 - SET SHOWPLAN_TEXT,
 - SET SHOWPLAN_ALL;
- при выполнении процедуры все объекты, на которые она ссылается, должны присутствовать в БД. Специальное свойство процедуры (позднее связывание имени) позволяет во время компиляции ссылаться на несуществующий объект. Благодаря этому хранимая процедура при создании может генерировать временные объекты, а затем ссылаться на них при запуске;
 - в процедуре нельзя создавать объект, а затем удалить его или создавать заново под одним и тем же именем;
 - процедура не может иметь более 1024 параметров;
 - можно создавать вложенные процедуры (поддерживается до 32 уровней вложенности)
 - как и в случае представлений, если в процедуре используется оператор SELECT * и в базовую таблицу добавляются поля после создания процедуры, то при ее выполнении эти новые поля использовать нельзя. Для этого нужно с помощью оператора ALTER PROCEDURE изменить хранимую процедуру.

Процесс выполнения хранимых процедур проходит **5 этапов:**

1. Лексикографический анализатор выражений разбивает процедуру на отдельные компоненты.
2. Компоненты, которые ссылаются на объекты БД (таблицы, представления и т.д.), сопоставляются с этими объектами (уже проверенными на существование). Этот процесс называется **расширением ссылок**.
3. Хранимая процедура регистрируется, то есть в sysobjects записывается ее название, а в syscomments – код создания.
4. Создается предварительный план выполнения запроса, то есть дерево запроса, которое сохраняется в системной таблице sysprocedures.
5. Считывается дерево запроса и процедура выполняется.

Среди преимуществ хранимых процедур можно выделить то, что план процедуры сберегается в процедурном кэше после ее первого исполнения, и уже в дальнейшем оттуда просто считывается. То есть процедура компилируется один раз, при первом ее вызове. Это приводит к повышению производительности и скорости выполнения хранимых процедур. Кроме того, существует возможность автоматического выполнения процедур при запуске SQL Server.

Перейдем к практике. Для начала напомним простую процедуру, которая позволяет просмотреть список авторов и количество их книг:

```

create procedure sp_authors
as
select a.firstname + ' ' + a.lastname, count(b.id_book) as
countBooks
from book.Authors a, book.Books b
where b.id_author = a.id_author
group by a.firstname + ' ' + a.lastname;
go
exec sp_authors;

```

Результат:

Results		Messages
	(No column name)	BooksCount
1	Esposito Dino	2
2	Knuth Donald	3
3	MacDonald Matthew	4
4	Richter Jeffrey	3
5	Schildt Herbert	4
6	Simak Clifford	4
7	Waymire Richard	1
8	Wells Herbert	1

А теперь рассмотрим как используется параметр "число" при создании процедуры. Как правило, он служит для создания группы хранимых процедур. Это может пригодиться, когда необходимо, чтобы несколько процедур выполнялись как одна, то есть одновременно. Например, напомним группу из двух процедур, которые получают различную информацию об авторах:

```

create procedure sp_grAuthors;1
as
select *
from books;
go
create procedure sp_grAuthors;2
as
select a.LastName + ' ' + a.FirstName, count(b.id) as
BooksCount
from Authors a, Books b
where b.id_author = a.id
group by a.LastName + ' ' + a.FirstName;

```

Даже физически группа запросов сохраняется как одна:



Запускаем:

```

exec sp_grAuthors;1 --executes the first procedure from
the group
exec sp_grAuthors --executes the first procedure from the
group too
exec sp_grAuthors;2 --executes the second procedure from
the group

```

Результат:

Results		Messages
	(No column name)	BooksCount
1	Esposito Dino	2
2	Knuth Donald	3
3	MacDonald Matthew	4
4	Richter Jeffrey	3
5	Schildt Herbert	4
6	Simak Clifford	4
7	Waymire Richard	1
8	Wells Herbert	1

Для удаления хранимой процедуры используется оператор **DROP PROCEDURE**, а в случае группы процедур – удаляется целая группа. Удаление отдельной хранимой процедуры из группы невозможно. Синтаксис данного оператора типичный, поэтому рассматривать подробно его мы не будем.

Рассмотрим пример передачи параметров в хранимую процедуру. Для этого напишем хранимую процедуру, которая добавляет два числа, переданных в качестве параметров, а результат записывает в выходной (output) параметр:

```
create procedure sp_summa
@a int,
@b int,
@res int output
as
set @res = @a + @b
```

Передавать при вызове параметры в хранимую процедуру можно двумя способами: явно и по позиции.

```

declare @summ int                -- объявляем
переменную, которая будет содержать результат
execute sp_summa @a = 5, @b = 25, @res = @summ output --
явная передача параметров
execute sp_summa 5, 25, @summ output -- передача
параметров по позиции
select '5 + 25 = ', @summ        -- выводим
результат

```

Результат:

Results		Messages
	(No column name)	(No column name)
1	5 + 25 =	30

Возвращать значения из процедуры можно, используя оператор **return**. Для этого перепишем нашу процедуру следующим образом:

```

create procedure sp_summa2
@a int,
@b int
as
declare @res int
set @res = @a + @b
return @res

```

Вызов такой хранимой процедуры будет следующим:

```

declare @summ int                -- объявляем
переменную, которая будет содержать результат
execute @summ = sp_summa2 @a = 5, @b = 25 -- явная
передача параметров
execute @summ = sp_summa2 5, 25        -- передача
параметров по позиции
select '5 + 25 = ', @summ            -- выводим
результат

```

Рассмотрим более сложный пример. Напишем процедуру, которая возвращает список авторов, которые живут в США:

```
create procedure sp_ListAuthors
@name varchar(25) output,
@surname varchar(25) output
as
select @name=a.FirstName, @surname=a.LastName
from Authors a, Countries c
where a.id_country=c.id and
      c.country='USA'
go
declare @name varchar(25), @surname varchar(25)
exec sp_ListAuthors @name output, @surname output
select 'List of Authors:', @name + ' ' + @surname
```

Результат:

Results		Messages	
	(No column name)		(No column name)
1	List of Authors:		Clifford Simak

Осталось рассмотреть опцию **RECOMPILE**. При ее использовании SQL Server будет игнорировать существующий план выполнения хранимой процедуры и при каждом ее выполнении создавать новый. На практике перекомпиляция хранимой процедуры используется очень редко, но она пригодится, например, при добавлении нового индекса, который улучшает работу хранимой процедуры. Опцию RECOMPILE можно использовать в двух случаях:

1. В операторе **CREATE PROC**, то есть при создании процедуры. В таком случае план выполнения процедуры не должен храниться в процедурном кэше и при

выполнении она будет заново перекомпилироваться. Это полезно для процедур с текущими параметрами. К примеру:

```
create proc sp_themes  
with recompile  
as  
select *  
from book.Themes
```

2. В операторе **EXEC PROC** – при вызове процедуры. В таком случае перекомпиляция осуществляется в текущем сеансе выполнения процедуры. Новый план сохраняется в кэше и потом может еще использоваться:

```
exec sp_themes with recompile
```

Чтобы перекомпилировать все хранимые процедуры и триггеры используется системная процедура **sp_recompile**, а для автоматического выполнения хранимой процедуры при запуске сервера следует выполнить процедуру **sp_procoption**.

Для того, чтобы просмотреть информацию про хранимую процедуру, то есть код ее создания, нужно вызвать системную процедуру **sp_helptext**:

```
exec sp_helptext sp_addtype
```

Результат:

Results		Messages	
	Text		
1	create procedure sys.sp_addtype		
2	@typename sysname, -- name of user-defined type		
3	@phystype sysname, -- physical system type of user-defined type		
4	@nulltype varchar(8) = null,-- nullability of new type		
5	@owner sysname = null,-- Owner of type (ignored)		

Для просмотра связанных с процедурой объектов, следует воспользоваться системной процедурой **sp_depends**:

```
exec sp_depends имя_процедуры
```

5. Пользовательские функции

В SQL Server существует большой набор стандартных функций, которыми вы уже не раз пользовались. И этого набора для обеспечения функциональности базы данных иногда может быть недостаточно. Поэтому SQL Server предоставляет возможность создавать свои собственные пользовательские функции (user-defined functions).

Типы пользовательских функций:

1. скалярные (scalar functions) – это функции, которые возвращают одно скалярное значение, то есть число, строка и т. п.
2. встроенные однотабличные или подставляемые табличные (inline table-valued functions) – это функции, которые возвращают результат в виде таблицы. Возвращаются они одним оператором SELECT. Причем, если в результате создается таблица, то имена ее полей являются псевдонимами полей при выборке данных.
3. многооператорные функции (multistatement table-valued functions) – это функции, при определении которой задаются новые имена полей и типы.

Кроме того функции разделяют по детерминизму. Детерминизм функции определяется постоянством ее результатов. Функция является детерминированной (deterministic function), если при одном и том же заданном входном значении она всегда возвращает один

и тот же результат. Например, встроенная функция DATEADD () является детерминированной, поскольку добавление трех дней к дате 5 мая 2010 г. Всегда дает дату 8 мая 2010, или функция COS(), которая возвращает косинус указанного угла.

Функция является недетерминированной (non-deterministic function), если она может возвращать разные значения при одном и том же заданном входном значении. Например, встроенная функция GETDATE() является недетерминированной, поскольку при каждом вызове она возвращает разные значения.

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

От детерминированности функции зависит, можно ли проиндексировать ее результат, а также можно ли определить кластеризованный индекс на представление, которое ссылается на эту функцию. Например, недетерминированные функции не могут быть использованы для создания индексов или расчетных полей. Что касается кластеризованного индекса, то он не может быть создан для представления, если оно обращается к недетерминированной функции (независимо от того, используется она в индексе или нет).

Пользовательские функции имеют ряд преимуществ, среди которых основным является повышение производительности выполнения, поскольку функции, как и хранимые процедуры, кэшируют код и повторно используют план выполнения.

Итак, рассмотрим типы функций по порядку. Начнем со скалярных. Синтаксис их объявления следующий:

```
CREATE FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию | default ]
      [ READONLY ]
      ] [, ... n]
    )
RETURNS тип_возвращаемого_значения
[ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
      }
  [ , ...n ]
]
[ AS ]
BEGIN
    тело_функции
RETURN (возвращаемое_скалярное_значение)
END
```

Как видно из синтаксиса, после имени функции, необходимо в скобках перечислить необходимые входные параметры, если они предусмотрены. Поскольку все параметры являются локальными переменными, то перед именем необходимо ставить символ @, после чего указывается его тип (*допускаются все типы данных, включая типы данных CLR, кроме timestamp (!)*). В случае необходимости можно задать значение по умолчанию для каждого из входных параметров или указать ключевое слово default. Если определено значение default, тогда параметру присваивается значение по умолчанию для данного типа.

Ключевое слово *READONLY* указывает на то, что параметр не может быть обновлен или изменен при определении функции. Заметим, что если тип параметра является пользовательским табличным типом, тогда обязательно указать ключевое слово *READONLY*.

Далее следует указать тип возвращаемого скалярного значения для функции (*returns*). Допускаются все типы данных, кроме нескаларных типов *cursor* и *table*, а также типы *rowversion* (*timestamp*), *text*, *ntext* или *image*. Их лучше заменить типами *uniqueidentifier* и *binary* (8).

Параметр *WITH* задает дополнительные характеристики для входных аргументов. С ключевым словом *ENCRPTION*, *SCHEMABINDING* и *EXECUTE AS* вы уже знакомы, поэтому на них останавливаться не будем. Хотя здесь есть несколько замечаний:

- параметр *SCHEMABINDING* нельзя указывать для функций *CLR* и функций, которые ссылаются на псевдонимы типов данных;
- параметр *EXECUTE AS* нельзя указывать для встроенных пользовательских функций.

Параметр *RETURNS* / *CALLED* может быть представлен одним из двух значений:

- *CALLED ON NULL INPUT* (по умолчанию) означает, что функция выполняется и в случаях, если в качестве аргумента передано значение *NULL*.
- *RETURNS NULL ON NULL INPUT* указанный для функций *CLR* и означает, что функция может вернуть *NULL* значения, если один из аргументов равен *NULL*. При этом код самой функции *SQL Server* не вызывает.

Тело функции размещается в середине блоков BEGIN..END, который обязательно должен содержать оператор RETURN для возврата результата. При написании кода тела функции следует помнить, что здесь существует ряд ограничений, основным из которых является запрет изменять состояние произвольного объекта базы данных или же базу данных.

Кстати, рекурсивные функции также поддерживаются. Допускается до 32 уровней вложенности.

Вызвать скалярную функцию можно одним из двух способов: с помощью оператора *select* или *execute*.

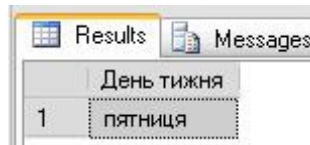
```
SELECT имя_функции (параметр1 [,... n])
EXEC[UTE] @переменная = имя_функции параметр1 [,... n]
```

Напишем функцию, которая возвращает день недели по указанной в качестве параметра дате.

```
create function DayOfWeek (@day datetime)
returns nvarchar(15)
as
begin
    declare @wday nvarchar(15)
    if (datename(dw, @day) = 'Monday')
        set @wday = 'понедельник'
    if (datename(dw, @day) = 'Friday')
        set @wday = 'пятница'
    else
        set @wday = 'другой'
    return @wday
end;

-- ВЫЗОВ
select DayOfWeek(GETDATE()) as 'День недели';
```

Результат:



Results	Messages
День тижня	
1	пятниця

Встроенные табличные функции подчиняются тем же правилам, что и скалярные. Их отличие от последних заключается в том, что они возвращают результат в виде таблицы. Табличные функции являются неплохой альтернативой представлениям и хранимым процедурам. Например, недостатком представлений является то, что они не могут принимать параметры, которые иногда необходимо передать. Хранимые процедуры в свою очередь могут принимать параметры, но не могут быть использованы в выражении *FROM* оператора *SELECT*, что несколько усложняет обработку результатов. Табличные функции решают вышеописанные проблемы.

Итак, синтаксис однотабличной функции выглядит так:

```
CREATE FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию | default ]
      [ READONLY ]
    ] [, ... n]
  )
  RETURNS TABLE
  [ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безопасности]
  }
  [ , ...n ]
]
```

```
[ AS ]
BEGIN
    тело_функции
RETURN [ ( ] оператор SELECT[ ) ]
END
```

Вызывается табличная функция следующим образом:

```
select * from имя_функции (параметр1 [, ... n])
```

Например, напомним функцию, которая выводит названия книг и количество магазинов, которые их продают.

```
create function countShops ()
returns table
as
return (select b.NameBook as 'Book Title',
        count(sh.id) as 'Number of Shops'
        from books b, shops sh, sales s
        where s.id_book = b.id and s.id_shop = sh.id
        group by b.NameBook);
go
-- call the function
select * from countShops();
```

Результат:

	Book Title	Number Of Shops
1	Applied Microsoft .NET Framework Programming	1
2	ASP .Net: The Complete Reference	1
3	CLR via C#	1
4	Java: A Beginner Guide	1
5	Java: The Complete Reference	2
6	Pro Wpf 4.5 in C#: Windows Presentation Foundati...	1
7	Programming ASP.NET Core (Developer Reference)	1
8	Ring Around the Sun	1
9	Swing: A Beginner Guide	1
10	The Art of Computer Programming, vol.1	1
11	Time is the Simplest Thing	2
12	Windows Runtime via C#	1

Синтаксис многооператорной функции:

```
CREATE FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию
    | default ]
      [ READONLY ]
      ][, ... n]
    )
RETURNS @возвращаемая_таблица
    TABLE структура_таблицы
[ WITH { [ ENCRYPTION ]
    | [ SCHEMABINDING ]
    | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
    | [ EXECUTE AS контекст_безопасности]
  }
  [ , ...n ]
]
[ AS ]
BEGIN
    тело_функции
RETURN
END
```

Отметим, что в многооператорной функции возвращаемая таблица не обязательно создается оператором `select`. Отсюда происходит и название функции. Здесь можно, например, выполнять предварительную обработку данных и создавать временную таблицу, после чего доработать ее и вернуть новую таблицу в вызывающую программу.

Следует также быть внимательным, поскольку блок тела функции может содержать несколько операторов `SELECT`. В таком случае, в выражении `RETURNS` нужно явно определить таблицу, которая будет возвращаться. Кроме того, поскольку оператор `RETURN`

в многооператорной табличной функции всегда возвращает таблицу, которая задана в RETURNS, то он должен выполняться без аргументов. Например, **RETURN**, а не **RETURN @myTable**.

Вызывается многооператорная табличная функция подобно встроенной табличной, то есть с помощью оператора SELECT:

```
select * from имя_функции (параметр1 [, ... n])
```

В качестве примера напишем многооператорную табличную функцию, которая возвращает название магазина (-ов), который продал наибольшее количество книг.

Данный процесс можно разделить на **два этапа**:

- **первый** – создадим временную таблицу, которая возвращает название книги и количество магазинов, которые их продают;
- **второй** – получаем магазин, продавший максимальное количество книг.

```
create function bestShops()
returns @tableBestShops table (ShopName varchar(30)
not null, BookCount int not null)
as
begin
-- creating a temporary table that returns the shop name
-- and number of books in the shop
declare @tmpTable table (id_book int not null,
bookNumber int not null)
insert @tmpTable
select b.id as 'Identifier', count(s.id_shop) as 'Number
Of Shops'
```

```

from books b, sales s
where s.id_book=b.id
group by b.id
insert @tableBestShops
select sh.Shop as 'Shop Name',
       'Number Of Books' = max(tt.bookNumber)
from @tmpTable tt, sales s, Shops sh
where tt.id_book=s.id_book and
      s.id_shop = sh.id
group by sh.Shop

```

```

return
end;
go
-- call the function
select * from bestShops();

```

Результат:

Results		Messages
	ShopName	BookCount
1	HashTag	2
2	Rare Books	2
3	Smith&Brown	2

Изменить существующую функцию пользователя можно с помощью оператора **ALTER FUNCTION**:

```

-- скалярная функция
ALTER FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
      [ = значение_по_умолчанию | default ]
      [ READONLY ]

```

```

        ][,... n]
    )
    RETURNS тип_возвращаемого_значения
    [ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
    }
    [ ,...n ]
]
[ AS ]
BEGIN
    тело_функции
    RETURN (возвращаемое_скалярное_значение)
END
-- встроенная однотабличная функция
ALTER FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип
        [ = значение_по_умолчанию | default ]
        [ READONLY ]
    ][,... n]
    )
    RETURNS TABLE
    [ WITH { [ ENCRYPTION ]
        | [ SCHEMABINDING ]
        | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
        | [ EXECUTE AS контекст_безопасности]
    }
    [ ,...n ]
]
[ AS ]
BEGIN
    тело_функции
    RETURN [ ( ] оператор SELECT[ ) ]
END
-- многооператорная функция
ALTER FUNCTION [ схема. ] имя_функции
    ( [ @параметр [ AS ] [ схема. ] тип

```



```

[ = значение_по_умолчанию
| default ]
[ READONLY ]
][,... n]
)
RETURNS @возвращаемая_таблица
TABLE структура_таблицы
[ WITH { [ ENCRYPTION ]
| [ SCHEMABINDING ]
| [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
| [ EXECUTE AS контекст_безопасности]
}
[ ,...n ]
]
[ AS ]
BEGIN
тело_функции
RETURN
END

```

Несмотря на то, что с помощью оператора ALTER FUNCTION можно изменить функцию практически полностью, использовать данный оператор для преобразования скалярной функции в табличную или наоборот запрещено. С помощью ALTER FUNCTION также нельзя превращать функцию T-SQL в функцию среды CLR или наоборот.

Удаление пользовательской функции осуществляется оператором **DROP FUNCTION**:

```

DROP FUNCTION [ схема. ] имя_функции [ ,...n ]

```

Получить список пользовательских функций можно из системного представления sys.sql_modules, а список параметров каждой из них размещается в представлении sys.parameters.

Список пользовательских функций CLR размещается по другому адресу, а именно в системном представлении **`sys.assembly_modules`**.

